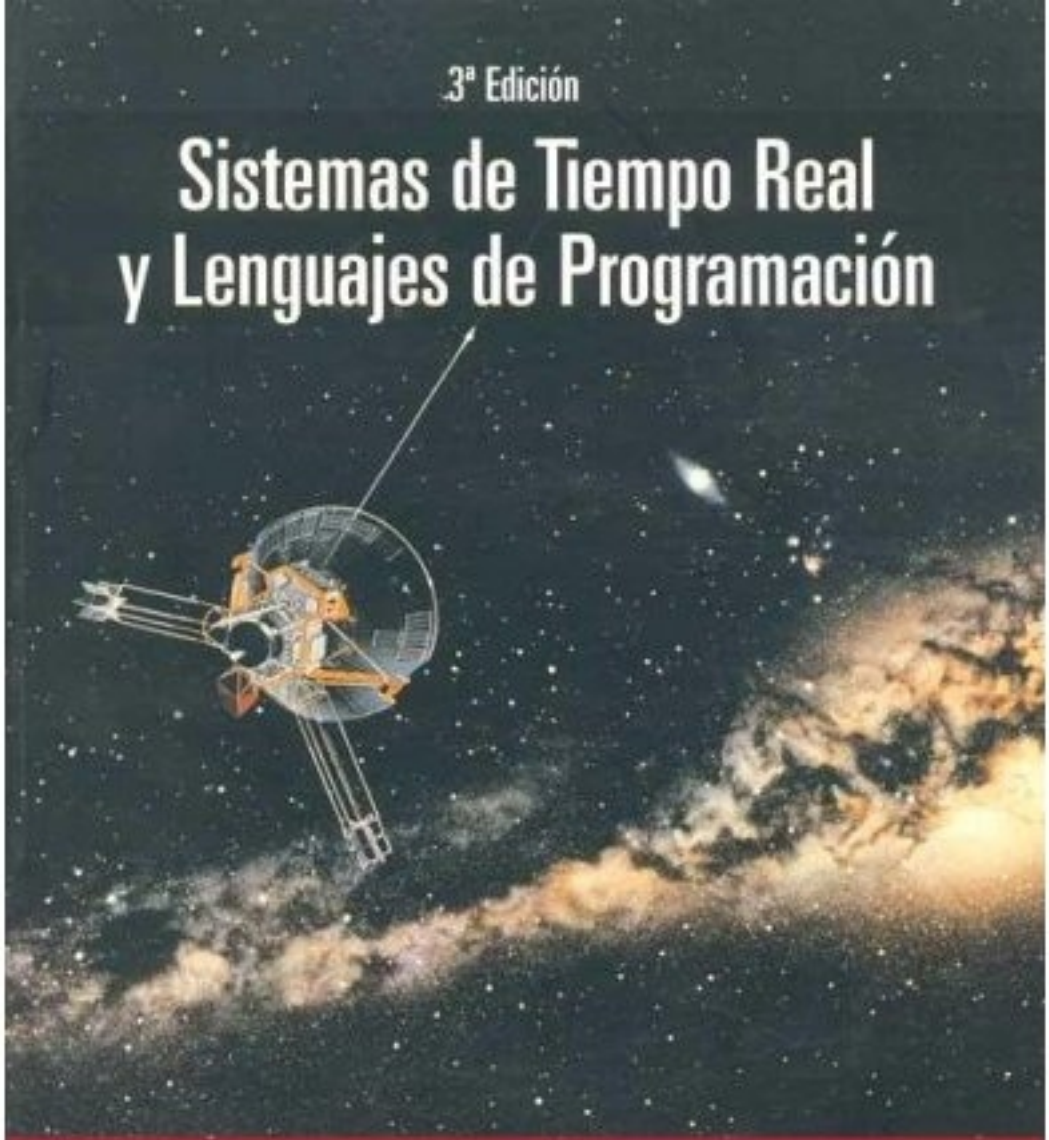
3ª Edición

Sistemas de Tiempo Real y Lenguajes de Programación



Addison Wesley

Alan Burns
Andy Wellings

Contenido

Capítulo 1

Introducción a los sistemas de tiempo real

A medida que los computadores son más pequeños, rápidos, fiables y baratos, su rango de aplicaciones se amplía. Construidos inicialmente para resolver ecuaciones, su influencia se ha extendido a todos los órdenes de la vida, desde lavadoras a control de tráfico aéreo. Una de las áreas de expansión más rápida de la explotación de computadores es aquélla que implica aplicaciones cuya función principal no es la de procesar información, pero que precisa dicho proceso de información con el fin de realizar su función principal. Una lavadora controlada por microprocesador es un buen ejemplo de dicho tipo de sistemas. En este caso, la función principal es la de lavar ropa; sin embargo, dependiendo del tipo de ropa que va a ser lavada, se deben ejecutar programas de lavado diferentes. Este tipo de aplicaciones de computador se conocen genéricamente como de **tiempo real** o **embebidas**. Se ha estimado que el 99 por ciento de la producción mundial de microprocesadores se utiliza en sistemas embebidos. Estos sistemas plantean requisitos particulares para los lenguajes de programación necesarios para programarlos, ya que tienen características diferentes de las de los sistemas de procesamiento de información tradicionales.

Este libro trata de los sistemas embebidos y de sus lenguajes de programación. Estudia las características particulares de estos sistemas, y trata de cómo han evolucionado los lenguajes de programación y los sistemas operativos de tiempo real modernos.

1.1 Definición de un sistema de tiempo real

Antes de continuar, merece la pena intentar definir de forma más precisa la expresión «sistema de tiempo real». Hay muchas interpretaciones sobre la naturaleza de un sistema de tiempo real; sin embargo, todas tienen en común la noción de tiempo de respuesta: el tiempo que precisa el sistema para generar la salida a partir de alguna entrada asociada. El *Oxford Dictionary of Computing* (*Diccionario Oxford de computación*) proporciona la siguiente definición de un sistema de tiempo real:

> Cualquier sistema en el que el tiempo en el que se produce la salida es significativo. Esto generalmente es porque la entrada corresponde a algún movimiento en el mundo físico, y la salida está relacionada con dicho movimiento. El intervalo entre el tiempo de entrada y el de salida debe ser lo suficientemente pequeño para una temporalidad aceptable.

Aquí, la palabra temporalidad se considera en el contexto del sistema total. Por ejemplo, en un sistema de guiado de misiles se requiere la salida en unos pocos milisegundos, mientras que en una línea de ensamblado de coches controlada por computador, se puede requerir la respuesta con un segundo. Para ilustrar las distintas formas de definición de los sistemas de «tiempo real» se proporcionan dos adicionales. Young (1982) define un sistema de tiempor real como:

> ... cualquier actividad o sistema de proceso de información que tiene que responder a un estímulo de entrada generado externamente en un periodo finito y especificado.

El proyecto PDCS (Predictably Dependable Computer Systems) proporciona la definición siguiente (Randell et al., 1995):

> Un sistema de tiempo real es aquél al que se le solicita que reaccione a estímulos del entorno (incluyendo el paso de tiempo físico) en intervalos del tiempo dictados por el entorno.

En su sentido más general, todas esas definiciones cubren un amplio rango de actividades de los computadores. Por ejemplo, un sistema operativo como Unix puede ser considerado de tiempo real en el sentido de que cuando un usuario introduce un comando, él o ella esperará una respuesta en pocos segundos. Afortunadamente, normalmente no es un desastre si la respuesta no se produce en ese tiempo. Estos tipos de sistemas se pueden distinguir de aquéllos otros en los que un fallo al responder puede ser considerado como una respuesta mala o incorrecta. Efectivamente, para algunos éste es el aspecto que distingue un sistema de tiempo real de otros en los que el tiempo de respuesta es importante pero no crucial. Por tanto, *la corrección de un sistema de tiempo real depende no sólo del resultado lógico de la computación, sino también del tiempo en el que se producen los resultados*. Las personas que trabajan en el campo del diseño de sistemas de tiempo real distinguen frecuentemente entre sistemas de tiempo real **estrictos** (hard) y **no estrictos** (soft). Los sistemas de tiempo real estrictos son aquéllos en los que es absolutamente imperativo que las respuestas se produzcan dentro del tiempo límite especificado. Los sistemas de tiempo real no estrictos son aquéllos en los que los tiempos de respuesta son importantes pero el sistema seguirá funcionando correctamente aunque los tiempos límite no se cumplan ocasionalmente. Los sistemas no estrictos se pueden distinguir de los interactivos, en los que no hay tiempo límite explícito. Por ejemplo, el sistema de control de vuelo de un avión de combate es un sistema de tiempo real estricto, porque un tiempo límite no cumplido puede conducir a una catástrofe, mientras que un sistema de adquisición de datos para una aplicación de control de procesos es no estricto, ya que puede estar definido para muestrear un sensor de entrada a intervalos regulares pero puede tolerar retrasos intermitentes. Naturalmente, muchos sistemas tendrán subsistemas tanto de tiempo real estricto como no estricto. Evidentemente, algunos servicios pueden tener un tiempo límite, tanto estricto como no estricto. Por ejemplo, una respuesta a algún suceso de aviso puede tener un tiempo límite *soft* de 50 ms (para una reacción óptimamente eficiente) y un tiempo estricto de 200 ms (para garantizar que no se produzcan daños sobre el equipo o el personal). Entre 50 ms y 200 ms el «valor» (o utilidad) de la salida disminuye.

Como ilustran estas definiciones y ejemplos, el uso del término *soft* no implica un tipo único de requisito, sino que incorpora un número de propiedades diferentes. Por ejemplo:

- El tiempo límite puede no alcanzarse ocasionalmente (normalmente con un límite superior de fallo en un intervalo definido).
- El servicio se puede proporcionar ocasionalmente tarde (de nuevo, con un límite superior en la tardanza).

Un tiempo límite que puede no cumplirse ocasionalmente, pero en el que no hay beneficio por la entrega retrasada, se llama **firme**. En algunos sistemas de tiempo real, se pueden proporcionar requisitos probabilísticos opcionales a los componentes firmes (por ejemplo, un servicio *hard* debe producir una salida cada 300 ms; al menos el 80 del tiempo esta salida será producida por un componente firme, *X*; en otras ocasiones se utilizará un componente *hard*, *Y*, funcionalmente mucho más sencillo.

En este libro, la expresión «sistema de tiempo real» se utiliza para referirse tanto a tiempo real estricto como no estricto. Cuando la discusión esté relacionada específicamente con sistemas de tiempo real estricto, se utilizará explícitamente el término «tiempo real estricto».

En un sistema de tiempo real, estricto o no estricto, el computador interfiere normalmente directamente con algún equipamiento físico, y se dedica a monitorizar o controlar la operación de dicho equipamiento. Una característica fundamental de todas estas aplicaciones es el papel del computador como componente de proceso de información en un sistema de ingeniería más grande. Ésta es la razón por la que tales aplicaciones se conocen como **sistemas embebidos**.

1.2 Ejemplos de sistemas de tiempo real

Una vez que se ha definido qué es significativo para los sistemas de tiempo real y embebidos, se dan algunos ejemplos de su uso.

1.2.1 Control de procesos

El primer uso de un computador como componente en un sistema más grande de ingeniería se produjo en la industria de control de procesos en los primeros años de la década de 1960. Actualmente, es norma habitual el uso de los microprocesadores. Consideremos el ejemplo sencillo mostrado en la Figura 1.1, donde el computador realiza una única actividad: garantizar un flujo estable de líquido en una tubería controlando una válvula. Si se detecta un incremento en el flujo, el computador debe responder alterando el ángulo de la válvula; esta respuesta se debe producir en un periodo finito si el equipamiento del final receptor de la tubería no está sobrecargado. Hay que indicar que la respuesta actual puede implicar una computación compleja en relación con el cálculo del nuevo ángulo de la válvula.

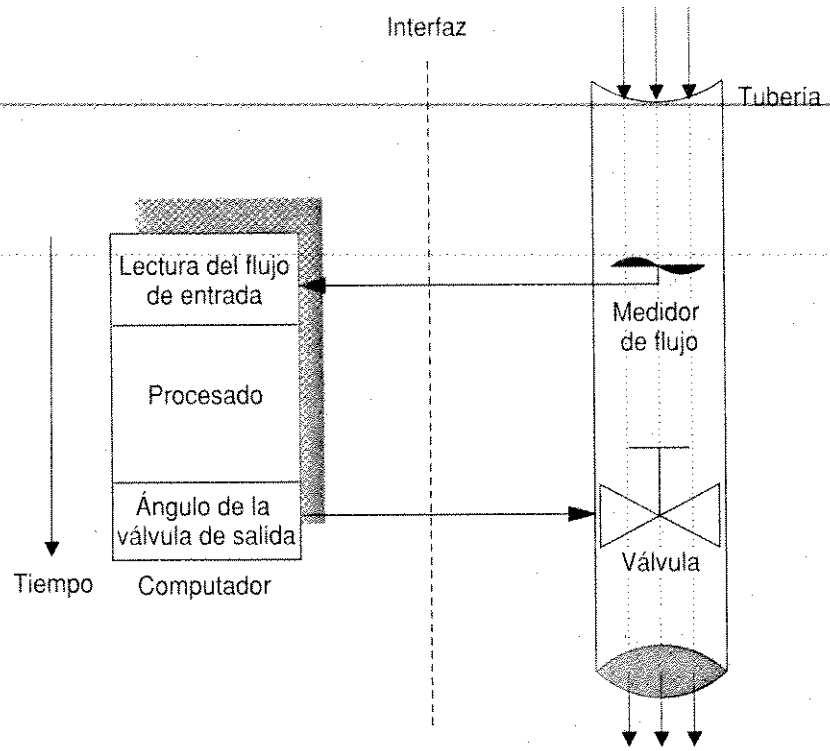


Figura 1.1. Un sistema de control de fluido.

Este ejemplo muestra concretamente un componente de un sistema de control más grande. La Figura 1.2 ilustra el papel de un computador de tiempo real embebido en un entorno completo de control de procesos. El computador interacciona con el equipamiento utilizando sensores y actuadores. Una válvula es un ejemplo de un actuador, y un transductor de presión o temperatura es un ejemplo de sensor. (Un transductor es un dispositivo que genera una señal eléctrica que es

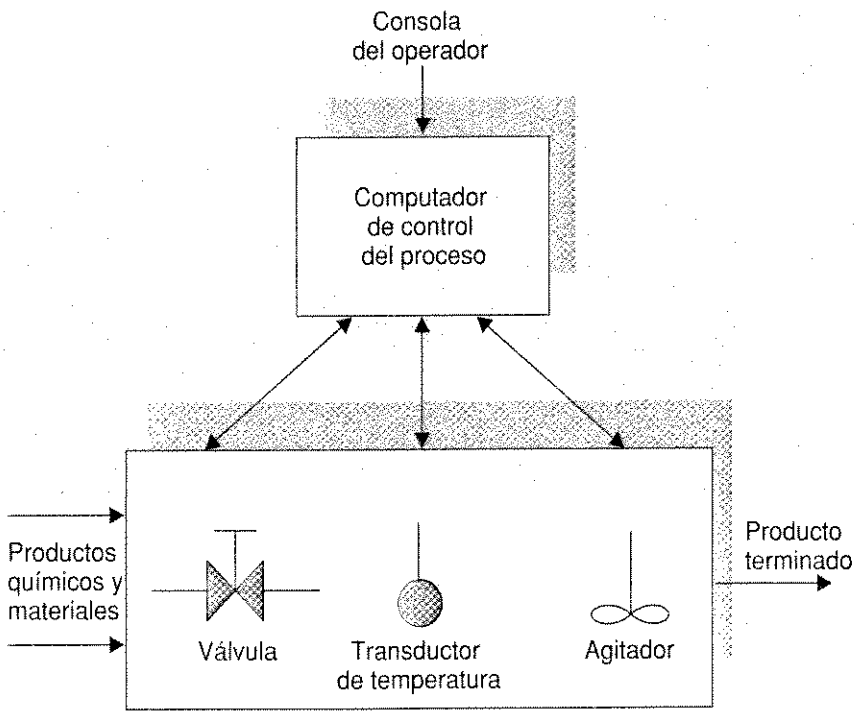


Figura 1.2. Un sistema de control de procesos.

proporcional a la cantidad física que está siendo medida.) El computador controla la operación de los sensores y actuadores para garantizar que las operaciones correctas de la planta se realizan en los tiempos apropiados. Donde sea necesario, se deben insertar convertidores analógico-digitales entre el proceso controlado y el computador.

1.2.2 Fabricación

El uso de computadores en fabricación ha llegado a ser esencial para mantener bajos los costes de producción e incrementer la productividad. Los computadores han permitido la integración del proceso completo de fabricación, desde el diseño a la propia fabricación. En este área del control de producción es donde los sistemas embebidos están mejor ilustrados. La Figura 1.3 representa esquemáticamente el papel del computador de control de producción en el proceso de fabricación. El sistema físico consta de una variedad de dispositivos mecánicos (como máquinas herramientas, manipuladores y cintas transportadoras), todos los cuales necesitan ser controlados y coordinados con el computador.

1.2.3 Comunicación, mando y control

Aunque **comunicación, mando y control** es una expresión militar, existe un amplio rango de aplicaciones dispares que exhiben características semejantes; por ejemplo, la reserva de plazas de una compañía aérea, las funcionalidades médicas para cuidado automático del paciente, el control del tráfico aéreo y la contabilidad bancaria remota. Cada uno de estos sistemas consta de un conjunto complejo de políticas, dispositivos de recogida de información y procedimientos administrativos que permiten apoyar decisiones y proporcionar los medios mediante los cuales puedan ser implementadas. A menudo, los dispositivos de recogida de información y los

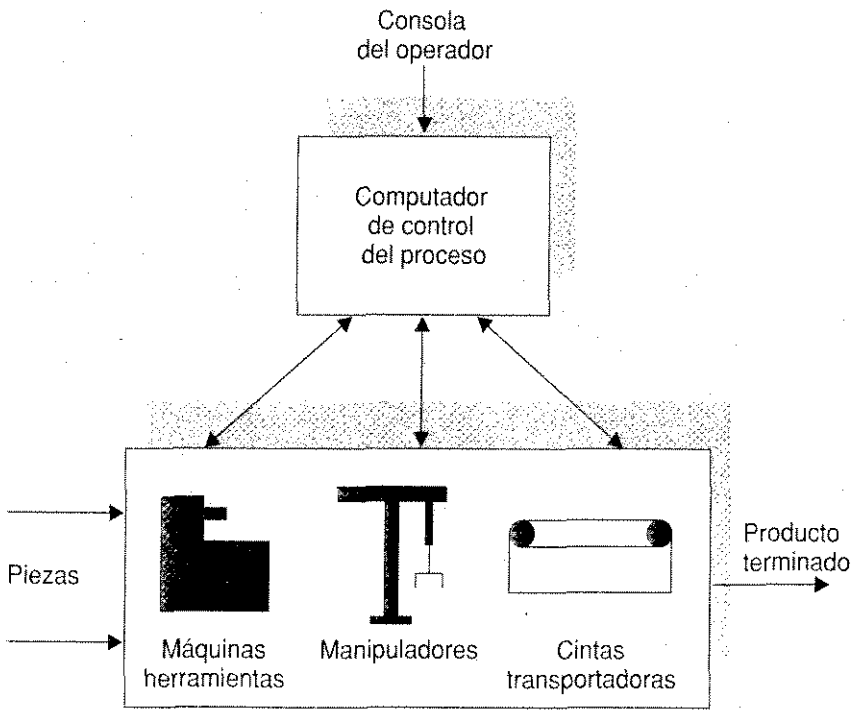


Figura 1.3. Un sistema de control de producción.

instrumentos requeridos para implementar decisiones están distribuidos sobre un área geográfica amplia. La Figura 1.4 representa un diagrama de dicho sistema.

1.2.4 Sistema de computador embebido generalizado

En cada uno de los ejemplos mostrados, el computador interacciona directamente con el equipamiento físico en el mundo real. Con el fin de controlar esos dispositivos del mundo real, el computador necesitará muestrear los dispositivos de medida a intervalos regulares; por lo tanto, se precisa un reloj de tiempo real. Normalmente, existe también una consola de operador para permitir la intervención manual. El operador humano se mantiene constantemente informado del estado del sistema mediante *displays* de varios tipos, incluyendo gráficas.

Los registros de los cambios de estado del sistema se guardan también en una base de datos que puede ser consultada por los operadores, ya sea para situaciones *post mortem* (en el caso de un caída del sistema) o para proporcionar información con propósitos administrativos. Por supuesto, esta información se está utilizando cada vez más para apoyo en la toma de decisiones en los sistemas que están funcionando habitualmente. Por ejemplo, en las industrias químicas y de procesado, la monitorización de la planta es esencial para maximizar las ventajes económicas, más que para maximizar simplemente la producción. Las decisiones relativas a la producción en una planta pueden tener repercusiones serias para otras plantas ubicadas en sitios remotos, particularmente cuando los productos de un proceso se están utilizados como materia prima para otro.

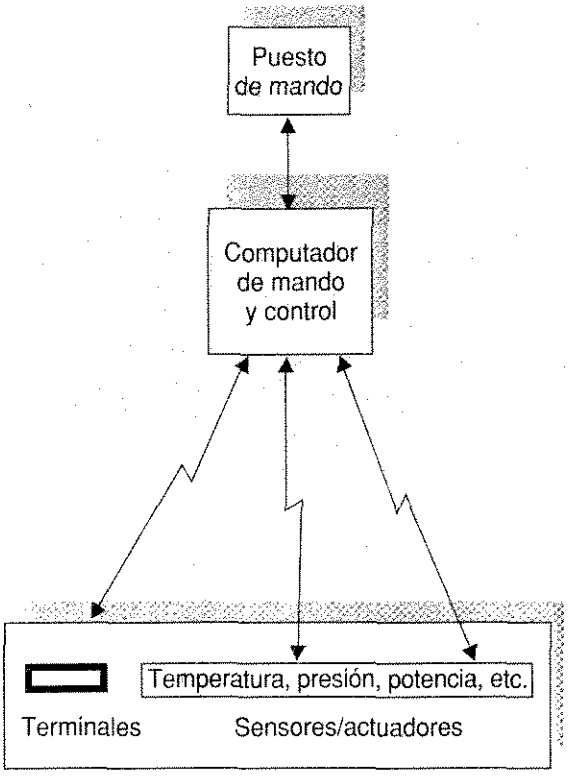


Figura 1.4. Un sistema de mando y control.

Un sistema embebido típico puede estar representado, por tanto, en la Figura 1.5. El software que controla las operaciones del sistema puede estar escrito en módulos que reflejan la naturaleza física del entorno. Normalmente, habrá un módulo que contenga los algoritmos necesarios para controlar físicamente los dispositivos, un módulo responsable del registro de los cambios de estado del sistema, un módulo para recuperar y presentar dichos cambios, y un módulo para interaccionar con el operador.

1.3 Características de los sistemas de tiempo real

Un sistema de tiempo real posee muchas características (bien inherentes bien impuestas) que se señalan en las siguientes secciones. Evidentemente, no todos los sistemas de tiempo real presentan todas estas características; sin embargo, cualquier lenguaje de propósito general (y cualquier sistema operativo) que vaya a ser utilizado para la programación efectiva de sistemas de tiempo real debe tener funcionalidades que soporten estas características.

1.3.1 Grande y complejo

A menudo se dice que la mayoría de los problemas asociados con el desarrollo de software están relacionados con el tamaño y la complejidad. Escribir pequeños programas no presenta ningún problema significativo, ya que pueden ser diseñados, codificados, mantenidos y comprendidos por una única persona. Si esta persona deja la compañía o la institución que está utilizando el software, alguna otra puede comprender el programa en un periodo de tiempo relativamente cor-

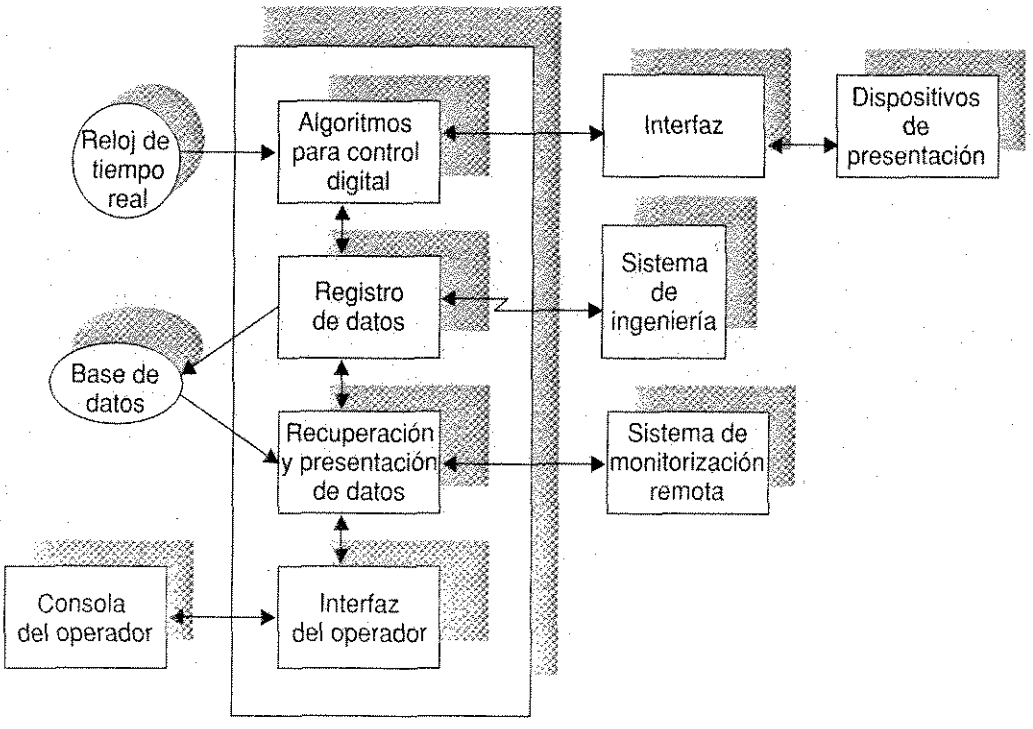


Figura 1.5. Un sistema embebido típico.

to. Por tanto, para estos programas hay un *arte* u *oficio* para su construcción, y podemos decir que *lo pequeño es bello*.

Lamentablemente, no todo el software exhibe esta característica tan deseable de pequeñez. Lehman y Belady (1985), intentando caracterizar los sistemas grandes, rechazan la noción simple, y quizá intuitiva, de que la grandeza es simplemente proporcional al número de instrucciones, líneas de código o módulos que forman un programa. En su caso, ellos relacionan grandeza con variedad, y la mayor o menor grandeza con el grado de variedad. Los indicadores tradicionales, como el número de instrucciones y el esfuerzo de desarrollo, son, por tanto, síntomas de la variedad.

> La variedad es la de las necesidades y actividades en el mundo real y su reflejo en un programa. Pero el mundo real está cambiando continuamente. Está evolucionando. También lo hacen, por consiguiente, las necesidades y actividades de la sociedad. Por tanto, los programas grandes, como todos los sistemas complejos, deben evolucionar continuamente.

Los sistemas embebidos deben responder, por definición, a eventos del mundo real. La variedad asociada con estos eventos debe ser atendida; los programas tenderán, por tanto, a exhibir la propiedad indeseable de grandeza. Inherente a la definición anterior de grandeza es la noción de *cambio continuo*. El coste de rediseñar y reescribir software para responder al cambio continuo en los requisitos del mundo real es prohibitivo. Por tanto, los sistemas de tiempo real precisan mantenimiento constante y mejoras durante sus ciclo de vida. Deben ser extensibles.

Aunque los sistemas de tiempo real son a menudo complejos, las características proporcionadas por los lenguajes y entornos de tiempo real permiten que esos sistemas complejos sean divididos en componentes más pequeños que se pueden gestionar de forma efectiva. Los Capítulos 2 y 4 considerarán estas características en detalle.

1.3.2 Manipulación de números reales

Como se ha indicado anteriormente, muchos sistemas de tiempo real implican el control de alguna actividad de ingeniería. La Figura 1.6 presenta el ejemplo de un sistema de control sencillo. La entidad controlada, la planta, tiene un vector de variables de salida, y, que cambian en el tiempo –aquí $y(t)$–. Estas salidas son comparadas con la señal deseada –o referencia $r(t)$– para producir una señal de error, $e(t)$. El controlador utiliza este vector de error para cambiar las variables de entrada a la planta, $u(t)$. Para un sistema muy sencillo, el controlador puede ser un dispositivo analógico trabajando con una señal continua.

La Figura 1.6 representa un controlador de realimentación. Ésta es la forma más habitual, aunque también se utilizan controladores alimentados por delante (feed-forward). Con el fin de calcular qué cambios deben realizarse en la variables de entrada para que tenga lugar un efecto deseable en el vector de salida, es necesario tener un modelo matemático de la planta. El objetivo de las distintas disciplinas de la teoría de control es la derivación de estos modelos. Muchas veces, una planta se modela como un conjunto de ecuaciones diferenciales de primer orden. Éstas enlazan la salida del sistema con el estado interno de la planta y sus variables de entrada. Cambiar la salida del sistema implica resolver estas ecuaciones para conseguir los valores de entrada

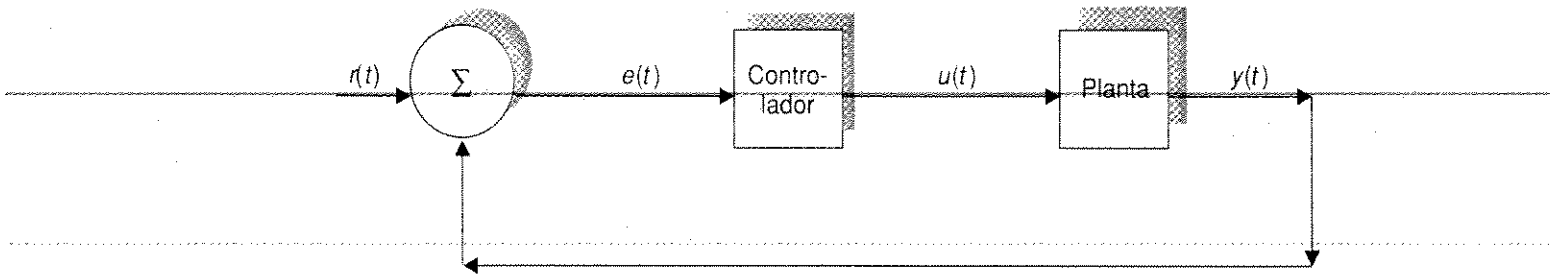


Figura 1.6. Un controlador sencillo.

requeridos. La mayoría de los sistemas físicos presentan inercia, por lo que el cambio no es instantáneo. Un requisito de tiempo real para adaptarse ante una nueva consigna en un periodo de tiempo fijo se añadirá a la complejidad de las manipulaciones necesarias, tanto en el modelo matemático como en el modelo físico. El hecho de que, en realidad, las ecuaciones lineales de primer orden sean sólo una aproximación a las características actuales del sistema también presenta complicaciones.

Debido a estas dificultades –la complejidad del modelo y el número de entradas y salidas distintas (pero no independientes)–, la mayoría de los controladores se implementan mediante computadores. La introducción de un componente digital en el sistema cambia la naturaleza del ciclo de control. La Figura 1.7 es una adaptación del modelo anterior. Los elementos marcados con el signo * son ahora valores discretos; la operación muestrear y mantener se realiza por un convertidor analógico-digital que está bajo control directo de un computador.

Con un computador se pueden resolver las ecuaciones diferenciales mediante técnicas numéricas, aunque los propios algoritmos necesitan ser adaptados para tener en cuenta el hecho de que las salidas de la planta se estén muestreando. El diseño de los algoritmos de control es un tema fuera del alcance de este libro; la implementación de estos algoritmos es, sin embargo, una implicación directa. Ellos pueden ser matemáticamente complejos y necesitar un grado de precisión elevado. Un requisito fundamental de un lenguaje de programación de tiempo real es, por tanto, la capacidad para manipular números reales o de coma flotante. Esto se trata en el Capítulo 3, junto con otros tipos de datos.

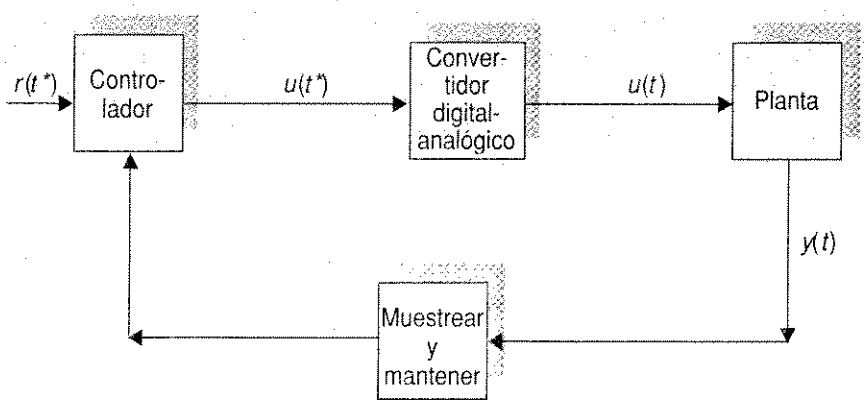


Figura 1.7. Un controlador sencillo computerizado.

1.3.3 Extremadamente fiable y seguro

Cuanto más entrega la sociedad el control de sus funciones vitales a los computadores, más necesario es que dichos computadores no fallen. El fallo de un sistema implicado en la transferencia automática de fondos entre bancos puede conducir a que millones de dólares se pierdan irremediablemente; un componente incorrecto en la generación de electricidad podría redundar en el fallo de un sistema de soporte vital, como una unidad de cuidados intensivos, y la rotura prematura de una planta química podría producir un costoso daño en la maquinaria o un daño ambiental. Estos ejemplos, a veces dramáticos, ilustran que el hardware y software de un computador deben ser fiables y seguros. Incluso en entornos hostiles, como en los que se encuentran las aplicaciones militares, debe ser posible diseñar e implementar sistemas que sólo fallen de una forma controlada. Además, donde se precise interacción con un operador, deberá tenerse especial cuidado en el diseño de la interfaz, con el fin de minimizar la posibilidad de error humano.

El tamaño y la complejidad de los sistemas de tiempo real exacerban el problema de la fiabilidad no sólo deben tenerse en consideración las dificultades esperadas inherentes a la aplicación, sino también aquéllas introducidas por un diseño de software defectuoso.

En los Capítulos 5 y 6 se considerarán los problemas de producir software fiable y seguro junto con las funcionalidades que han introducido los lenguajes para enfrentarse a condiciones de errores, tanto esperadas como inesperadas.

1.3.4 Control concurrente de los distintos componentes separados del sistema

Un sistema embebido suele constar de computadores y varios elementos coexistentes externos con los que los programas deben interactuar simultáneamente. La naturaleza de estos elementos externos del mundo real es existir en paralelo. En nuestro ejemplo típico de computador embebido, el programa debe interaccionar con un sistema de ingeniería (que constará de muchos elementos paralelos, como robots, cintas transportadoras, sensores, actuadores, etc.), y con dispositivos de pantalla del computador (la consola del operador, la base de datos y el reloj de tiempo real). Afortunadamente, la velocidad de un computador moderno es tal, que normalmente estas acciones se pueden realizar en secuencia, dando la ilusión de ser simultáneas. En algunos sistemas embebidos, sin embargo, puede no ser así, como ocurre, por ejemplo, donde los datos son recogidos y procesados en varios lugares distribuidos geográficamente, o donde el tiempo de respuesta de los componentes individuales no se puede satisfacer por un único computador. En estos casos, es necesario considerar sistemas embebidos distribuidos o multiprocesadores.

Uno de los problemas principales asociados con la producción de software de sistemas que presentan concurrencia, es cómo expresar la concurrencia en la estructura del programa. Una posibilidad es dejar todo al programador, que debe construir su sistema de modo que suponga la ejecución cíclica de una secuencia de programa que maneje las distintas tareas concurrentes. Sin embargo, hay varias razones por las que se desaconseja esto:

- Complica la ya difícil tarea del programador, y le implica a él o ella en consideraciones sobre estructuras que son irrelevantes para el control de las tareas a mano.
- Los programas resultantes serán más oscuros y poco elegantes.
- Hace más difícil probar la corrección de programas, y hace más compleja la descomposición del problema.
- Será mucho más difícil conseguir la ejecución del programa en más de un procesador.
- Es más problemática la ubicación del código que trata con los fallos.

Los lenguajes de programación de tiempo real antiguos, por ejemplo RTL/2 y Coral 66, confiaban en el soporte del sistema operativo para la concurrencia; C está asociado normalmente con Unix o POSIX. Sin embargo, los lenguajes más modernos, como Ada, Java, Pearl y occam, tienen soporte directo para programación concurrente. En los Capítulos 7, 8 y 9 se consideran en detalle varios modelos de programación concurrente. La atención en los capítulos siguientes se enfoca en conseguir comunicación y sincronización fiable entre procesos concurrentes en presencia de errores de diseño. En el Capítulo 14, se discuten temas de ejecución en un entorno distribuido, junto con los problemas de tolerancia a fallos de procesador y comunicaciones.

1.3.5 Funcionalidades de tiempo real

El tiempo de respuesta es crucial en cualquier sistema embebido. Lamentablemente, es muy difícil diseñar e implementar sistemas que garanticen que la salida apropiada sea generada en los tiempos adecuados bajo todas las condiciones posibles. Hacer esto y hacer uso de todos los recursos de cómputo todas las veces es, a menudo, imposible. Por esta razón, los sistemas de tiempo real se construyen habitualmente utilizando procesadores con considerable capacidad adicional, garantizando de este modo que el «comportamiento en el peor caso» no produzca ningún retraso inorportuno durante los periodos críticos de operación del sistema.

Dada una adecuada potencia de proceso, se precisa soporte de lenguaje y de ejecución para permitir al programador:

- Especificar los tiempos en los que deben ser realizadas las acciones.
- Especificar los tiempos en los que las acciones deben ser completadas.
- Responder a situaciones en las que no todos los requisitos temporales se pueden satisfacer.
- Responder a situaciones en las que los requisitos temporales cambian dinámicamente (modos de cambio).

Estos requisitos se llaman funcionalidades de control en tiempo real. Permiten que el programa se sincronice con el tiempo. Por ejemplo, con algoritmos de control digital directo es necesario muestrear las lecturas de los sensores en determinados periodos del día (por ejemplo a las dos de la tarde, a las tres de la tarde, y así sucesivamente), o a intervalos regulares (por ejemplo, cada 5 segundos); con los convertidores analógico-digitales, las tasas de muestreo pueden variar desde

unos pocos hertzios a varios cientos de megahertzios. Como resultado de estas lecturas, se precisará realizar otras acciones. Por ejemplo, en una central eléctrica es necesario incrementar el suministro de electricidad a los usuarios domésticos a las 5 de la tarde de lunes a viernes cada semana. Esto es como consecuencia de que se produce un pico en la demanda cuando las familias vuelven a su casa desde el trabajo, encienden las luces, cocinan la cena, etc. En años recientes, en Gran Bretaña la demanda de electricidad doméstica alcanzó un pico al final de la final del Campeonato de Fútbol de 1990, cuando millones de televidentes dejaron sus salones, encendieron las luces de la cocina, y pusieron agua a hervir con el fin de hacer té o café.

Un ejemplo de cambio se puede encontrar en los sistemas de control de vuelo de los aviones. Si un avión ha experimentado despresurización, existe una necesidad inmediata de que todos los recursos de computación se dediquen a hacerse cargo de la emergencia.

Con el fin de cumplir los tiempos de respuesta, es necesario que el comportamiento del sistema sea predecible. Esto se trata en los Capítulos 12 y 13, junto con las funcionalidades del lenguaje que ayuda en la programación de las operaciones críticas en el tiempo.

1.3.6 Interacción con interfaces hardware

La naturaleza de los sistemas embebidos requiere componentes de computador para interaccionar con el mundo externo. Eso se necesita para monitorizar sensores y controlar actuadores para un amplia variedad de dispositivos de tiempo real. Estos dispositivos interactuan con el computador mediante los registros de entrada y salida, y sus requisitos operativos son dependientes de dispositivo y computador. Los dispositivos pueden generar también interrupciones para indicar al procesador que se han realizado ciertas operaciones o que se han alcanzado ciertas condiciones de error.

En el pasado, la interacción con los dispositivos o bien se realizaba bajo control del sistema operativo, o bien requería que el programador de aplicaciones recurriera a inserciones en el lenguaje ensamblador para controlar y manipular los registros e interrupciones. Actualmente, debido a la variedad de dispositivos y a la naturaleza de tiempo crítico de sus interacciones asociadas, su control debe ser directo, y no a través de una capa de funciones del sistema operativo. Además, los requisitos de calidad son argumentos contra el uso de técnicas de programación de bajo nivel.

En el Capítulo 15 se tratarán las funcionalidades de los lenguajes de programación de tiempo real que permiten la especificación de registros de dispositivos y control de interrupciones.

1.3.7 Implementación eficiente y entorno de ejecución

Puesto que los sistemas de tiempo real son críticos respecto al tiempo, la eficiencia de la implementación será más importante que en otros sistemas. Es interesante que uno de los principales beneficios de utilizar un lenguaje de alto nivel es que permite al programador abstraerse de los detalles de implementación y concentrarse en la resolución del problema. Lamentablemente, el

programador de sistemas embebidos no puede permitirse este lujo. Debe preocuparse del coste de utilizar las características de un lenguaje particular. Por ejemplo, si se precisa que la respuesta a alguna entrada sea un microsegundo, no tiene sentido utilizar una característica cuya ejecución precisa un milisegundo.

En el Capítulo 16 se examinará el papel del entorno de ejecución en la obtención de implementaciones eficientes y predecibles.

Resumen

En este capítulo se ha definido un sistema de tiempo real como

> ... cualquier actividad o sistema de proceso de información que tiene que responder a un estímulo de entrada generado externamente con un retardo finito y especificado.

Se han identificado dos clases de estos sistemas: sistemas de tiempo real estrictos, donde es absolutamente imperativo que las respuestas ocurran dentro del tiempo límite especificado, y sistemas de tiempo real no estrictos, donde los tiempos de respuesta son importantes, pero el sistema continuará funcionando correctamente si ocasionalmente no se cumplen los plazos de tiempo.

Se han considerado las características básicas de un sistema de tiempo real o embebido. Éstas son:

- Grandeza y complejidad.
- Manipulación de números reales.
- Fiabilidad y seguridad extrema.
- Control concurrente de componentes separados del sistema.
- Control en tiempo real.
- Interacción con interfaces hardware.
- Implementación eficiente.

Lecturas complementarias

Bailey, D. L., y Buhr, R. J. A. (1998), *Introduction to Real-Time Systems: From Design to Networking with C/C++*, Upper Saddle River, NJ: Prentice Hall.

Buttazzo, G. C. (1997), *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, New York: Kluwer Academic.

Capítulo 2

Diseño de sistemas de tiempo real

Obviamente, la etapa más importante del desarrollo de cualquier sistema de tiempo real es la generación de un diseño consistente que satisfaga una especificación acreditada de los requisitos. En esto, los sistemas de tiempo real no difieren de las demás aplicaciones, aunque su escala en conjunto conlleva problemas de diseño fundamentales. La disciplina de la ingeniería del software se encuentra ampliamente aceptada como el núcleo del desarrollo de métodos, herramientas y técnicas con el fin de asegurar la correcta gestión del proceso de producción de software y la consecución de programas correctos y fiables. Se presupone que los lectores están familiarizados con los temas básicos de la ingeniería del software, de modo que sólo consideramos los problemas y requisitos relacionados con los sistemas de tiempo real embebidos. Incluso con esta restricción, no es posible dar completa cuenta de las muchas metodologías de diseño propuestas. Los temas de diseño *per se* no son nuestro principal foco de atención en este libro. Más bien, el tema central de este libro es investigar las primitivas de los lenguajes y los sistemas operativos, que nos permiten construir los sistemas diseñados. En esta tesitura, trataremos con detalle los lenguajes Ada, Java (para tiempo real), occam2 y C (con POSIX para tiempo real). Los lectores deberán consultar la lista de lecturas complementarias del final de este capítulo para encontrar material adicional sobre el proceso de diseño.

Aunque la mayoría de las aproximaciones al diseño son descendentes, se fundamentan en la comprensión de qué es realizable en los niveles inferiores. En esencia, todos los métodos de diseño incluyen una secuencia de transformaciones desde el estado de los requisitos iniciales hasta el código ejecutable. Este capítulo proporciona una visión general de algunas de las fases habituales por las que transcurre esta ruta, es decir:

- Especificación de requisitos.
- Diseño arquitectónico.
- Diseño detallado.
- Implementación.

- Prueba.

También se discutirán otras actividades importantes:

- Prototipado previo a la implementación final.
- Diseño de la interfaz hombre-máquina.
- Criterios para la evaluación de los lenguajes de implementación.

Puesto que son actividades aisladas, se necesita cierta notación que permita documentar cada etapa. Así, las transformaciones de una etapa hacia otra no son más que traducciones de una notación a otra. Por ejemplo, un compilador producirá código ejecutable partiendo de cierto código fuente expresado en un lenguaje de programación. Sin embargo, no encontramos este mismo nivel de definición para otras traducciones (menor cuanto más subimos en esta jerarquía); normalmente, porque las notaciones empleadas son demasiado vagas e imprecisas, y no pueden captar completamente la semántica, tanto de los requisitos como del diseño.

2.1 Niveles de notación

Hay varios modos de clasificar las formas de notación (o representación). McDermid (1989) proporciona una útil descomposición para nuestros fines y destaca tres técnicas:

(1) Informal

(2) Estructurada

(3) Formal

Los métodos informales suelen hacer uso del lenguaje natural y de diversos tipos de diagramas imprecisos. Cuentan con la ventaja de que estas notaciones son legibles para un gran número de personas (todos aquéllos que conozcan dicho lenguaje natural). Es bien sabido, sin embargo, que las frases en inglés, por poner un caso, pueden interpretarse de formas diversas.

Los métodos estructurados suelen emplear notación gráfica, pero a diferencia de los diagramas informales estos gráficos están bien definidos. Dichos esquemas se construyen a partir de un pequeño número de componentes predefinidas que pueden conectarse de una forma controlada. La forma gráfica puede tener también una representación sintáctica en algún lenguaje bien definido.

A pesar de que pueden idearse métodos estructurados muy rigurosos, no pueden, en sí, ser analizados o manipulados. Para poder efectuar este tipo de operaciones, la notación debe tener una base matemática. Los métodos que presentan dichas propiedades matemáticas se conocen como métodos formales. Éstos tienen la ventaja evidente de que mediante estas notaciones es posible realizar descripciones precisas. Además, es posible probar la existencia de ciertas propiedades necesarias; por ejemplo, que el diseño de alto nivel satisface la especificación de requisitos. Sin

embargo, las técnicas formales son difíciles de comprender para aquellas personas que no han recibido la formación necesaria o que son incapaces de adquirir una suficiente familiaridad con la notación.

Los requisitos de alta fiabilidad propios de los sistemas de tiempo real han provocado un desplazamiento desde las aproximaciones informales hacia las notaciones estructuradas y, cada vez más, hacia las formales. Las empresas especializadas en tiempo real comienzan a emplear, cada vez más, técnicas rigurosas de verificación. Aun así, pocos ingenieros de software poseen las habilidades matemáticas necesarias para explotar completamente el potencial de la verificación. Además, las técnicas formales no suelen disponer de herramientas de estructuración adecuadas para hacer frente a grandes especificaciones. Como consecuencia de ello, la tendencia actual es emplear una combinación adecuada de métodos formales y estructurados.

2.2 Especificación de requisitos

Casi cualquier proyecto de aplicación arranca de una descripción informal de lo que se desea. A ésta debería seguirle un análisis profundo de los requisitos. Es en esta etapa donde se define la funcionalidad del sistema. En cuanto a los factores concretos de tiempo real, deberá hacerse suficientemente explícito el comportamiento temporal del sistema, así como los requisitos de fiabilidad y el comportamiento deseado del sistema en caso de fallos en los componentes. La fase de requisitos deberá definir también los test de aceptación que se aplicarán al software.

A parte del sistema en sí, es necesario construir un modelo del entorno de la aplicación. Es característico de los sistemas de tiempo real presentar interacciones importantes con su entorno. Por tanto, cuestiones como la tasa máxima de interrupciones, el número máximo de objetos externos dinámicos (por ejemplo, las aeronaves en un sistema de control de tráfico aéreo) y los modos de fallos, son importantes.

En el análisis de requisitos se emplean técnicas y notaciones estructuradas, como PSL (Teichrowy Hershey, 1977) y CORE (Mullery, 1979), por sus claras ventajas a la hora de obtener un conjunto de requisitos no ambiguos. Además, los métodos orientados al objeto se están haciendo muy populares (Monarchi y Puhr, 1992), y existe un estándar industrial, UML (véase la Sección 2.4.4). Se han realizado esfuerzos en la línea de los métodos formales para el análisis de requisitos, entre los que destacamos el proyecto FOREST (Goldsack y Finkelstein, 1991), que define un esquema lógico para tratar con los requisitos y un método para la elicitación de requisitos. Más recientemente, en el proyecto Esprit II ICARUS se ha desarrollado el lenguaje ALBERT (Lenguaje Orientado a Agentes para la Construcción y Elicitación de Requisitos para Sistemas de Tiempo Real; *Agent-oriented Language for Building and Eliciting Requirement for real-Time systems*) (Dubois et al., 1995; Bois, 1995). Sin querer hacer de menos estos avances, hay que decir que ninguna notación estructurada o formal capturará los requisitos que el «cliente» haya olvidado mencionar.

La fase de análisis proporciona una especificación acreditada de requisitos. De ella emergerá el diseño. No hay una fase más crítica en todo el ciclo de vida del software, y los documentos en

lenguaje natural son aún la notación habitual de estas especificaciones. Por dar un ejemplo, aunque la sintaxis de los lenguajes para computadores pueden expresarse formalmente (utilizando algún tipo de BNF), la semántica suele venir dada en lenguaje llano, por ejemplo en inglés. La versión original de Ada «fue definida» así, y fue necesario crear una comisión para dilucidar el contenido del documento que define el lenguaje (y que es un estándar internacional). Los fabricantes de compiladores han creído necesario pedir miles de aclaraciones a esta comisión. El estándar actual aún se encuentra definido empleando el lenguaje natural. Por comparación, la semántica de occam se definió utilizando una semántica denotacional (Roscoe, 1985). Esto abre la puerta a la posibilidad de verificar formalmente los compiladores y de disponer de herramientas de programación rigurosas (por ejemplo, para la transformación).

Quizás el método formal más popular, y que comienza a usarse bastante ampliamente, es VDM (Jones, 1986). Otra técnica que ha conseguido gran apoyo es Z (Spivey, 1989). Ambos métodos emplean la teoría de conjuntos y la lógica de predicados, y representan una mejora considerable sobre las técnicas informales y las meramente estructuradas. En su forma actual, sin embargo, no permiten tratar completamente con la especificación de los sistemas de tiempo real.

2.3 Actividades de diseño

El diseño de un sistema embebido grande no se puede hacer de una vez. Debe estructurarse de alguna manera. Para gestionar el desarrollo de los sistemas de tiempo real complejos, se suelen emplear dos aproximaciones complementarias. Éstas forman conjuntamente la base de la mayoría de los métodos de ingeniería de software. La descomposición, como sugiere su nombre, implica una participación sistemática del sistema complejo en partes más pequeñas, hasta poder aislar componentes que puedan ser comprendidos y diseñados por individuos o grupos pequeños. En cada nivel de descomposición, deberá haber un nivel de descripción apropiado y un método para documentar (expresar) esta descripción. La abstracción permitirá posponer cualquier consideración referente a los detalles, particularmente las concernientes a la implementación. Esto permite tener una visión simplificada del sistema y de los objetos contenidos en el que, aun así, contendrá las propiedades y características esenciales. La utilización de la abstracción y la descomposición impregna todo el proceso de ingeniería, y ha influido en el diseño de los lenguajes de programación de tiempo real y en los métodos de diseño de software asociados.

Si se emplea una notación formal para la especificación de requisitos, los diseños de alto nivel deberían utilizar la misma notación, para así poder demostrar si cumplen dicha especificación. Sin embargo, muchas notaciones estructuradas pretenden servir para completar el diseño de alto nivel, o incluso para remplazar la notación formal. De cualquier forma, un diseño estructurado de alto nivel debería ser una especificación acreditada de los requisitos.

2.3.1 Encapsulamiento

El desarrollo jerárquico del software conduce a la especificación y subsiguiente desarrollo de los subcomponentes del programa. Por la naturaleza de la abstracción, estos subcomponentes debe-

rán tener roles bien definidos, e interfaces y conexiones claras e inequívocas. Cuando la especificación completa del sistema software puede verificarse teniendo en cuenta únicamente la especificación de los subcomponentes inmediatos, se dice que la descomposición es **composicional**. Ésta es una propiedad importante cuando se trata de analizar formalmente programas.

Los programas secuenciales son particularmente idóneos para los métodos composicionales, y existen técnicas para encapsular y representar subcomponentes. Simula introdujo la significativa construcción *class* (clase). Modula-2 emplea la estructura de módulo, que, a pesar de ser menos potente, sigue siendo importante. Más recientemente, han aparecido lenguajes *orientados al objeto*, como C++, Java y Eiffel, partiendo de la construcción clase. Ada emplea una combinación de módulos y extensiones de tipo para soportar la programación orientada al objeto. Más adelante, en el Capítulo 4, se discutirán estas herramientas.

Los objetos, si bien proporcionan una interfaz abstracta, requieren elementos adicionales si han de ser usados en un entorno concurrente. Habitualmente, esto conlleva la inclusión de alguna noción de proceso. La abstracción de *proceso* es sobre la que tratará este libro en lo sucesivo. En el Capítulo 7, se presenta la noción de proceso, y el Capítulo 8 considerará la interacción de procesos mediante variables compartidas. La comunicación de procesos basada en mensajes, sin embargo, proporciona una interfaz más controlada y abstracta, y será tratada en el Capítulo 9.

Tanto la abstracción de objeto como la abstracción de proceso son importantes en el diseño e implementación de los sistemas embebidos fiables.

2.3.2 Cohesión y acoplamiento

Las dos formas anteriores de ensapsulamiento conducen al empleo de módulos con interfaces bien definidas (y abstractas). Pero, ¿cómo debería descomponerse en módulos un sistema grande? En gran medida, la respuesta a esta pregunta está en el trasfondo de todas las actividades de diseño de software. Sin embargo, antes de la discusión de algunos de estos métodos, debemos considerar algunos principios generales más que conducen a un buen encapsulamiento. La cohesión y el acoplamiento son dos medidas que describen la vinculación entre módulos.

La cohesión expresa el grado de unión de un módulo: su fuerza interna. Allworth y Zobel (1987) indican seis medidas de cohesión que van desde la más débil a la más fuerte:

- **Casual**: los elementos del módulo mantienen tan solo vínculos muy superficiales; por ejemplo, el haber sido escritos en el mismo mes.
- **Lógica**: los elementos del módulo están relacionados desde el punto de vista del sistema completo, pero no en términos reales de software; por ejemplo, todos los gestores de dispositivos de salida.
- **Temporal**: los elementos del módulo se ejecutan en momentos similares; por ejemplo, las rutinas de arranque.
- **Procedural**: los elementos del módulo se emplean en la misma sección del programa; por ejemplo, los componentes de la interfaz de usuario.

- **De comunicación** (*sic*): los elementos del módulo operan sobre la misma estructura de datos.
- **Funcional**: los elementos del módulo operan conjuntamente para contribuir a la ejecución de una única función del sistema; por ejemplo, proporcionar un sistema de archivos distribuidos.

El acoplamiento, por otra parte, es una medida de la interdependencia entre dos módulos de un programa. Si dos módulos intercambian información de control entre ellos, se dice que poseen un acoplamiento alto (rígido, o fuerte). Por el contrario, el acoplamiento es débil si únicamente se intercambian datos. Otra forma de contemplar el acoplamiento es considerar la facilidad con que puede eliminarse un módulo (de un sistema ya completo) y remplazarlo con un módulo alternativo.

En cualquier método de diseño, una buena descomposición es aquélla que posee fuerte cohesión y débil acoplamiento. Este principio es igualmente válido tanto para el dominio de la programación secuencial como para el de la programación concurrente.

2.3.3 Aproximaciones formales

El empleo de redes sitio-transición (Place-Transition) (Brauer, 1980) para el modelado de sistemas concurrentes fue propuesto por C. A. Petri hace más de treinta años. Estas redes se construyen como un grafo bipartito dirigido con dos tipos de nodos: elementos *S*, que denotan estados atómicos locales, y elementos *T*, que denotan transiciones. Los arcos del grafo proporcionan la relación entre los elementos *S* y *T*. El marcado del grafo se indica mediante marcas sobre los elementos S; el desplazamiento de un testigo representa un cambio en el estado del programa. Mediante reglas se especifica cuándo y cómo se puede mover un testigo de un elemento S a otro a través de un elemento de transición. Las redes de Petri se definen matemáticamente, y son susceptibles de un análisis formal.

Las redes sitio-transición poseen las útiles cualidades de ser simples, abstractas y gráficas, y proporcionan un marco de trabajo general para analizar muchos tipos de sistemas distribuidos y concurrentes. En contrapartida, pueden dar lugar a representaciones muy grandes y complicadas. Para poder trabajar con un modelo más conciso de tales sistemas se han confeccionado las redes predicado-transición. Con estas redes, un elemento *S* puede modelar varios elementos *S* normales (de igual modo elementos *T*), y las marcas, que en origen no tenían estructura interna, pueden ser «coloreadas» por tuplas de datos.

Una alternativa a las redes de Petri es el uso de autómatas temporizados y comprobación de modelos (Model Checking). De nuevo, el sistema concurrente se representa mediante un conjunto de máquinas de estado finito, y se emplean técnicas de exploración de estados para investigar los posibles comportamientos del sistema.

Las redes de Petri y los autómatas temporizados representan una forma de modelado de los sistemas concurrentes. Otras aproximaciones rigurosas requieren una descripción formal del lenguaje de implementación propuesto. Tras obtener estos axiomas, es posible desarrollar reglas de

prueba para analizar el comportamiento de los sistemas concurrentes. Desgraciadamente, se han desarrollado pocos lenguajes de implementación con una descripción formal en mente (siendo occam una notable excepción). La notación *Communicating Sequential Processes* (CSP; procesos secuenciales que se comunican) fue desarrollada para permitir la especificación y el análisis de sistemas concurrentes. En CSP, cada proceso se describe en términos de eventos externos, esto es, de acuerdo con la comunicación que mantiene con otros procesos. La historia de un proceso se representa mediante una *traza*, que es una secuencia finita de eventos.

Un sistema representado en CSP puede ser analizado para determinar su comportamiento. En particular, podrán examinarse las propiedades de *seguridad* (safety) y *vivacidad* (liveness). Owicki y Lamport (1982) caracterizaron estos dos conceptos como:

- **Seguridad**: «algo malo no va a ocurrir».
- **Vivacidad**: «algo bueno va a ocurrir».

Las técnicas de comprobación de modelos pueden usarse también para verificar las propiedades de seguridad y vivacidad.

Aunque los sistemas de tiempo real son concurrentes, también tienen requisitos de temporización. Su análisis, entonces, requiere un tipo de lógica apropiada. La lógica temporal es una extensión del cálculo de proposiciones y de predicados, con nuevos operadores para expresar propiedades relativas al tiempo real. Los operadores usuales son: **siempre, en-algún-momento** (sometimes), **hasta, desde** y **conduce-a**. Por ejemplo, **en-algún-momento** (<>) indica que la siguiente propiedad será cierta en algún momento del futuro; por ejemplo, <> (`y > N`) implica que eventualmente `y` tomará un valor mayor que `N`.

Muchas de las aplicaciones de la lógica temporal se encuentran más en la verificación de programas existentes que en la especificación jerárquica y en el desarrollo riguroso de otros nuevos. Es posible acusar a la lógica de ser demasiado global y poco modular, siendo necesario habitualmente poseer el programa completo para poder analizar cualquier parte de él. Para soslayar estas dificultades, es posible extender el formalismo de modo que las transiciones mismas se conviertan efectivamente en proposiciones de la lógica. Esta aproximación ha sido promovida por Lamport (1983), y Barringer y Kuiper (1983). Refinamientos posteriores a la lógica temporal permiten agregar tiempos máximos a estos operadores temporales. Así, por ejemplo, no solamente `y` llegará a ser mayor que `N`, sino que esto ocurrirá en un tiempo acotado. El formalismo RTL (lógica de tiempo real; *real-time logic*) es un buen ejemplo de un método que combina el tiempo con la lógica de predicados de primer orden (Jahanian y Mok, 1986). La importante cuestión de representar los requisitos de temporización y la planificación con tiempos límite (deadlines) se discute de nuevo, en detalle, en los Capítulos 12 y 13.

2.4 Métodos de diseño

Ya se comentó que la mayoría de los diseñadores de sistemas de tiempo real promueven un proceso de abstracción de objetos, y que existen técnicas formales que permiten especificar y anali-

zar sistemas concurrentes con restricciones temporales. No cabe duda de que estas técnicas no están aún suficientemente maduras para constituir métodos de diseño «probados y contrastados». Más bien, la industria de tiempo real usa, en el mejor de los casos, métodos estructurados y aproximaciones de ingeniería de software aplicables a la mayoría de los sistemas de procesamiento de información. Estos métodos no proporcionan un soporte específico para el dominio de tiempo real, y carecen de la riqueza necesaria para explotar completamente la potencia de los lenguajes de implementación.

Existen muchos métodos de diseño estructurados orientados a los sistemas de tiempo real: MASCOT, JSD, Yourdon, MOON, OOD, RTSA, HOOD, DARTS, ADARTS, CODARTS, EPOS, MCSE, PAMELA, HRT-HOOD, Octopus, etc. Gomaa (1994) sugiere cuatro objetivos importantes para un método de diseño de tiempo real. Debe ser capaz de:

(1) Estructurar un sistema en tareas concurrentes.

(2) Dar soporte al desarrollo de componentes reusables mediante la ocultación de información.

(3) Definir los aspectos del comportamiento mediante máquinas de estado finito.

(4) Analizar las prestaciones de un diseño para determinar sus propiedades de tiempo real.

Ninguno de los métodos anteriores cumple todos estos objetivos. Por ejemplo, RTSA es débil en la estructuración de tareas y ocultación de información, y JSD no da soporte a la ocultación de información o a las máquinas de estado finito. Además, pocos de estos métodos soportan directamente las abstracciones habituales del tiempo real estricto (tales como actividades periódicas y esporádicas) que se encuentran en la mayoría de los sistemas de tiempo real estrictos. Incluso, sólo una minoría impone un modelo computacional que asegure que pueda realizarse un análisis de la temporización efectiva del sistema final. Consecuentemente, su utilización es propensa a errores, y puede conducir a sistemas cuyas propiedades de tiempo real no puedan ser analizadas.

PAMELA permite la representación de diagramas de actividades cíclicas, máquinas de estado y manejadores de interrupciones. Sin embargo, la notación no está respaldada por abstracciones para los recursos, y en consecuencia los diseños no son necesariamente adecuados para el análisis de temporización. HRT-HOOD da soporte a tres de los cuatro requisitos, careciendo, sin embargo, de la posibilidad de definir los aspectos de comportamiento mediante máquinas de estado finito. También, al ser sólo basado en objetos, sus mecanismos de apoyo a la reutilización de componentes es débil. HOOD V4 intenta resolver las debilidades de HOOD V3, e incorpora los puntos fuertes de HRT-HOOD. Sin embargo, su soporte para el diseño arquitectónico reusable es débil, y el análisis de prestaciones del diseño no está garantizado.

EPOS (Lauber, 1989) es otro método que da soporte al ciclo de vida completo de un sistema de tiempo real. Proporciona tres lenguajes de especificación: uno para describir los requisitos del cliente, otro para describir la especificación del sistema, y otro para describir la gestión del proyecto, la gestión de la configuración y las garantías de calidad. Como HOOD y HRT-HOOD, la especificación del sistema tiene una representación gráfica y textual. Pueden representarse actividades periódicas y esporádicas (aunque no son directamente visibles cuando se manipulan los diagramas), y pueden especificarse los requisitos de temporización. Además, dentro del contex-

to de EPOS (Lauber, 1989) se aborda el reconocimiento temprano del comportamiento de las aplicaciones de tiempo real; sin embargo, este trabajo se basa en una animación de la especificación del sistema, y no es un análisis de temporización.

Para una comparación de estos métodos, véase Hull et al. (1991), Cooling (1991), Calvez (1993) y Gomaa (1994).

Como se comentó en la introducción, la mayoría de los métodos tradicionales de desarrollo de software incorporan un modelo de ciclo de vida donde se ubican las siguientes actividades:

- Especificación de requisitos: durante la cual se produce una especificación acreditada del comportamiento funcional y no funcional del sistema.
- Diseño arquitectónico: durante el cual se desarrolla una descripción de alto nivel del sistema propuesto.
- Diseño detallado: durante el cual se especifica el diseño completo del sistema.
- Codificación: durante la cual se implementa el sistema.
- Prueba: durante la cual se comprueba la eficacia del sistema.

Para los sistemas de tiempo real estrictos, se presenta la desventaja importante de que los problemas de temporización sólo se podrán detectar durante la prueba, o, lo que es incluso peor, tras el despliegue (deployment) de la aplicación.

Como es normal, un método de diseño estructurado empleará un diagrama en el que las flechas etiquetadas muestran el flujo de datos a través del sistema, y ciertos nodos representan lugares donde se transforman dichos datos (esto es, los procesos). En las siguientes secciones se muestran brevemente JSD y Mascot3; ambas técnicas han sido utilizadas extensamente en el dominio de tiempo real. Posteriormente, se presenta un método específico para Ada: HRT-HOOD. Éste es un método especialmente indicado para sistemas de tiempo real estrictos, y se empleará en el Capítulo 17. Finalmente, se comentará UML.

2.4.1 JSD

El método de desarrollo de sistemas de Jackson (Jackson, 1975) emplea una notación precisa para la especificación (diseño de alto nivel) y la implementación (diseño detallado). Curiosamente, la implementación no es sólo una reescritura detallada de la especificación, sino el resultado de la aplicación de ciertas transformaciones sobre la especificación inicial con el objeto de incrementar la eficiencia.

Un grafo JSD consta de procesos y una red de interconexión. Los procesos son de tres clases:

- Procesos de entrada, que detectan acciones en el entorno y las transfieren al sistema.
- Procesos de salida, que transfieren al entorno las respuestas del sistema.
- Procesos internos.

Los procesos pueden enlazarse de dos formas distintas:

- Mediante conexiones de flujos de datos asíncronos con búfer.
- Mediante conexiones de vectores de estado (o inspecciones).

Una conexión de vector de estados permite que un proceso vea el estado interno de otro proceso sin necesidad de comunicarse con el.

El grafo JSD proporciona una arquitectura del sistema. Aún faltan por añadir las estructuras de datos pertinentes para la información que se transfiere realmente en el sistema y el detalle de las acciones incluidas dentro de cada proceso. Desgraciadamente, JSD no proporciona una forma estándar de incluir las restricciones de temporización, por lo que uno mismo tiene que añadir anotaciones a los diagramas.

Como es lógico, lo que proporciona JSD es una plataforma para expresar un diseño; no realiza el diseño por usted. El diseño incorpora, inevitablemente, la experiencia y la creatividad humanas (como en la programación). Se ha dicho a menudo que los métodos estructurados y formales tienen como objetivo restringir la creatividad, y no es así. Lo que ofrecen estas técnicas es una notación bien conocida para expresar el diseño, y técnicas para comprobar que la creatividad ocupa su lugar adecuado; esto es, que el diseño cumple la especificación, y que el software implementa el diseño.

El tema principal del diseño en JSD se encuentra en el flujo de datos. Así, un «buen» diseño es el que incluye este flujo natural. Ya se han discutido los factores que conducen a una buena descomposición; son éstas, precisamente, las cuestiones que deberían influir en un diseño JSD (y en todos los demás métodos de diseño). Los procesos se categorizan, por sí mismos, en unos pocos tipos diferentes, y una aproximación de diseño de alto nivel que tenga estos tipos como objetivos dará como resultado un diseño fácil de implementar.

Otra ventaja del enfoque de flujo de datos es que las restricciones de temporización se expresan a menudo como atributos de los datos que atraviesan el sistema. Una interrupción genera una señal de control; en esencia, la señal de control es una transformación de la interrupción. Estas transformaciones tienen lugar dentro de los procesos, y llevan cierto tiempo. Una adecuada elección de los procesos proporcionara unos límites de tiempo claros que podrán ser planificados (aunque la planificabilidad real no pueda comprobarse directamente).

Una vez logrado el diseño, la implementación debe poder obtenerse de forma sistemática. Esto es mucho más fácil cuando el lenguaje de implementación proporciona un modelo de concurrencia basado en mensajes, dado que tanto los procesos del diseño como los flujos de datos con búfer pueden codificarse como procesos del programa. Desgraciadamente, esto puede ocasionar un exceso de procesos y una implementación ineficiente. Para evitar esto, hay dos caminos posibles:

- Transformar el diseño de modo que requiera menos procesos.
- Una vez obtenido el programa con exceso de procesos, tratar de reducir el número de objetos concurrentes.

La mayoría de los lenguajes no son apropiados para las técnicas de transformación. (Occam2 es, de nuevo, una notable excepción, dado que su semántica se encuentra definida formalmente.) La aproximación aconsejada por el método JSD es una transformación conocida como **inversión**. En ella, se remplaza cada proceso del diseño por un procedimiento con un proceso planificador simple que controla la ejecución de un conjunto de estos procedimientos; así, en lugar de un conducto con cinco procesos, el planificador llamaría a cada uno de los procesos cuando hubiera datos (si los cinco procesos fueran idénticos habría un solo procedimiento, que sería llamado cinco veces). De nuevo, son un problema las restricciones temporales, dado que son difíciles de preservar durante la inversión.

Aunque JSD no fue diseñado originalmente para aplicaciones de tiempo real, ha sido empleado con éxito en algunos sistemas realmente grandes. Con JSD, se han obtenido implementaciones en Ada u occam2, donde gran parte del código ha sido generado automáticamente (Lawton y France, 1988).

2.4.2 Mascot3

Mientras que JSD sólo ha sido utilizado recientemente en el dominio de tiempo real, Mascot se diseñó específicamente para el diseño, la construcción y la ejecución de software de tiempo real. Mascot1 apareció a principios de la década de 1970, pero fue mejorado prontamente con Mascot2, y ya en la década de 1980 con Mascot3.

Mascot3 se caracteriza por emplear gráficos de redes de flujo de datos y un diseño jerárquico. La modularidad es un elemento clave de este método, en el que se pueden identificar módulos para el diseño, la construcción, la implementación y la prueba. Un diseño en Mascot3 puede expresarse tanto en forma textual como gráfica.

Además de los flujos de datos, una descripción en Mascot3 puede contener:

- Subsistemas.
- Áreas de intercomunicación de datos (IDA; intercomunication data area) generales.
- Actividades (procesos).
- Canales (un IDA que actúa como un búfer).
- Depósitos (un IDA que actúa como un repositorio de información).
- Servidores (un elemento de diseño que comunica con dispositivos hardware externos).

Se proporcionan las sincronizaciones necesarias sobre el uso de los canales y los depósitos. Los subsistemas pueden contener conjuntos de otros elementos, incluyendo más subsistemas.

La implementación de un diseño en Mascot3 puede lograrse de dos formas bien diferentes. Puede emplearse un lenguaje de programación concurrente apropiado, o una plataforma de ejecución estándar de tiempo real. Con Mascot2, los algoritmos se codifican en un lenguaje secuencial, como Coral 66 o RTL2 (aunque también se han empleado FORTRAN, Pascal, C y

Algol 60), y después este software se aloja en una plataforma de ejecución Mascot de tiempo real. Al emplearse un lenguaje de programación concurrente conseguiremos que todo el sistema se implemente en un solo lenguaje, con lo que se facilita significativamente la integración.

Los lenguajes como Ada, que dan soporte a la descomposición y poseen un modelo de sincronización basado en mensajes (ya sea predefinido o programable) presentan una solución claramente favorable en cuanto a la tarea de implementación. Actualmente se tiende, cada vez más, a emplear Ada con Mascot3, a pesar de que sus modelos de acción básicos no son completamente compatibles (Jackson, 1986). Debe hacerse notar, sin embargo, que Mascot también presenta el problema de la proliferación de procesos.

2.4.3 HRT-HOOD

HRT-HOOD (Burns y Wellings, 1995) difiere de Mascot y JSD en que aborda directamente las cuestiones de los sistemas de tiempo real estrictos. Se ve el proceso de diseño como una progresión de *compromisos* específicos crecientes (Burns y Lister, 1991). Estos compromisos definen propiedades del diseño del sistema que los diseñadores de niveles más detallados no pueden cambiar. Aquellos aspectos de un diseño sobre los que no se ha contraído ningún compromiso en ningún nivel concreto de la jerarquía son objeto de *obligaciones* que deberán resolver los niveles inferiores del diseño. Ya en las etapas tempranas del diseño habrá compromisos sobre la estructura arquitectónica del sistema, en términos de definiciones y relaciones entre objetos. Sin embargo, el comportamiento detallado de los objetos definidos es objeto de las obligaciones que deberán cumplimentarse en un diseño más detallado y en la implementación.

El proceso de refinamiento del diseño (donde se transforman obligaciones en compromisos) suele estar sujeto a *restricciones* impuestas principalmente por el entorno de ejecución. El entorno de ejecución es el conjunto de componentes hardware y software (por ejemplo, procesadores, distribuidores de tareas, o controladores de dispositivos) sobre los que se construye el sistema. Éste puede imponer tanto restricciones sobre recursos (por ejemplo, velocidad de procesamiento, o ancho de banda de comunicación) como restricciones de mecanismos (por ejemplo, prioridades de interrupción, distribución de tareas, o bloqueo de datos). Estas restricciones vienen fijadas en la medida en que sea inmutable el entorno de ejecución.

Las obligaciones, los compromisos y las restricciones tienen una influencia importante sobre el diseño arquitectónico de cualquier aplicación. Por esto, HRT-HOOD define dos actividades para el diseño arquitectónico:

- La actividad de diseño de la arquitectura lógica.
- La actividad de diseño de la arquitectura física.

La arquitectura lógica incorpora compromisos que pueden conseguirse con independencia de las restricciones impuestas por el entorno de ejecución, y se orienta primordialmente a satisfacer los requisitos funcionales (aunque la existencia de requisitos de temporización, como las cotas de tiempo de extremo a extremo, influirán profundamente sobre la descomposición de la arquitectura lógica). La arquitectura física toma en cuenta estos requisitos funcionales y otras restricciones, e in-

cluye los requisitos no funcionales. La arquitectura física proporciona la base para poder afirmar que se cumplirán los requisitos no funcionales de la aplicación una vez hayan tenido lugar el diseño detallado y la implementación. Aborda los requisitos de temporización y confiabilidad, y el preceptivo análisis de planificabilidad que asegurará (garantizará) que el sistema, una vez construido, funcione correctamente, tanto en el dominio de los valores como en el de tiempo.

Aunque la arquitectura física es un refinamiento de la arquitectura lógica, su desarrollo será, normalmente, un proceso iterativo y concurrente en el que ambos modelos son desarrollados/modificados. Las técnicas de análisis incluidas en la arquitectura física pueden y deben aplicarse lo más tempranamente posible. Los supuestos iniciales sobre los recursos pueden definirse de modo que estén sujetos a modificación y revisión según se refina la arquitectura lógica. De esta forma, se persigue un diseño «realizable» desde los requisitos hasta el despliegue.

En el Capítulo 17 se darán más detalles sobre HRT-HOOD, y se desarrollará un caso de estudio (en este método de diseño) para mostrar cierto número de cuestiones mencionadas en los capítulos siguientes a éste.

2.4.4 El lenguaje unificado de modelado (UML)

Durante los últimos diez años, se ha desarrollado una plétora de métodos de análisis y diseño con orientación al objeto (OO). ¡Graham (1994) da un número superior a 60! Desgraciadamente, la primera toma de contacto de la industria fue obscurecida por la falta de un proceso de estandarización. En el dominio de tiempo real, la situación fue exacerbada por el fracaso de las técnicas OO para dar un soporte adecuado a los sistemas de tiempo real. Y esto a pesar de los nuevos logros teóricos sobre el análisis de los programas de tiempo real, que hoy en día son capaces de dar soporte a la mayoría de los requisitos de tiempo real (véase el Capítulo 13).

Recientemente, se ha hecho un esfuerzo significativo por parte del Object Management Group para desarrollar el lenguaje unificado de modelado (UML; Unified Modeling Language) como estándar para la notación de orientación al objeto. UML es un lenguaje gráfico para visualizar, especificar, construir y documentar los artefactos de un sistema compuesto principalmente de software (Booch et al., 1999). Está claro que, durante los próximos diez años, UML se convertirá en la notación dominante para el análisis y diseño orientados al objeto. Incluso el resto de los métodos orientados al objeto, como Fusion (Coleman et al., 1994), están actualizándose a la nueva notación.

Los principales aspectos de UML que se adecuan al modelado de sistemas embebidos de tiempo real son (Douglass, 1999):

- Un modelo de objetos (que incorpora atributos para datos, estado, comportamiento, identidad y responsabilidad) que permite capturar la estructura del sistema.
- Escenarios de casos de uso, que permiten identificar respuestas clave del sistema o las entradas del usuario.
- Modelado del comportamiento, con máquinas de estado finito (diagramas de estados; *statecharts*) que facilitan el modelado de la dinámica del comportamiento del sistema.

- Empaquetado, que proporciona mecanismos para organizar los elementos del modelado; representaciones para la concurrencia, la comunicación y la sincronización (para modelar entidades del mundo real).
- Modelos de la topología física, donde se emplean diagramas de despliegue para mostrar cómo está compuesto el sistema a partir de dispositivos y procesadores.
- Soporte para patrones y marcos orientados al objeto, que permiten representar soluciones comunes a problemas comunes.

Sin embargo, y como su nombre indica, UML ofrece un lenguaje, no un método. En consecuencia, es preciso aumentar la notación con un proceso de diseño. Douglass (1999) propone ROPES (*Rapid Object-oriented Process for Embedded Systems;* proceso rápido orientado al objeto para sistemas embebidos) con UML. ROPES se basa en un ciclo de vida iterativo orientado a la generación rápida de prototipos. Las actividades son: análisis, diseño, implementación y prueba. En concordancia con otros procesos UML (Jacobson et al., 1999), ROPES está dirigido por los casos de uso. Se identifica un conjunto de casos de uso y se clasifican según su prioridad, riesgo y rasgos en común. Después, se emplean para producir prototipos de los sistemas (mediante herramientas de generación automática de código, cuando sea posible).

El trabajo actual en UML (Selic et al., 2000) se centra en cómo definir paradigmas estándar para el modelado de sistemas de tiempo real usando un perfil UML. Las cuestiones a dilucidar son, entre otras, cómo modelar el tiempo y otros recursos, y cómo predecir el rendimiento de los sistemas (utilizando análisis de planificabilidad).

2.5 Implementación

Entre la especificación de requisitos de alto nivel y el código de máquina en funcionamiento encontramos un gran hueco, ocupado por el lenguaje de programación. El desarrollo de lenguajes para la implementación de sistemas de tiempo real es el tema central de este libro. El diseño de lenguajes es aún un tema de investigación muy activo. Aunque el diseño de los sistemas debería conducir de modo natural hacia la implementación, el poder expresivo de la mayoría de los lenguajes modernos está aún lejos de alcanzarse con las metodologías de diseño actuales. Sólo comprendiendo qué es posible en la etapa de implementación, podremos conseguir una aproximación apropiada al diseño.

Podemos identificar tres clases de lenguajes de programación que o bien se utilizan, o bien han sido utilizados en el desarrollo de sistemas de tiempo real. Estos son: los lenguajes de ensamblado, los lenguajes de implementación de sistemas secuenciales, y los lenguajes concurrentes de alto nivel.

2.5.1 Ensamblador

Inicialmente, la mayoría de los sistemas de tiempo real se programaron en el lenguaje de ensamblado (o ensamblador) del computador embebido. La causa era que la mayoría de los lenguajes

de alto nivel no disponían de soporte suficiente para la mayoría de los microcomputadores, y el lenguaje de programación de ensamblado parecía ser la única forma de obtener una implementación eficiente que pudiera acceder a los recursos del hardware.

El principal inconveniente de los lenguajes de ensamblado es que están orientados a la máquina en lugar de al problema. El programador debe preocuparse de problemas que nada tienen que ver con los algoritmos en sí, de manera que éstos se vuelven irreconocibles. Esto hace que los costes de desarrollo se eleven y que resulte difícil modificar los programas cuando se encuentran errores, o cuando es preciso efectuar mejoras.

Aún más dificultades aparecen cuando es preciso reescribir los programas, dado que no es posible llevarlos de una máquina a otra. Además, se hace necesario volver a formar a los programadores si se necesita trabajar con otras máquinas diferentes.

2.5.2 Lenguajes de implementación de sistemas secuenciales

A medida que los computadores fueron siendo más potentes y los lenguajes de programación más maduros, y progresaba la tecnología de los compiladores, las ventajas de escribir software de tiempo real en un lenguaje de alto nivel sobrepasaron a los inconvenientes. Para tratar con las deficiencias de lenguajes como FORTRAN, se desarrollaron nuevos lenguajes específicos para la programación embebida. En las Fuerzas Aéreas de EE.UU, por ejemplo, se trabajó de modo habitual con Jovial. En el Reino Unido, el Ministerio de Defensa se adhirió al estándar de Coral 66; grandes corporaciones industriales, como ICI, se adhirieron a RTL/2. Más recientemente, han ido ganando popularidad los lenguajes de programación C y C++.

Todos estos lenguajes tienen un punto en común: son secuenciales. También suelen presentar carencias en lo que respecta a sus funciones de control y fiabilidad para tiempo real. En consecuencia, suele ser necesario basarse en el soporte del sistema operativo y fragmentos de código en ensamblador.

2.5.3 Lenguajes de programación concurrente de alto nivel

Pese al creciente empleo de lenguajes orientados a aplicaciones (como los lenguajes de implementación de sistemas secuenciales para ciertas aplicaciones, por ejemplo COBOL para aplicaciones de procesamiento de datos y FORTRAN para aplicaciones científicas e ingenieriles), la producción de software fue haciéndose progresivamente más difícil durante la década de 1970, a medida que los sistemas informáticos se volvieron más grandes y sofisticados.

Es habitual referirse a estos problemas como la crisis del software. Se detectaron diversos síntomas de esta crisis (Booch, 1986):

- Idoneidad: a menudo, los sistemas de producción que han sido automatizados no se ajustan a las necesidades de los usuarios.

- Fiabilidad: el software no es de fiar, y a menudo fallará con respecto a su especificación.
- Coste: es difícil predecir el coste final del software.
- Facilidad de modificación: el mantenimiento del software es complejo, costoso y propenso a errores.
- Puntualidad: el software se suele entregar con demora.
- Portabilidad: el software de un sistema raramente sirve para otro sistema.
- Eficiencia: los esfuerzos de desarrollo de software no consiguen dar un empleo óptimo a los recursos involucrados.

Quizás una de las mejores muestras del impacto de la crisis del software puede encontrarse en la búsqueda por parte del Departamento de Defensa Americano (DoD) de un lenguaje de programación de alto nivel común para todas sus aplicaciones. A medida que comenzaban a caer los precios del hardware durante los años 1970, el DoD observó los crecientes costes de su software embebido. Se estimó que en 1973 se habían gastado tres mil millones de dólares tan sólo en software. Una encuesta que tenía como objeto los lenguajes de programación mostró que en la construcción de aplicaciones embebidas se utilizaban más de 450 lenguajes de propósito general y dialectos incompatibles. En 1976 se evaluaron los lenguajes existentes en base a un conjunto de requisitos de creciente importancia. A partir de estas evaluaciones, se extrajeron cuatro conclusiones principales (Whitaker, 1978):

(1) Ningún lenguaje del momento era apropiado.

(2) El objetivo deseable era obtener un único lenguaje.

(3) El estado actual del arte del diseño de lenguajes no servía para cumplir los requisitos.

(4) Este desarrollo debería arrancar de un lenguaje base apropiado; entre los que se recomendó Pascal, PL1 y Algol68.

El resultado fue el nacimiento, en 1983, de un nuevo lenguaje, llamado Ada. En 1995, se modernizó el lenguaje para reflejar los últimos 10 años de experiencia y avances del diseño moderno de lenguajes de programación.

Aunque el esfuerzo sobre el lenguaje de programación Ada ha dominado la escena de la investigación en lenguajes de programación de computadores embebidos desde principios de los años 1970, han aparecido también otros lenguajes. Por ejemplo Modula-1 (Wirth, 1977b), que fue desarrollado por Wirth para su empleo en dispositivos programables y que pretendió

> ... conquistar esa fortaleza de la programación en ensamblador, o al menos asediarla vigorosamente.

Quizás el lenguaje con más éxito en esta área sea C (Kernighan y Ritchie, 1978). Actualmente, quizás sea éste el lenguaje de programación más popular en el mundo.

Gran parte de la experiencia obtenida en la implementación y utilización de Modula se aprovechó para Modula-2 (Wirth, 1983), un lenguaje para implementación de sistemas de propósito más general. El lenguaje C se ha desdoblado en muchas otras variantes; la más importante es C++, que soporta directamente la programación orientada al objeto. Otros nuevos lenguajes de nota son: PEARL, empleado extensamente para aplicaciones de control de procesos en Alemania; Mesa (Xerox Corporation, 1985), utilizado por Xerox en su equipamiento de automatización de oficinas; y CHILL (CCITT, 1980), desarrollado para la programación de aplicaciones de telecomunicaciones, en respuesta a los requisitos de CCITT. Incluso existe una versión de Basic para tiempo real, que aun careciendo de muchas características que habitualmente se asocian a los lenguajes de alto nivel (como los tipos definidos por los usuarios), proporciona elementos de programación concurrente y algunas otras características relacionadas con el tiempo real.

Con la llegada de Internet, el lenguaje de programación Java ha obtenido mucha popularidad. A pesar de no ser adecuado, inicialmente, para la programación de tiempo real, se ha invertido recientemente mucho trabajo para producir versiones de Java para tiempo real (US National Institute of Standards and Technology, 1999; Bollella et al., 2000; J Consortium, 2000).

Modula, Modula-2, PEARL, Mesa, CHILL, Java y Ada son todos ellos lenguajes de programación concurrente de alto nivel que incluyen aspectos orientados a ayudar al desarrollo de sistemas computacionales embebidos. El lenguaje occam, por contra, es un lenguaje mucho más reducido. No da un soporte auténtico para la programación de sistemas grandes y complejos, aunque proporciona las estructuras de control habituales, así como procedimientos no recursivos. El desarrollo de occam ha estado muy ligado al del *transputer* (May y Shepherd, 1984), un procesador con memoria integrada en el chip y controladores de enlaces que permiten construir fácilmente sistemas *multi-transputer*. Occam tuvo un importante papel en el campo emergente de las aplicaciones embebidas distribuidas débilmente acopladas. Sin embargo, en los últimos años su popularidad ha ido desvaneciéndose.

2.5.4 Criterios generales de diseño de lenguajes

Aunque un lenguaje de tiempo real pudiera estar diseñado, en origen, para cumplir los requisitos de la programación de informática embebida, raramente se limita a esta área. La mayoría de los lenguajes de tiempo real se emplean también como lenguajes para la implementación de sistemas para aplicaciones, tales como compiladores y sistemas operativos.

Young (1982) detalla los siguientes seis criterios, a veces discutidos, como la base del diseño de un lenguaje de tiempo real: seguridad, legibilidad, flexibilidad, sencillez, portabilidad y eficiencia. En los requisitos originales de Ada aparece también una lista similar.

Seguridad

La seguridad del diseño de un lenguaje es la medida del grado en el que el compilador, o el sistema de soporte en tiempo de ejecución, pueden detectar automáticamente los errores de programación. Obviamente, existe un límite en la cantidad y en los tipos de errores que pueden ser detectados por el sistema del lenguaje; por ejemplo, no es posible detectar de modo automático

los errores de lógica del programador. Un lenguaje seguro debe, en consecuencia, estar bien estructurado y ser legible, de modo que tales errores puedan ser localizados fácilmente.

Entre los beneficios de la seguridad encontramos:

- La detección temprana de errores en el desarrollo del programa, lo que supone una reducción general del coste.
- Las comprobaciones de tiempo de compilación no sobrecargan la ejecución del proceso, y un programa se ejecuta más veces de las que se compila.

El inconveniente de la seguridad es que puede llegar a complicar el lenguaje e incrementar tanto el tiempo de compilación como la complejidad del compilador.

Legibilidad

La legibilidad de un lenguaje depende de un conjunto de factores, como una elección adecuada de palabras clave, la facilidad para definir tipos, y los mecanismos de modularización de programas. Según apunta Young:

> ... el objetivo es ofrecer un lenguaje para la notación suficientemente claro, que permita que los principales conceptos del funcionamiento de un programa puedan ser asimilados tan sólo leyendo el texto del programa, sin tener que recurrir a diagramas de flujo y descripciones textuales auxiliares. (Young, 1982)

Los beneficios de una buena legibilidad son:

- Menores costes de documentación.
- Mayor seguridad.
- Mayor facilidad de mantenimiento.

El principal inconveniente es la mayor longitud de los programas.

Flexibilidad

Un lenguaje debe ser lo suficientemente flexible para permitir al programador expresar todas las operaciones oportunas de una forma directa y coherente. De otro modo, como con los lenguajes secuenciales anteriores, el programador tendrá que recurrir a funciones del sistema operativo o a fragmentos en código máquina para obtener el resultado deseado.

Simplicidad

La simplicidad es un objetivo nunca suficientemente alabado de cualquier sistema, ya sea la proyectada estación espacial internacional o una simple calculadora. En los lenguajes de programación, la simplicidad aporta las ventajas de:

- Minimizar el esfuerzo de producción de compiladores.
- Reducir el coste asociado a la formación de los programadores.
- Disminuir la posibilidad de cometer errores de programación como consecuencia de una mala interpretación de las características del lenguaje.

La flexibilidad y la simplicidad pueden ponerse también en relación con el **poder expresivo** (la posibilidad de ofrecer soluciones a un gran abanico de problemas) y con la **facilidad de uso** del lenguaje.

Portabilidad

Un programa deberá ser independiente, en la medida de lo posible, del hardware sobre el que se ejecuta. Una de las principales autoproclamas de Java es que los programas se compilan una sola vez y se ejecutan en muy diversos sitios. Para un sistema de tiempo real, esto es difícil de lograr –incluso con la llegada de los códigos binarios portables (Venners, 1999; X/Open Company Ltd., 1996)–, dado que una parte importante de cualquier programa estará involucrada en la manipulación de recursos hardware. A pesar de ello, un lenguaje deberá ser capaz de aislar la parte del programa que depende de la máquina de la parte *independiente* de la máquina.

Eficiencia

En un sistema de tiempo real, los tiempos de respuesta deberán estar garantizados; así pues, el lenguaje deberá permitir producir programas eficientes y predecibles. Hay que evitar aquellos mecanismos que conduzcan a sobrecargas impredecibles del tiempo de ejecución. Obviamente, los requisitos de eficiencia deberán estar compensados con los requisitos de seguridad, flexibilidad y legibilidad.

2.6 Prueba

Dados los requisitos de alta confiabilidad que conforman la esencia de la mayoría de los sistemas de tiempo real, queda claro que las pruebas deben ser extremadamente exigentes. Una estrategia de prueba completa incluirá muchas técnicas, la mayoría de las cuales son aplicables a cualquier producto de software. Partimos de la suposición de que el lector se encuentra familiarizado con estas técnicas.

El problema de los programas concurrentes de tiempo real es que los errores de sistema más difíciles de tratar surgen habitualmente de sutiles interacciones entre procesos. A menudo, los errores dependen del instante de tiempo, y sólo se manifestarán en situaciones poco comunes. La ley de Murphy sostiene que estos estados poco comunes suelen ser también de importancia crucial, y se dan sólamente cuando el sistema controlado está en alguna forma de estado crítico. Debiéramos, quizás, hacer hincapié aquí en que los métodos apropiados de diseño formal no evitan la necesidad de hacer pruebas: son estrategias complementarias.

Las pruebas, desde luego, no se restringen al sistema final ya ensamblado. La descomposición contemplada en el diseño y que se manifiesta en la modularidad del programa (y de los módulos) proporciona una arquitectura natural para la prueba de los componentes. De particular importancia (y dificultad) en los sistemas de tiempo real es probar no sólo el correcto comportamiento en el entorno correcto, sino también comprobar la confiabilidad de su comportamiento en un entorno arbitrariamente incorrecto. Habrá que contrastar todos los modos de recuperación e investigar los efectos de los errores simultáneos.

Como ayuda para cualquier tarea de prueba compleja, el disponer de un entorno de prueba presenta muchos atractivos. Para el software, tales entornos de prueba se denominan simuladores.

2.6.1 Simuladores

Un simulador es un programa que imita las acciones del sistema en el que se encuentra inmerso el software de tiempo real. Simula la generación de interrupciones y efectúa otras acciones de E/S en tiempo real. Empleando un simulador, podrán crearse situaciones de comportamiento «normales» y anormales. Incluso cuando el sistema final haya sido completado, determinados estados de error solo podrán ser experimentados de un modo seguro mediante un simulador. Un ejemplo obvio es la fusión del núcleo de un reactor nuclear.

Los simuladores pueden reproducir con exactitud la secuencia de eventos que se esperan del sistema real. Además, es posible repetir experimentos en modos que normalmente son imposibles de obtener en el funcionamiento en vivo. Desgraciadamente, para recrear de modo realista acciones simultáneas se hace necesario disponer de un simulador multiprocesador. Además, hay que tener en cuenta que para aplicaciones muy complicadas pudiera no ser posible construir un simulador apropiado.

Aunque los simuladores no exigen requisitos de muy alta fiabilidad, son, en cualquier aspecto, sistemas de tiempo real por propio derecho. Ellos mismos deberán ser probados ampliamente, aunque eventualmente podamos tolerar algunos errores. Una técnica muy habitual en sistemas de tiempo real sin requisitos de alta fiabilidad (como un simulador de vuelo) es eliminar la protección de exclusión mutua de los recursos compartidos. Esto proporciona una mayor eficiencia, al precio de asumir fallos intermitentes. La construcción de un simulador viene facilitada también por el modelo de proceso del mismo sistema embebido. El Capítulo 15 muestra cómo podemos considerar a los dispositivos externos como procesos hardware cuyas interrupciones se corresponden con la primitiva de sincronización disponible. Si se ha seguido este modelo, el remplazo de un proceso hardware por uno software es relativamente directo.

Aparte de lo dicho, los simuladores son sistemas caros de desarrollar, y suponen un esfuerzo considerable. Además, pueden requerir hardware especial. En el proyecto de la lanzadera espacial de la NASA los simuladores costaron más que el mismísimo software de tiempo real. Finalmente, resulta un dinero bien gastado, por los muchos errores encontrados en el sistema durante las horas de «vuelo» simulado.

2.7 Prototipado

La aproximación estándar en «cascada» al desarrollo del software (esto es, requisitos, especificación, diseño, implementación, integración, prueba y despliegue o implantación) conlleva el problema fundamental de que los errores en los requisitos iniciales o en las fases de especificación sólo se reconocen al entregar el producto (o en el mejor de los casos, durante las pruebas). Corregir estas carencias en esta etapa tan tardía es muy largo y costoso. El prototipado intenta descubrir estos fallos lo antes posible, cuando presentamos al cliente una «portada» del sistema.

El principal objetivo de un prototipo es ayudar a cerciorarnos de que en la especificación de requisitos ha sido captado lo que realmente quiere el cliente. Esto tiene dos aspectos:

- ¿Es correcta la especificación de requisitos (en términos de lo que desea el cliente)?
- ¿Es completa la especificación de requisitos (ha incluido el cliente todo lo que desea)?

Uno de los beneficios de la puesta en marcha de un prototipo es que el cliente puede experimentar situaciones que han sido entendidas sólo vagamente. Es casi inevitable que en el desarrollo de esta actividad haya que realizar cambios en los requisitos. Concretamente, saldrán a relucir nuevas condiciones de error y modos de recuperación de fallos.

Si las técnicas de especificación empleadas no son formales, el prototipado puede utilizarse para darnos confianza en la consistencia de nuestro diseño global. Así, si usamos un gran diagrama de flujo, un prototipo podría ayudarnos a comprobar que todas las conexiones necesarias están en su lugar y que todos los fragmentos de datos que fluyen por el sistema visitan efectivamente las actividades requeridas.

Para cumplir con los objetivos de coste, debe ser posible construir el prototipo por menos dinero (mucho menos) que el que cuesta el auténtico software de implementación. Por esto, se desaconseja utilizar los mismos métodos de diseño que los estándares que se aplicarán al sistema construido. Podemos lograr una reducción significativa en los costes mediante el uso de lenguajes de más alto nivel. El lenguaje APL (Iverson, 1962) ha sido un lenguaje popular para el prototipado; también se han usado lenguajes de programación funcionales y lógicos. Éstos presentan la ventaja de que capturan de hecho el comportamiento funcional importante del sistema sin detenerse sobre los aspectos no funcionales. Más recientemente, las herramientas de diseño vienen integrando mecanismos de generación de código. Sin embargo, no suelen producir código de la calidad necesaria para la implementación final, aunque pueden servir para un prototipo. La desventaja más evidente de estos prototipos es que no tienen en cuenta los aspectos de tiempo real de la aplicación. Para obtener esto, necesitamos una simulación del sistema.

Ya se ha indicado que probar que un sistema cumple todas las restricciones temporales bajo todas las condiciones de funcionamiento es muy difícil. Un método para examinar si un diseño es realizable es intentar la simulación del comportamiento del software. Realizando hipótesis sobre las estructuras del código y la velocidad de ejecución del procesador, es posible construir un modelo que emule las características de tiempo de ejecución; por ejemplo, que muestre la tasa

máxima de interrupciones que podrán manejarse mientras se siguen conservando las restricciones de tiempo.

Emular un sistema de tiempo real grande es muy caro; su desarrollo es costoso, y cada ciclo de funcionamiento del emulador requerirá considerables recursos de cómputo. Generalmente, serán necesarios muchos ciclos de funcionamiento. Sin embargo, este coste puede compensarse por las consecuencias económicas y sociales de construir un sistema de tiempo real erróneo.

2.8 Interacción hombre-máquina

Bajo un punto de vista amplio, todos los sistemas de tiempo real incluyen o afectan a los seres humanos. La mayoría, de hecho, involucran comunicación directa entre el software en ejecución y uno o más operadores humanos. Como el comportamiento humano introduce la mayor de las fuentes de variabilidad en un sistema en funcionamiento, es lógico que el diseño del componente HCI (*Human-Computer Interaction*; interacción hombre-máquina) sea uno de los más críticos de toda la estructura. Hay muchos ejemplos de malos diseños de HCI y de las consecuencias, a veces trágicas, que se derivan de su debilidad. Por ejemplo, en el incidente nuclear conocido como «de la Isla de las Tres Millas», los operadores ignoraron ciertos fallos ante la incapacidad de manejar la avalancha de información que se mostraba durante un encadenamiento de situaciones críticas (Kemeny et al., 1979).

La línea de investigación sobre los principios de diseño en los que basar la construcción de componentes HCI es muy activa. Tras haber estado en segundo plano durante muchos años, el HCI es apreciado hoy en día como un punto central para la producción de software bien diseñado (de cualquier software). La primera cuestión importante es la modularidad; las actividades de HCI deben encontrarse aisladas y con especificaciones de interfaces bien definidas. Estas interfaces constan a su vez de dos componentes, siendo las funciones de cada una de ellas bien diferentes. En primer lugar, se definen aquellos objetos que se transfieren entre el operador y el software. En segundo lugar (muy importante), es necesario especificar cómo hay que presentar y recibir estos objetos del usuario. Una definición rigurosa del primer tipo de componente de la interfaz debe incluir predicados sobre las acciones permitidas para un operador en cada estado del sistema. Por ejemplo, ciertas operaciones podrían requerir autorización (o sólo podrían ser llevadas a cabo sensatamente o de modo seguro) cuando el sistema estuviera correctamente dispuesto. Por poner un ejemplo, un avión a control remoto ignorará una operación que le conduzca fuera de su ruta de vuelo segura.

Una importante cuestión de diseño en cualquier sistema interactivo es quién está al mando. En un extremo, la persona podría dirigir explícitamente al computador para que realizara funciones concretas, mientras que en el otro extremo el computador podría tener el control completo (aunque eventualmente podría solicitar información al operador humano). Obviamente, la mayoría de los sistemas de tiempo real estarán entre estos dos extremos; se les conoce como sistemas de **iniciativa mixta**. En algunas ocasiones el usuario está al mando (por ejemplo, cuando se proporciona un nuevo mandato), y en otras el sistema controla el diálogo (por ejemplo, cuando se extraen los datos necesarios para efectuar una operación que se acaba de solicitar).

El principal objetivo en el diseño de la componente de interfaz es la captura de todos los errores derivados del usuario en el software de control de la interfaz, y no en el software de aplicación o en el resto del sistema de tiempo real. Si esto es así, el software de aplicación podrá presuponer un **usuario perfecto**, y en consecuencia su diseño e implementación se harán más simples (Burns, 1983). En nuestra discusión, el término **error** se refiere a una acción no intencionada. Reason (1979) lo denomina **lapsus** (slip), y usa el término **equivocación** para referirse a una acción errónea deliberada. Aunque una interfaz pueda ser capaz de bloquear las equivocaciones conducentes a situaciones peligrosas, no será posible reconocer como una equivicación un cambio «serio» de un parámetro de la operación. La única forma de eliminar estos errores es mediante la formación apropiada y la supervisión de los operadores.

Aunque estos lapsus ocurrirán inevitablemente (y en consecuencia debemos protegernos contra ellos), su frecuencia puede reducirse de modo significativo si el segundo componente de la interfaz, el auténtico módulo de entrada/salida del operador, está bien definido. En este contexto, sin embargo, el término «bien definido» no es fácil de establecer. La especificación de un módulo de E/S cae esencialmente bajo la competencia de la psicología. El punto de arranque es un modelo del operador o usuario final. Desgraciadamente, hay muchos modelos diferentes; Rouse (1981) proporciona una revisión interesante. De la comprensión de estos modelos, podemos establecer ciertos principios de diseño, entre los que encontramos estos tres importantes:

(1) Predecibilidad: es preciso proporcionar datos suficientes al usuario para que éste conozca de modo exacto los efectos de cada operación a partir del conocimiento sobre esta operación.

(2) Conmutatividad: el orden en que el usuario cumplimenta los parámetros de una operación no deberá afectar al sentido de la operación.

(3) Sensibilidad: si el sistema puede presentar diferentes modos de operación, deberá mostrarse siempre el modo actual.

Los dos primeros principios son aportados por Dix et al. (1987). Los sistemas de tiempo real tienen la dificultad añadida de que los operadores humanos no son los únicos actores que cambian el estado del sistema. Una interrupción puede causar un cambio de modo, y la naturaleza asíncrona de la entrada de operación podría ocasionar que un mandato, que era válido cuando el operador lo inició, no esté disponible para el nuevo modo.

Gran parte del trabajo sobre HCI ha estado relacionado con el diseño de pantallas para la presentación no ambigua de datos y para que la captura de datos de entrada se efectúe con el menor número de pulsaciones. Existen, sin embargo, otros factores relacionados con la ergonomía de la estación de trabajo, así como cuestiones más amplias sobre el entorno de trabajo de los operadores. Los diseños de pantallas deben ser «probados» mediante un prototipo, pero la satisfacción en el trabajo es una métrica que sólamente podemos aplicar sobre intervalos de tiempo de mayor duración que los que se ofrecen al experimentar con un prototipo. Puede que los operadores trabajen en turnos de ocho horas, cinco días a la semana, año tras año. Si además añadimos un trabajo agobiante (como el de controlar el trafico aéreo), la satisfacción en el trabajo y las prestaciones se convierten en factores críticos que dependen de:

- La multitud de tareas interdependientes que hay que llevar a cabo.
- El nivel de control de que dispone el operador.
- El grado en que el operador comprende la operación del sistema completo.
- El número de tareas constructivas que se le permite realizar al operador.

La comprensión puede mejorarse si presentamos siempre al operador una imagen de «qué está pasando». Y ciertamente, es posible construir sistemas de control de procesos que ilustren el comportamiento del sistema en forma gráfica, pictográfica, o como una hoja de cálculo. El operador puede experimentar con los datos para modelar formas de mejorar las prestaciones del sistema. Es muy razonable incorporar tales **sistemas de ayuda a la decisión** en el bucle de control, de modo que el operador pueda ver el efecto de cambios menores sobre los parámetros de la operación. De esta forma, se puede incrementar la productividad y mejorar el entorno de trabajo de los operadores. Además, también se reduce el número de equivocaciones que se cometen.

2.9 Gestión del diseño

Este capítulo ha intentado proporcionar una panorámica de las cuestiones importantes asociadas con el diseño de sistemas de tiempo real. Sólo obtendremos un producto fiable y correcto si las actividades de especificación, diseño, implementación y prueba se llevan a cabo con una completa y alta calidad. Hay un amplio rango de técnicas que pueden ayudar a lograr esta calidad. Una cuestión primordial en este libro es el uso apropiado de un lenguaje de alto nivel bien definido. Para obtener el resultado deseado es preciso gestionar adecuadamente tanto la programación como el diseño. Los lenguajes de programación concurrente modernos tienen un papel importante en este proceso de gestión.

La clave para lograr la calidad descansa en la adecuada verificación y validación. En cada etapa de diseño e implementación, es preciso usar procedimientos bien definidos para asegurar que las técnicas necesarias se realizan correctamente. Donde se precisen fuertes garantías, pueden usarse demostradores de teoremas y comprobadores de modelos para verificar formalmente los componentes especificados. Más ampliamente se utilizan muchas otras actividades estructuradas; por ejemplo, análisis de peligros, análisis de árboles de fallos en software, análisis de modos de fallo y de efectos, e inspecciones de código. Cada vez hay más necesidad de herramientas de software como ayuda para la validación y para la verificación. Tales herramientas realizan un trabajo importante, y su uso puede reducir la necesidad de inspección humana. Así pues, estas herramientas deben ser de una calidad extremadamente alta.

Las herramientas de apoyo, sin embargo, no pueden substituir a un equipo de personas experimentado y bien entrenado. La aplicación de técnicas rigurosas de ingeniería de software y el empleo de las características del lenguaje que se describen en este libro establecen la diferencia en cuanto a sistemas de tiempo real de calidad y asequibles. Muchas muertes han sido atribuidas a errores de software. Es posible detener una futura epidemia, pero sólo si la industria se aparta de los procedimientos *ad hoc*, los métodos informales, y los lenguajes inapropiados de bajo nivel.

Resumen

En este capítulo se han resumido las principales etapas del diseño e implementación de los sistemas de tiempo real. Entre éstas están, por lo general, la especificación de requisitos, el diseño del sistema, el diseño detallado, la codificación y la prueba. Los requisitos de alta fiabilidad de los sistemas de tiempo real nos dictan que, en la medida de lo posible, empleemos métodos rigurosos.

La gestión del desarrollo de los sistemas complejos de tiempo real requiere la correcta aplicación de la descomposición, el encapsulamiento y la abstracción. Un método de diseño jerárquico permite pues aislar los subcomponentes, o módulos, tanto basados en procesos como en objetos. Ambas formas de módulo deberán exhibir una fuerte cohesión y undébil acoplamiento.

La implementación, que es el objetivo principal de este libro, requiere el uso de un lenguaje de programación. Los lenguajes de tiempo real primitivos carecían de poder expresivo para poder manejar adecuadamente este dominio de aplicación. Los lenguajes más modernos han intentado incorporar mecanismos de concurrencia y gestión de errores. En los capítulos siguientes se discuten estas características. Los siguientes criterios generales se consideran un fundamento útil para el diseño de los lenguajes de tiempo real:

- Seguridad.
- Legibilidad.
- Flexibilidad.
- Simplicidad.
- Portabilidad.
- Eficiencia.

Independientemente del rigor en el proceso de diseño, es preciso realizar pruebas adecuadas. Como ayuda para esta actividad, las implementaciones de prototipos, los simuladores y los emuladores, juegan un papel importante.

Como punto final del capítulo, se presta especial atención a la interfaz hombre-máquina. Durante demasiado tiempo, se ha tratado este tema de forma poco rigurosa, y la aplicación sistemática de los principios de la ingeniería brillaba por su ausencia. Esta situación está cambiando, con la concienciación general de que la interfaz es una fuente importante de errores potenciales, y de que el número de éstos puede reducirse mediante la aplicación de los conocimientos actuales, derivados de la investigación y el desarrollo.

En muchos sentidos, este capítulo es diferente a los demás. Se han presentado más situaciones problemáticas y cuestiones ingenieriles de las que posiblemente pudieran abordarse en un solo libro. Este amplio recorrido a través del «proceso de diseño» está orientado a situar en el

contexto adecuado el resto del libro. A partir de ahora, la discusión se centrará en cuestiones del lenguaje y en las actividades de la programación, de modo que el lector sea capaz de comprender el «producto final» del diseño, y juzgar hasta qué punto son apropiadas las metodologías actuales, las técnicas, y las herramientas.

Lecturas complementarias

Booch G., Rumbaugh J., y Jacobson I. (2000), *The Unified Modeling Language User Guide*, Harlow: Addison-Wesley.

Burns A., y Wellings A. J. (1995), *Hard Real-Time HOOD: A Structured Design Method for Hard Real-time Ada Systems*, New York: Elsevier.

Cooling J. E. (1995), *Software Design for Real-Time Systems*, London: International Thompson Computer Press.

Dix A. (1991), *Formal Methods for Interactive Systems*, New York: Academic Press.

Douglass B. P. (1999), *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns*, Harlow: Addison-Wesley.

Gomaa H. (1993), *Software Design Methods for Concurrent and Real-Time Systems*, Reading, MA: Addison-Wesley.

Jacobson I., Booch G., y Rumbaugh J. (2000), *The Unified Software Development Process*, Harlow: Addison Wesley Longman.

Jones C. B. (1990), *Systematic Software Development Using VDM*, New York: Prentice Hall.

Joseph M. (ed) (1996), *Real-Time Systems: Specification, Verification and Analysis*, London: Prentice Hall, 1996.

Levi S. T., y Agrawala A. K. (1990), *Real-Time Systems Design*, New York: McGraw-Hill.

Robinson P. (1992), *Hierarchical Object Oriented Design*, New York: Prentice Hall.

Rosen J.-P. (1997), *HOOD: An Industrial Approach to Software Design*, HOOD Technical Group.

Schneider, S. (1999), *Concurrent and Real-time Systems: The CSP Approach*, New York: Wiley.

Woodcock J. (1994), *Using Standard Z*, Hemel Hempstead: Prentice Hall.

Ejercicios

2.1 Hasta qué punto la elección de un método de diseño debería estar influenciada por:

(a) Un probable lenguaje de implementación.

(b) Las herramientas de apoyo.

(c) Los requisitos de fiabilidad de la aplicación.

(d) Los requisitos de formación del personal.

(e) Las consideraciones del marketing.

(f) Las experiencias previas.

(g) El coste.

2.2 Además de los criterios indicados en este capítulo, ¿qué otros factores podríamos tener en cuanta al evaluar los lenguajes de programación?

2.3 ¿En qué momento del proceso de diseño deberían utilizarse las vistas que se presentan al usuario final?

2.4 ¿Deberían ser los ingenieros de software responsables de las consecuencias de los errores de sus sistemas de tiempo real?

2.5 Cada nuevo medicamento debe pasar pruebas y validaciones hasta que puede ser presentado al mercado. ¿Deberían estar sujetos los sistemas de tiempo real a una legislación similar? Si una aplicación dada es demasiado compleja para ser simulada, ¿debería ser construida?

2.6 ¿Debería ser el lenguaje Ada el único que pudiera utilizarse en los sistemas embebidos de tiempo real?

2.7 ¿En qué medida permite UML el diseño y el análisis de los sistemas de tiempo real estrictos?

2.8 ¿Es UML una notación de diseño estructurada, o formal?

Capítulo 3

Programar lo pequeño

Al considerar las características de los lenguajes de alto nivel, resulta útil distinguir entre aquéllas que ayudan en la descomposición de procesos y las que facilitan la programación de componentes bien definidos. Estas dos características han sido descritas como:

- Soporte para programar lo grande.
- Soporte para programar lo pequeño.

Programar lo pequeño es una actividad bien conocida, y será tratada en este capítulo dentro del contexto de un repaso de los lenguajes ADA 95 (denominado Ada a lo largo de este libro), Java 1.2 (denominado Java en adelante), ANSI C y occam2. El Capítulo 4 está dedicado a la programación de lo grande, y tratará el tema más problemático de la gestión de sistemas complejos.

3.1 Repaso de Ada, Java, C, y occam2

En un libro dedicado a los sistemas de tiempo real y a sus lenguajes de programación, resulta imposible dar una descripción detallada de todos los lenguajes utilizados en este dominio de aplicación. Se ha optado pues por limitar el estudio a cuatro lenguajes de programación de alto nivel. Ada es importante, ya que su uso se está extendiendo en los sistemas cuya seguridad es crítica; Java se está convirtiendo en el lenguaje estándar *de facto* para las aplicaciones basadas en Internet. C (y su derivado C++) es quizás el lenguaje de programación más popular utilizado hoy en día, y occam2 es el lenguaje de propósito general que mejor encarna los formalismos de CSP. Occam2 también está diseñado específicamente para la ejecución en múltiples computadores, lo que cada vez es más usual, y crece su importancia en el dominio del tiempo real. A pesar de esto, se considerarán las características específicas de otros lenguajes cuando resulte apropiado.

El resumen aquí presentado presupone un conocimiento de lenguajes tipo Pascal. Se aportará un grado de detalle suficiente de cada lenguaje como para entender los programas que se verán

después en el libro a modo de ejemplos. Para una descripción comprensible de cada lenguaje, el lector debe dirigirse a la sección de libros especializados en cada lenguaje incluidos en las lecturas complementarias que aparecen al final de este capítulo.

3.2 Convenciones léxicas

Los programas se escriben una vez, pero se leen varias veces; de esto se desprende que el estilo léxico de la sintaxis del lenguaje debiera estar más orientado al lector que al que lo escribe. Una forma simple de incrementar la legibilidad es utilizar nombres que aporten significado. Los lenguajes no deberían restringir, por lo tanto, la longitud de los identificadores; Ada, Java, C y occam2 no lo hacen. La forma de un nombre también se puede mejorar utilizando un separador. Ada, Java y C permiten incluir un signo «_» en los identificadores, mientras que los identificadores occam2 pueden incluir un signo «.» (ésta es una elección desafortunada, ya que el signo «.» se utiliza a menudo en los lenguajes para indicar un subcomponente, como un campo dentro de un registro). Si un lenguaje no admite separador alguno, se recomienda utilizar la técnica de mezclar caracteres en mayúsculas y en minúsculas. Aunque Java no permite la inclusión de un signo «_», el estilo predominante utiliza una mezcla de mayúsculas y minúsculas. Los siguientes son ejemplos de identificadores.

```
Nombre_Ejemplo_En_Ada
nombreEjemploEnJava
nombre_ejemplo_en_C
nombre.ejemplo.en.occam2
```

Un ejemplo clásico de pobre convención léxica es el de FORTRAN. Este lenguaje permite el carácter en blanco como separador. Por poner un caso, un simple error al teclear un programa (un signo «.» en lugar de un signo «,») terminó, según se dice, con la pérdida de la sonda norteamericana Viking en su camino a Venus. En lugar de la siguiente línea:

```
DO 20 I = 1,100
```

que es un bucle (salta a la etiqueta `20` mientras `I` toma valores de `1` a `100`), se compiló una asignación (el operador asignación es «=»):

```
DO 20 I = 1.100
```

o, puesto que los espacios pueden ser ignorados en los identificadores,

```
DO20I = 1.100
```

Las variables en FORTRAN no necesitan ser definidas; los identificadores que comienzan con «D» son asumidos como reales, y 1.100 es un literal real.

No es sorprendente que los lenguajes modernos exijan que las variables se definan explícitamente.

3.3 Estilo general

Los cuatro lenguajes revisados aquí están, en mayor o menor medida, estructurados en bloques. Un bloque en Ada consiste en

(1) La declaración de objetos que son locales en ese bloque (si no existen tales objetos, entonces esta parte de la declaración puede ser omitida).

(2) Una secuencia de sentencias.

(3) Un conjunto de gestores de excepciones (del mismo modo, puede ser omitido si está vacío).

Un esquema de uno de estos bloques es:

```
declare
   <parte declarativa>
begin
   <secuencia de sentencias>
exception
   <gestores de excepciones>
end;
```

Los gestores de excepciones capturan errores que han ocurrido en el bloque. Son considerados en detalle en el Capítulo 6.

Dentro de un programa Ada se puede colocar un bloque en cualquier lugar donde pudiera darse una sentencia. Pueden utilizarse de forma jerárquica, y facilitan la descomposición dentro de una unidad de programa. El sencillo ejemplo siguiente ilustra el modo en que se introduce una nueva variable entera `Temp` para intercambiar los valores contenidos en los dos enteros `A` y `B`. Hay que destacar que un comentario en Ada comienza con un guión doble, y abarca hasta el final de la línea.

```
declare
   Temp: Integer:= A; -- valor inicial dado a la variable temporal
begin
   A:= B;             -- := es el operador asignación
   B:= Temp;
end;                  -- no existen excepciones
```

En C y en Java, un bloque (o una sentencia compuesta) se delimita por un par de llaves ({ y }), y tiene la siguiente estructura:

```
{
   < parte declarativa >
   < secuencia de sentencias >
}
```

Como en Ada, Java puede tener gestores de excepciones al final de un bloque si el bloque es etiquetado como un bloque «try» (véase el Capítulo 6). Cada sentencia puede ser a su vez una sentencia compuesta. El código de intercambio anterior podría ser como sigue:

```
{
   int temp = A; /* declaración e inicialización */
   /* En C y en Java (y en occam2) el tipo aparece antes */
   /* mientras que en Ada, aparece después del nombre de la variable. */
   A =B;
   B = temp;
}
```

Hay que destacar que el operador de asignación en C y en Java es «=», y que los comentarios están delimitador por «/*» y por «*/». Java también permite utilizar «//» para comentarios que finalizan con la línea. A lo largo de este libro, se utilizará «//» para comentar los programas Java, de forma que sean fácilmente distinguibles de los programas escritos en C.

Antes de considerar el equivalente occam2 de este programa, hay que realizar algunas consideraciones sobre el lenguaje. Ada y Java son lenguajes con los que se pueden escribir programas concurrentes. C es un lenguaje secuencial que puede ser utilizado junto con el sistema operativo para crear procesos concurrentes. Por contra, todos los programas occam2 son concurrentes. Lo que en Ada, Java y C sería una secuencia de sentencias, en occam2 es una secuencia de procesos. Esta diferencia es fundamental a la hora de comparar la naturaleza de los modelos de concurrencia en los cuatro lenguajes (véanse los Capítulos 7, 8 y 9), pero no impide la comprensión de los elementos primitivos del lenguaje.

Occam2, como Ada, es un lenguaje totalmente estructurado en bloques. Cualquier proceso puede ser precedido por la declaración de objetos que serán usados por ese proceso. Para intercambiar dos enteros (`INT` en occam2) se necesita un constructor `SEQ` que especifica que las sentencias que siguen deben ser ejecutadas de forma secuencial:

```
INT temp: -- Una declaración termina en dos puntos.
SEQ
  temp := A
  A :=B
  B := temp
```

Resulta interesante estudiar los diferentes modos en que los cuatro lenguajes separan las distintas acciones. Del mismo modo que Pascal, Ada utiliza un punto y coma como final de una sentencia (por eso encontramos uno al final de `B := Temp;`). C y Java también emplean un punto y coma como terminador de sentencias (pero no se coloca un punto y coma al final de una sentencia compuesta, aunque Java lo tolera). Occam2 no utiliza el punto y coma para nada: exige que cada acción (proceso) ocupe una línea separada. Más aún, el uso de la identación, que mejora (de forma importante) la legibilidad en Ada, Java y C, tiene un significado sintáctico en occam2. Las tres sentencias de asignación en el fragmento de código anterior tienen que comenzar en la misma columna de la `Q` de `SEQ`.

3.4 Tipos de datos

De la misma manera que todos los lenguajes de alto nivel, Ada, Java, C y occam2 conciben programas para manipular objetos que han sido abstraídos de su implementación hardware final. Los programadores no necesitan estar al tanto de la representación o la ubicación de las entidades que sus programas manipulan. Más aún, al distribuir esas entidades entre distintos tipos, el compilador puede comprobar un uso inconsistente, y por lo tanto incrementar la seguridad asociada con la utilización de los lenguajes.

Ada permite la declaración de constantes, tipos, variables, subprogramas (procedimientos y funciones) y paquetes. Los subprogramas serán tratados más adelante en este mismo capítulo, mientras que los paquetes se describen en el Capítulo 4. La utilización de constantes, tipos y variables es similar a la de Pascal, excepto que pueden aparecer en cualquier orden, siempre que un objeto sea declarado antes de ser referenciado. De forma similar, C y Java permiten definir clases y paquetes (véase también el Capítulo 4). En comparación con estos lenguajes, el modelo de tipos de occam2 es más restrictivo; en particular, no se permiten los tipos definidos por el usuario.

3.4.1 Tipos discretos

La Tabla 3.1 muestra los tipos discretos predefinidos soportados en los cuatro lenguajes.

Se dispone de todos los operadores habituales para esos tipos. Hay que destacar que el tipo `char` de Java soporta caracteres Unicode, y por lo tanto es equivalente al tipo `Wide_Character` de Ada o al `wchar_t` de C.

Ada, Java y occam2 son fuertemente tipados (esto es, las asignaciones y las expresiones deben involucrar objetos del mismo tipo), pero soportan las conversiones explícitas de tipos. El lenguaje C no es seguro respecto a los tipos; por ejemplo, un entero puede ser asignado a un entero

Tabla 3.1. Tipos discretos.

Ada	Java	C	Occam2
`Integer`	`int`	`int`	`INT`
	`short`	`short`	`INT16`
	`long`	`long`	`INT32`
			`INT64`
	`byte`	`BYTE`	
`Boolean`	`boolean`		`BOOL`
`Character`		`char`	
`Wide_Character`	`char`	`wchar_t`	

corto sin una conversión explícita de tipo. Java permite la conversión sólo cuando no se produce una pérdida de información; por ejemplo de entero a entero largo, pero no en el otro sentido.

Tanto Ada como C también permiten que los tipos enteros básicos sean con signo o sin signo. Inicialmente son con signo, pero se pueden crear tipos sin signo (o módulo). Aunque el lenguaje no lo requiere, una implementación Ada podría soportar los tipos `Short_Integer` y `Long_Integer`.

Además de estos tipos predefinidos, Ada y C permiten la definición de tipos enumerados. Ni occam2 ni Java proporcionan esta posibilidad, aunque se han realizado propuestas para añadir esta característica al lenguaje Java (Cairns, 1999). En C, las constantes enumeradas deben ser determinadas de forma unívoca, y son realmente poco más que definiciones de constantes enteras; sin embargo, el modelo Ada, al permitir la sobrecarga de nombres, no es tan restrictivo. Ambos lenguajes proporcionan medios para la manipulación de objetos de esos tipos citados: C a través de las operaciones estándares de enteros, y Ada mediante el uso de atributos. (Los atributos se utilizan en Ada para obtener información sobre los tipos y los objetos.) Los siguientes ejemplos ilustran estos aspectos.

```
/* C */
{
   typedef enum {planox, planoy, planoz} dimension;
   /* typedef introduce un nuevo nombre de tipo, */
   /* enum indica que es un tipo enumerado; */
   /* dimension es el nuevo nombre del tipo */

   dimension linea, fuerza;
   linea = planox;
   fuerza = linea + 1;
   /* fuerza tiene el valor planoy porque el compilador */
   /* genera los siguientes literales enteros planox = 0, */
   /* planoy = 1, planoz = 2 */
}
```

```
-- Ada
type Dimension is (planoX,planoY,planoZ);
type Mapa is (planoX,planoY);
Linea,Fuerza : Dimension;
Cuadricula : Mapa;
begin
   Linea := planoX;
   Fuerza := Dimension'Succ(planoX);
   -- Fuerza ahora tiene el valor planoY independientemente
   -- de la tácnica de implementación
```

```
   Cuadricula := planoY; -- el nombre 'planoY' no resulta ambiguo
                         -- ya que Cuadricula es del tipo 'mapa'
   Cuadricula := Linea;  -- ilegal - conflicto de tipos
end;
```

Otra posibilidad que soporta Ada es el uso de subrangos o subtipos para restringir los valores de un objeto (de un tipo base concreto). Esto permite una asociación más estrecha entre los objetos del programa y los valores que pueden ser tomados por esos objetos en el dominio de la aplicación.

```
-- Ada
subtype Superficie is Dimension range planoX .. planoY;
```

Nótese que Ada tiene subtipos predefinidos para enteros naturales y positivos.

Merece la pena destacar que en Ada y en C todos los tipos pueden duplicarse, definiéndose un tipo como una nueva versión de un tipo previo:

```
-- en Ada
type Nuevo_Entero is new Integer;
type Proyeccion is new Dimension range planoX .. planoY;

/* en C */
typedef int nuevo_entero;
typedef dimension proyeccion;
```

Mientras que en Ada los objetos de un tipo y sus subtipos pueden mezclarse (en las expresiones), esto no puede hacerse con los objetos de un tipo y de un tipo derivado: los dos tipos son distintos:

```
-- Ada
D : Dimension;
S : Superficie;
P : Proyeccion;
begin
   D := S; -- legal
   S := D; -- legal pero podría causar un error de ejecución si
           -- D contiene el valor "planoZ"
   P := D; -- ilegal - conflicto de tipos
   P := Proyeccion(D); -- legal, conversión explícita de tipo
end;
```

Esta condición (y su uso) incrementa significativamente la seguridad de los programas Ada. En C, `typedef` no proporciona este nivel de seguridad.

En Java, los tipos nuevos se crean utilizando las posibilidades de la programación orientada al objeto, que serán tratadas en la Sección 4.4.

3.4.2 Números reales

Muchas aplicaciones de tiempo real (por ejemplo, el procesamiento de señal, la simulación y el control de procesos) requieren capacidades de cálculo numérico que van más allá de la aritmética entera. Existe la necesidad general de ser capaz de manipular números *reales*, aunque la sofisticación de la aritmética necesaria varía mucho entre aplicaciones. En esencia, existen dos modos distintos de representar valores reales en un lenguaje de alto nivel:

(1) Coma flotante.

(2) Enteros escalados.

Los números en coma flotante son una aproximación finita a los números reales aplicable en cálculos en los que no se necesite una precisión especial en el resultado. Un número en coma flotante está representado por tres valores: una mantisa, M, un exponente, E, y una base, B. Su valor se calcula según la fórmula $M \times B^E$. La base está (implícitamente) definida en la implementación, y normalmente tiene el valor 2. Como la mantisa está limitada en longitud, la representación tiene limitada la precisión. La divergencia entre el número en coma flotante y su correspondiente valor «real» está en proporción con el tamaño del número (por lo que se dice que es un **error relativo**).

El uso de enteros escalados está indicado para los cálculos numéricos exactos. Con una elección apropiada de la escala, se puede conseguir cualquier valor. Los enteros escalados ofrecen una alternativa a los números en coma flotante cuando se necesitan operaciones no enteras. Sin embargo, la escala debe ser conocida en tiempo de compilación; si la escala de un valor no está disponible hasta la ejecución, debe utilizarse una representación de coma flotante. Aunque los enteros escalados proporcionan valores exactos, no se pueden representar exactamente todos los números en el dominio matemático. Por ejemplo, el valor 1/3 no puede ser visto como un entero decimal escalado. La diferencia entre un entero escalado y su valor «real» es un **error absoluto**.

Los enteros escalados pueden tener la ventaja (sobre los de coma flotante) de representar valores numéricos exactos y de utilizar aritmética entera. Las operaciones de coma flotante necesitan un hardware especial (una unidad de coma flotante) o un software complejo, lo que, al final, hace que las operaciones en coma flotante sean varias veces más lentas que las equivalentes operaciones con enteros. Los enteros escalados son, sin embargo, más difíciles de utilizar, especialmente si las expresiones que tienen que ser evaluadas contienen valores con diferentes escalas.

Tradicionalmente, los lenguajes han soportado un único tipo de coma flotante (conocido usualmente como **real**), el cual tiene una precisión que depende de la implementación. El uso de enteros escalados se confia al usuario (esto es, el programador ha tenido que implementar la aritmética de enteros escalados utilizando los tipos enteros definidos en el sistema).

Ada y C utilizan el término `float` para su implementación dependiente del tipo «real». No existe un equivalente en occam2, ya que se debe especificar el número de bits. Los diseñadores de occam2 vieron que la necesidad de un tipo de dato `real` abstracto no es tan grande como la necesidad del programador de no tener que preocuparse por la precisión de las operaciones que se están llevando a cabo. Los «reales» de occam2 son `REAL16`, `REAL32` y `REAL64`. C aporta los tipos `float` y `double` para soportar respectivamente los valores en coma flotante IEEE 754 de 32 y 64 bits. Hay que destacar que en Java los literales en coma flotante se consideran automáticamente como de doble precisión. Consecuentemente, deben convertirse explícitamente a **float** o deben finalizar con la letra `f`. Por ejemplo:

```
float temperaturaCorporal = (float) 36.6;
// o
float temperaturaCorporal = 36.6f;
```

Además del tipo predefinido `Float`, Ada proporciona a los usuarios la posibilidad de crear números de coma flotante de diferente precisión, así como números de coma fija. Los números de coma fija se implementan como enteros escalados. Los siguientes ejemplos ilustran la definición de tipos. Para definir un tipo de coma flotante se necesita un límite inferior y otro superior, y una declaración de la precisión requerida (en decimal):

```
type Nuevo_Float is digits 10 range -1.0E18..1.0E18
```

Un subtipo de éste puede restringir el rango o la precisión:

```
subtype Otro_Float is Nuevo_Float digits 2;
subtype Otro_Nuevo_Float is Nuevo_Float range 0.0..1000.0;
```

La declaración de la precisión define el mínimo exigido, de forma que una implementación concreta puede conseguir una precisión mayor. Si no puede lograrse la precisión mínima, se genera un error en tiempo de compilación.

Si no se fija la precisión, el lenguaje depende de la implementación para elegir un rango seguro.

Los números de coma fija de Ada salvan al usuario de tener que implementar los operadores de enteros escalados, ya que están predefinidos. Para construir un tipo de coma fija se necesita un rango y un límite de error absoluto, llamado **delta**. Por ejemplo, la siguiente definición de tipo fija un delta de 0,05, o 1/20:

```
type Entero_Escalado is delta 0.05 range -100.00..100.00
```

Para representar todos estos valores decimales (–100,00, –99,95, –99,90 ... 99,95, 100,00), se necesita un número de bits concreto, que puede ser calculado fácilmente. La potencia de 2 más cercana (menor) a 1/20 es 1/32, que es 2^{-5}. Estos 5 bits son necesarios para proporcionar la parte fraccionaria. El rango –100,00..100,00 está contenido dentro del rango –128..128, el cual necesita 8 bits (incluido el bit de signo). En total, por lo tanto, se necesitan 13 bits:

```
sbbbbbbb.fffff
```

donde s es el bit de signo, b representa un bit entero, y f denota un bit de fracción. Resulta evidente que este tipo de coma flotante puede ser implementado en una arquitectura de 16 bits.

Nuevamente, hay que destacar que aunque un tipo de coma fija representa, exactamente, un rango de fracciones binarias, no todas las constantes decimales dentro del rango tienen una representación exacta. Por ejemplo, el número decimal 5,1 será almacenado como 00000101.00011 (en binario) o como 5,09375 (en decimal) en el tipo de coma fija definido previamente.

3.4.3 Tipos de datos estructurados

Las posibilidades de manejo de tipos de datos estructurados para cada uno de los cuatro lenguajes pueden establecerse bastante fácilmente. Occam2 y Java soportan arrays, y Ada y C soportan arrays y registros. Primero, un ejemplo con arrays:

```
-- occam2
INT Max IS 10: -- definición de una constante en occam2
[Max]REAL32 Lectura: -- Lectura es un array con diez
                     -- elementos Lectura[0] .. Lectura[9]
[Max][Max]BOOL Indicadores: -- array de 2 dimensiones
```

Todos los arrays en occam2 comienzan en el elemento cero.

```
/* C */
#define MAX 10 /* define MAX como 10 */
typedef float lectura_t[MAX]; /* el índice es 0 .. MAX-1 */
typedef short int indicadores_t[MAX][MAX]; /* no existen booleanos en C */

lectura_t lectura;
banderas_t indicadores;

// Java
static final int max = 10; // una constante

float lectura[] = new float[max]; // el índice es 0 .. max-1
boolean indicadores[] [] = new boolean[max][max];

-- Ada
Max: Const Integer:= 10;
type Lectura_T is array(0 .. Max-1) of Float;
Talla: Const Integer:= Max - 1;
type Indicadores_T is array(0 .. Talla, 0 .. Talla) of Boolean;
Lectura: Lectura_T;
Indicadores: Indicadores_T;
```

Nótese que Ada utiliza paréntesis para los arrays, mientras que occam2, Java y C utilizan los más convencionales corchetes. También, los arrays Ada pueden tener cualquier índice de comienzo, mientras que en C, Java y occam2 éste es siempre 0. En Java los arrays son representados por objetos. Para crear un objeto en Java se necesita utilizar una asignación (véase la Sección 3.4.4).

Aunque no existe una razón especial por la que no hayan sido considerados los registros en occam2, su ausencia muestra indicios de cuáles han sido las prioridades de los diseñadores e implementadores del lenguaje. Los tipos de registro en Ada son bastante inmediatos:

```
-- Ada
type Dia_T is new Integer range 1 .. 31;
type Mes_T is new Integer range 1 .. 12;
type Año_T is new Integer range 1900 .. 2050;
type Fecha_T is
   record
      Dia: Dia_T:= 1;
      Mes: Mes_T:= 1;
      Año: Año_T;
   end record;
```

Sin embargo, el modo en que C utiliza su `struct` es confuso, ya que introduce un tipo sin utilizar un `typedef` para darle un nombre (aunque se puede utilizar `typedef` si así se considera oportuno).

```
/* C */
typedef short int dia_t;
typedef short int mes_t;
typedef int año_t;
struct fecha_t {
   dia_t dia;
   mes_t mes;
   año_t año; };
/* como fecha_t no se ha presentado con un typedef, */
/* su nombre es 'struct fecha_t' */

typedef struct {
   dia_t dia;
   mes_t mes;
   año_t año; } fecha2_t ;
/* aquí puede utilizarse el nombre del tipo 'fecha2_t' */
```

En el ejemplo en C, los campos son enteros derivados, mientras que el código Ada anterior tiene tipos nuevos restringidos para los componentes. El ejemplo de Ada también muestra cómo se pueden dar valores iniciales a algunos (aunque no necesariamente a todos) los campos de un re-

gistro. Ambos lenguajes utilizan la notación «punto» para indicar los componentes individuales y para permitir la asignación de registros. Ada también soporta una asignación de registro completo utilizando agregados de registro (también están disponibles los agregados de array):

```
-- Ada
F: Fecha_t ;
begin
   F.Año:= 1989; -- notación punto
   -- F tiene ahora el valor 1-1-1989 debido
   -- a la inicialización
   F:= (3, 1, 1953); -- asignación completa
   F:= (Año => 1974, Dia => 4, Mes => 7);
      -- asignación completa utilizando la notación de nombres
   ...
end;
```

en cambio, C sólo permite una inicialización completa de registros en el caso de datos estáticos:

```
/* C */
struct fecha_t F = {1,1,1};
```

La utilización de la notación de nombres en Ada mejora la legibilidad y elimina los errores que pudieran deberse a fallos en las posiciones; por ejemplo, escribiendo (1, 3, 1953) en lugar de (3, 1, 1953).

Ada también permite extender los tipos de registros con campos nuevos. Esto facilita la programación orientada al objeto, de la que trataremos en la Sección 4.4.

Java no tiene la estructura de registro como tal. Sin embargo, se puede conseguir el mismo efecto mediante la utilización de clases. Los detalles de las clases se mostrarán en la Sección 4.4, pero mientras tanto se puede analizar la clase Java para las fechas:

```
// Java
class Fecha
{
   int día, mes, año;
}
Fecha cumpleaños = new Fecha();

cumpleaños.día = 31;
cumpleaños.mes = 1;
cumpleaños.año = 2000;
```

Como sucede en el caso de C, no es posible expresar restricciones de rango en los valores de los componentes de la fecha. Aunque, puesto que un «registro» es un objeto, se necesita utilizar

un asignador para crear una instancia de `Fecha`. La inicialización de los objetos se puede realizar mediante los constructores (véase la Sección 4.4.2).

3.4.4 Tipos de datos dinámicos y apuntadores

Existen muchas situaciones en programación, en las cuales no se puede predecir antes de la ejecución del programa el tamaño exacto de un conjunto de objetos. Aunque tanto Ada como C proporcionan arrays de longitud variable, sólo se puede conseguir una estructura de datos dinámica flexible utilizando referencias, en lugar de nombres directos. Éste es el caso de los arrays y «registros» Java.

La implementación de tipos de datos dinámicos representa una considerable sobrecarga en el tiempo de ejecución para un lenguaje. Por esta razón, occam2 no tiene estructuras dinámicas. C permite el uso de apuntadores a cualquier objeto. A continuación, se muestra un ejemplo de lista enlazada.

```
...
{
   typedef struct nodo {
      int valor;
      struct nodo *sig; /* un apuntador a la propia estructura */
   } nodo_t;

   int V;
   nodo_t *Apunt;

   Apunt = malloc(sizeof(nodo_t));
   Apunt->valor = V; /* -> acceso a lo apuntado */
   Apunt->sig = 0;
   ...
}
```

El procedimiento `malloc` de la biblioteca estándar es capaz de asignar memoria de forma dinámica. El operador `sizeof` devuelve el numero de unidades de almacenamiento que el compilador dedica al tipo `nodo_t`.

En Ada, se utiliza un tipo acceso en lugar de un apuntador (aunque el concepto es similar).

```
type Nodo; -- declaración incompleta
type Acc is access Nodo;
type Nodo is
   record
      Valor: Integer;
      Sig: Acc;
   end record;
```

```
V: Integer;
A1: Acc;
begin
   A1:= new(Nodo); -- construcción del primer nodo
   A1.Valor:= V; -- se referencia a la variable de acceso
                 -- y se identifica el componente
   A1.Sig:= null; -- predefinido
   ...
end;
```

El fragmento de programa anterior ilustra el uso de la herramienta «new» para asignar dinámicamente un área de memoria (del montón). Al contrario que en C, el operador «new» de Ada está definido en el lenguaje, pero, sin embargo, no existe un operador para desalojar. En su lugar, se proporciona un operador genérico que elimina el almacenamiento de los objetos designados. Este procedimiento, llamado liberación sin comprobación (`Unchecked_Deallocation`), no realiza comprobación alguna para ver si existe alguna otra referencia al objeto pendiente.

Ni Ada ni C especifican la necesidad de un recolector de basura. No debe sorprender esa omisión, ya que los recolectores de basura normalmente producen sobrecargas impredecibles en tiempo de ejecución. Estas sobrecargas pueden llegar a ser inaceptables en los sistemas de tiempo real (véase la Sección 15.9.1).

Los apuntadores también pueden apuntar a objetos estáticos y objetos de la pila. El siguiente ejemplo nos muestra cómo C permite apuntadores a tipos estáticos, así como la aritmética de apuntadores:

```
{
   typedef fecha_t eventos_t[MAX], *sgte_evento_t;

   eventos_t historia;
   sgte_evento_t sgte_evento;

   sgte_evento = &historia[0];
   /* toma la dirección del primer elemento del array */

   sgte_evento++;
   /* incrementa el apuntador sgte_evento; */
   /* el incremento añade el tamaño del registro */
   /* fecha al apuntador y por lo tanto sgte_evento */
   /* apunta al siguiente elemento del array */
}
```

La desventaja del enfoque de C es que puede que los apuntadores acaben apuntando fuera del rango apropiado. A esto se le denomina problema de los **apuntadores colgados**. Ada proporcio-

na una solución segura a estos apuntadores con sus tipos alias. Sólo los tipos alias pueden ser referencias:

```
Objeto : aliased Un_Tipo;
   -- aliased para decir que puede ser
   -- referenciado por un tipo de acceso

type Apunt_General is access all Un_Tipo;
   -- access all indica que una variable acceso de este tipo
   -- puede apuntar tanto a objetos estáticos como dinámicos

Ag : Apunt_General := Objecto'Access;
   -- asigna la referencia de Objeto a Ag
```

Una última forma de definición de tipo de acceso en Ada permite imponer la restricción de sólo lectura en los accesos:

```
Objeto1 : aliased Un_Tipo;
Objeto2 : aliased constant Un_Tipo := ...;

type Apunt_General is access constant Un_Tipo;

Ag1 : Apunt_General := Objeto1'Access;
   -- Ag1 ahora sólo puede leer el valor de Objeto1

Ag2 : General_Ptr := Objeto2'Access;
   -- Ag2 es una referencia a una constante
```

Al contrario que en Ada y C, todos los objetos en Java son referencias al objeto real que contiene los datos, por lo que no existe un tipo acceso o apuntador. Además, tampoco se necesita la declaración del tipo Nodo, que en Ada sí era necesaria.

```
// Java
{
   class Nodo
   {
      int valor;
      Nodo sgte;
   }
   Nodo Ref = new Nodo();
}
```

Como resultado de representar los objetos como referencias, la comparación de dos objetos en Java es la comparación de sus apuntadores, y *no* la de sus valores. Por lo tanto:

```
Nodo Ref1 = new Nodo();
Nodo Ref2 = new Nodo();
...
if(Ref1 == Ref2) { ... }
```

comparará la ubicación de los objetos, no los valores encapsulados en esos objetos (en este caso, los valores de los campos `valor` y `sgte`). Para comparar los valores se necesita que la clase implemente explícitamente el método `compareTo`. Por supuesto, lo mismo es cierto para las variables acceso de Ada. Sin embargo, Ada permite utilizar un sufijo **.all** en las variables acceso para indicar que se trata de los datos del objeto, y no de su referencia.

Una situación similar se da en la asignación de objetos. En Java, el operador asignación asigna el apuntador. Para crear una copia del objeto en cuestión se requiere que la clase proporcione los métodos para clonar (`clone`) el objeto.

3.4.5 Archivos

Ni Ada, ni Java, ni C ni occam2 tienen un constructor de tipo archivo como Pascal. En su lugar, cada lenguaje permite una implementación que soporte archivos vía bibliotecas. Ada, por ejemplo, necesita que todos los compiladores proporcionen archivos de acceso, tanto secuencial como directo.

3.5 Estructuras de control

Aunque existe alguna variación en las estructuras de datos aportadas (particularmente entre occam2 y los demás lenguajes), existe un acuerdo mucho más general sobre las estructuras de control requeridas. Así como los lenguajes de programación progresaron desde el código máquina, a través de los lenguajes ensambladores, hasta los lenguajes de alto nivel, las instrucciones de control evolucionaron hacia sentencias de control. Existe un acuerdo general sobre la abstracción de control necesaria en un lenguaje de programación secuencial. Estas abstracciones se pueden agrupar en tres categorías: secuencias, decisiones y bucles. Vamos a tratar cada una de ellas por separado. Las estructuras de control necesarias en las partes concurrentes de los lenguajes serán tratadas en los Capítulos 7, 8 y 9.

3.5.1 Secuencia

La ejecución secuencial de sentencias es el modo normal de comportamiento en un lenguaje de programación (no concurrente). La mayoría de los lenguajes (incluidos Ada, Java y C) consideran la ejecución secuencial de forma implícita, y no necesitan de una estructura de control específica para ello. Tanto en Ada como en Java y en C, las definiciones de bloques indican que una secuencia de sentencias se encuentra entre un «begin» ({) y un «end» (}), y se exige que la ejecución siga dicha secuencia.

En occam2, la ejecución normal de las sentencias (llamadas procesos en el lenguaje) podría ser concurrente, por lo que resulta necesario establecer de forma explícita el que un conjunto de acciones deba seguir cierta secuencia. Esto se consigue mediante el constructor SEQ ya mencionado; por ejemplo:

```
SEQ
  acción 1
  acción 2
  .
  .
  .
```

Si por alguna circunstancia particular la secuencia estuviera vacía, Ada exige la utilización explícita de una sentencia null:

```
begin -- Ada
   null;
end;
```

Java y C permiten la existencia de bloques vacíos:

```
{ /* Java/C */
}
```

y occam2 utiliza un proceso `SKIP` para dar a entender la no acción:

```
SEQ
  SKIP -- occam2
```

o simplemente

```
SKIP
```

En el otro extremo de la acción nula, está aquélla que causa la detención de la secuencia. Java y C proporcionan un procedimiento predefinido llamado `exit`, que provoca la finalización del programa entero. El proceso `STOP` de occam2 tiene un efecto similar. Ada no tiene una primitiva equivalente, pero se pueden programar las excepciones para que tengan el mismo resultado (véase el Capítulo 6), o se pueda abortar el programa principal. En los cuatro lenguajes, la acción drástica de terminar prematuramente un programa sólo se utiliza como respuesta a la detección de una condición de error que no se puede reparar. Después de llevar al sistema controlado a una condición segura, el programa no puede efectuar ninguna otra operación que la finalización del mismo.

3.5.2 Estructuras de decisión

Una estructura de decisión proporciona la posibilidad de elegir el camino que sigue la ejecución a partir de un punto dado en la secuencia de un programa. La decisión de la ruta a seguir depen-

de de los valores actuales de ciertos objetos relevantes. La propiedad más importante de una estructura de control de decisión es que todas las rutas se agrupan al final. Con estas estructuras de control abstractas no existe la necesidad de utilizar la sentencia de control `goto`, la cuál a menudo conduce a programas difíciles de comprobar, leer y mantener. Java y occam2 no poseen la sentencia `goto`. Ada y C sí la tienen, pero debería ser utilizada con mesura.

La forma de estructura de decisión más común es la declaración `if`. Aunque los requisitos de una estructura de este tipo están claros, los lenguajes anteriores, como Algol-60, adolecían de cierta pobreza sintáctica que podía producir confusión. En concreto, en las construcciones `if` anidadas no estaba claro a cuál de los `if` se asociaba un `else` concreto. Desafortunadamente, C todavía sufre este problema, mientras que Ada y occam2 tienen unas estructuras totalmente inequívocas. Para ilustrar este aspecto, considérese un problema simple: hallar si $B/A > 10$ (¡comprobando antes que A es distinto de cero!). A continuación se da una solución Ada, aunque el código tiene el inconveniente de que no se asignará valor alguno a la variable `Mayor` si $A = 0$, el significado del fragmento resulta claro debido a la utilización del elemento **`end if`**:

```
if A /= 0 then
   if B/A > 10 then
      Mayor:= True;
   else
      Mayor:= False;
   end if;
end if;
```

Ada también proporciona dos formas de acortar el `if`, **`and then`** y **`or else`**, que permiten una expresión más concisa del código anterior.

La estructura correspondiente en C no necesita un fin explícito:

```
if(A != 0)
   if(B/A > 10) mayor = 1;
   else mayor = 0;
```

Sin embargo, la estructura puede quedar mucho más clara si se agrupan las sentencias entre llaves:

```
if(A != 0) {
   if(B/A > 10) {
      mayor = 1;
   }
   else {
      mayor = 0;
   }
}
```

Java sigue la estructura de C; sin embargo, se evita el problema de la asociación de los `else` al prohibir las sentencias ambiguas.

La estructura IF de occam2 tiene un estilo algo diferente con respecto a los otros tres lenguajes, aunque la funcionalidad es la misma. Consideremos primero la estructura general. En el siguiente esquema, B1 . . Bn son expresiones booleanas, y A1 . . An acciones:

```
IF
  B1
    A1
  B2
    A2
  .
  .
  Bn
    An
```

Primeramente, es importante recordar que la disposición física de los programas occam2 es sintácticamente significativa. Las expresiones booleanas están en líneas separadas (identadas dos espacios desde IF), del mismo modo que las acciones (identadas otros dos espacios más). Como en C, no se utiliza un elemento «then». En la ejecución de este IF, se evalúa la expresión booleana B1. Si resulta a TRUE, entonces se ejecuta la acción A1, y esto es todo lo que hace el IF. Sin embargo, si B1 es FALSE, se evalúa B2. Las expresiones booleanas se evaluan una tras otra hasta que se encuentra una que sea TRUE, en cuyo caso se toma la acción correspondiente. Si ninguna de las expresiones booleanas es TRUE, entonces se da una condición de error, y la construcción IF se comporta como el STOP visto en la sección anterior.

Con esta forma de declaración IF, no es necesaria una parte ELSE: la expresión booleana TRUE utilizada como último test se tendrá en cuenta si todas las demás condiciones han fallado. El ejemplo b/a > 10 en occam2, por lo tanto, toma la siguiente forma:

```
IF
  a /= 0
    IF
      b/a > 10
        mayor := TRUE
      TRUE
        mayor := FALSE
  TRUE      -- Estas dos líneas son necesarias
    SKIP    -- para que el IF no se convierta
            -- en STOP si a vale 0
```

Para mostrar otro ejemplo del constructor «if», consideremos un caso de elección múltiple. En el siguiente ejemplo (primeramente en Ada), se calcula el número de dígitos en una variable entera positiva Numero. Se supone que el máximo de dígitos es 5:

```
if Numero < 10 then
  Num_Digitos := 1;
```

```
else
   if Numero < 100 then
      Num_Digitos := 2;
   else
      if Numero < 1000 then
         Num_Digitos := 3;
      else
         if Numero < 10000 then
            Num_Digitos := 4;
         else
            Num_Digitos := 5;
         end if;
      end if;
   end if;
end if;
```

Esta forma, bastante común, implica la anidación de la parte `else`, y produce una cola de `end if`s al final. Para eliminar esta estructura poco elegante, puede reescribirse lo anterior de una manera más concisa como:

```
-- Ada
if Numero < 10 then
   Num_Digitos := 1;
elsif Numero < 100 then
   Num_Digitos := 2;
elsif Numero < 1000 then
   Num_Digitos := 3;
elsif Numero < 1000 then
   Num_Digitos := 4;
else
   Num_Digitos := 5;
end if;
```

En occam2, el código es bastante claro:

```
IF
  numero < 10
    digitos := 1
  numero < 100
    digitos := 2
  numero < 1000
    digitos := 3
```

```
  numero < 10000
    digitos := 4
  TRUE
    digitos := 5
```

Lo anterior es un caso de selección múltiple construida a partir de selecciones binarias. En general, puede establecerse una decisión múltiple de una forma más eficiente utilizando una estructura `case` (o `switch`). Los cuatro lenguajes poseen una estructura de este tipo, aunque la versión occam2 resulte algo más restringida. Para ver esto, considérese un carácter (un byte) almacenado en «orden» que se utiliza para decidir entre cuatro acciones posibles:

```
-- Ada
case Orden is
   when 'A' | 'a'    => Accion1;    -- A o a
   when 't'          => Accion2;
   when 'e'          => Accion3;
   when 'x' .. 'z'   => Accion4;    -- x, y o z
   when others       => null;       -- sin acción
end case;
```

```
/* C y Java */
switch(orden) {
   case 'A'   :
   case 'a'   : accion1; break;     /* A o a */
   case 't'   : accion2; break;
   case 'e'   : accion3; break;
   case 'x'   :
   case 'y'   :
   case 'z'   : accion4; break;     /* x, y o z */
   default    : break;              /* sin acción */
}
```

Nótese que en Java y C es necesario insertar sentencias para romper la secuencia una vez que ha sido identificada la orden. Sin ellas, el control continua a la siguiente opción (como en el caso de «A»).

En occam2 no están soportados los valores alternativos ni los rangos, por lo que el código es más extenso:

```
CASE orden
   'A'
     accion1
   'a'
     accion1
```

```
  't'
    accion2
  'e'
    accion3
  'x'
    accion4
  'y'
    accion4
  'z'
    accion4
  ELSE
    SKIP
```

3.5.3 Estructuras repetitivas

Una estructura repetitiva permite al programador especificar que una sentencia, o un conjunto de ellas, sea ejecutada más de una vez. Existen dos formas distintas de construir dichos bucles:

(1) Iteración

(2) Recursión

La característica distintiva de la *iteración* es que se completa cada ejecución del bucle antes de comenzar la siguiente. Con una estructura de control recursiva, se detiene el primer ciclo para comenzar el segundo, el cual puede ser detenido a su vez para comenzar el tercero, y así sucesivamente. En algún punto, el ciclo n será capaz de completarse, momento en el cual podrá acabar el ciclo $n - 1$, y también el $n - 2$, y así hasta que el primer ciclo también acabe. La recursión se implementa normalmente mediante llamadas a procedimientos recursivos. Aquí, se tratará la iteración.

La iteración se presenta bajo dos formas: un bucle en el cual el número de iteraciones es fijado previamente a la ejecución del ciclo, y un bucle en el que en cada iteración se comprueba si se cumple cierta condición. La primera forma se conoce como declaración `for`, mientras que la segunda es conocida como declaración `while`. La mayoría de las construcciones `for` de los lenguajes también proporcionan un contador que puede ser utilizado para indicar la iteración actualmente en ejecución.

El siguiente ejemplo ilustra la construcción `for` en nuestros cuatro lenguajes; el código asigna a los primeros diez elementos del array `A` el valor de su posición dentro del array:

```
-- Ada
for I in 0 ..9 loop   -- I es definido por el bucle
  A(I):= I;           -- I es de sólo lectura en el bucle
end loop;             -- I está fuera de alcance después del bucle
```

```
/* C y Java */
for(i = 0; i <= 9; i++) {    /* i debe ser definido previamente */
  A[i]= i;                   /* i puede ser leído/escrito en el bucle */
  }                          /* el valor de i está definido */
                             /* después del bucle */
```

```
-- occam2
SEQ i = 0 FOR 10    -- i es definido por el constructor
  A[i]:= i          -- i es de sólo lectura en el bucle
                    -- i está fuera de alcance en el exterior del bucle
-- el rango de i es de 0 a 9,
-- como en los ejemplos de Ada y C
```

Hay que destacar que Ada y occam2 tienen restringido el acceso a la variable del bucle. El uso sin limitaciones de la misma en Java y en C puede ser fuente de errores. Por esta razón, Java también permite declarar una variable local dentro del bucle `for`.

```
// Java
for(int i = 0; i <= max; i++) {
   A[i]= i;
}
```

Además de las formas anteriores, Ada permite ejecutar el bucle en orden inverso.

La principal variación con respecto a las declaraciones `while` concierne al punto en el cual se realiza la comprobación para salirse del bucle. La forma más común realiza la comprobación a la entrada del bucle, y por lo tanto antes de cualquier iteración:

```
-- Ada
while <Expresión Booleana> loop
   <Sentencias>
end loop;
```

```
/* Java y C */
while(<expresión>) {
   /* una expresión evaluada a 0 implica la terminación del bucle */
   <sentencia>
}
```

```
-- occam2
WHILE <expresión booleana>
  SEQ
    <sentencias>
```

Java también soporta una variante donde la comprobación se da al final del bucle:

```
do {
   < secuencia de sentencias>
} while (<expresión>);
```

Ada, Java y C incrementan la flexibilidad de la construcción, permitiendo que el control pase al exterior del bucle (esto es, que el bucle finalice) desde cualquier punto de dentro del mismo:

```
-- Ada
loop
   .
   .
   exit when <Expresión Booleana>;
   .
   .
end loop;

   /* C y Java */
while(1) {
   .
   .
   if(<expresión>) break;
   .
   .
}
```

Un error común de programación es el caso de un bucle que o bien no termina (cuando tendría que haber terminado) o termina en un estado erróneo. Afortunadamente, ahora se comprenden bien los métodos formales de análisis de estructuras cíclicas. Éstos involucran la definición de precondiciones y postcondiciones para los bucles, la condición de terminación, y el invariante del bucle. El invariante del bucle es una sentencia que es cierta al final de cada iteración, pero que puede no serlo durante la misma. En esencia, el análisis del bucle involucra la demostración de que la precondición va a conducir a la terminación del bucle, y que a la terminación, el invariante del bucle conduce a demostrar que se satisface la postcondición. Para facilitar la utilización de estas aproximaciones formales, se desaconseja utilizar la salida del bucle durante la iteración. Siempre que sea posible, la construcción `while` estándar es la mejor elección.

La última puntualización sobre los bucles es que los sistemas de tiempo real necesitan a menudo un bucle sin fin. De un ciclo de control se esperará que esté ejecutándose indefinidamente (esto es, hasta que se apague la máquina). Aunque el bucle **`while`** `True` facilita un bucle de estas características, puede ser ineficaz, y no expresa la esencia de un bucle infinito. Por esta razón, Ada proporciona una estructura de bucle simple:

```
-- Ada
loop
   <Sentencias>
end loop;
```

3.6 Subprogramas

Incluso en la construcción de un componente o un módulo, resulta deseable una descomposición más completa. Esto se puede conseguir mediante procedimientos y funciones, conocidos colectivamente como *subprogramas*.

Los subprogramas no sólo ayudan en la descomposición, sino que también representan una forma importante de abstracción. Permiten definir cálculos complejos y arbitrarios y después invocarlos mediante un simple identificador. Esto posibilita que tales componentes sean reutilizados tanto dentro de un programa como entre programas. La generalidad, y por lo tanto la utilidad, de los subprogramas se incrementa, por supuesto, mediante la inclusión de parámetros.

3.6.1 Modos y mecanismos para el paso de parámetros

Un parámetro es una forma de comunicación: es un objeto de datos que está siendo transferido entre el usuario de un subprograma y el subprograma mismo. Existen varios modos de describir el mecanismo utilizado para esta transferencia de datos. Primeramente, se puede considerar el modo en que los parámetros son transferidos. Desde el punto de vista del invocador, existen tres modos distintos de transferencia.

(1) Los datos se pasan hacia dentro del subprograma.

(2) Los datos se pasan hacia fuera del subprograma.

(3) Los datos se pasan hacia dentro del subprograma, se cambian, y luego se pasan hacia fuera del subprograma.

Estos tres modelos se denominan: **dentro, fuera** y **dentro fuera** (*in*, *out* e *in out*).

La segunda forma de describir la transferencia es considerar el enlace entre el parámetro formal del subprograma y el parámetro real de la llamada. Existen dos métodos generales de hacer esto. Un parámetro puede estar vinculado por valor o por referencia. Un parámetro vinculado por valor sólo contiene el valor del parámetro comunicado al programa (a menudo mediante una copia dentro del espacio de memoria del subprograma), de forma que no se puede devolver ninguna información al invocador vía este parámetro. Cuando un parámetro se vincula por referencia, cualquier actualización de ese parámetro dentro del subprograma va a tener efecto en la ubicación de memoria del parámetro real.

El último modo de considerar el mecanismo de paso de parámetros es examinar los métodos utilizados por la implementación. El compilador debe satisfacer la semántica del lenguaje, ya esté ésta expresada en términos de modo o de enlace, pero a la vez es libre de implementar una llamada a un subprograma de la forma más eficiente posible. Por ejemplo, un parámetro con un array muy grande pasado por valor no tiene que ser copiado si no se realizan asignaciones a sus elementos dentro del subprograma. Un simple apuntador al array real será más eficiente, y es equivalente respecto a su comportamiento. De forma similar, una llamada por referencia a un parámetro puede implementarse mediante un algoritmo de copia hacia dentro y copia hacia fuera.

Ada utiliza modos de parámetros para expresar el significado de la transferencia de datos hacia y desde un subprograma. Por ejemplo, considérese un procedimiento que devuelve las raíces reales (si existen) de una ecuación cuadrática.

```
procedure Cuadratica (A, B, C  : in Float;
                      R1, R2   : out Float;
                      Ok       : out Boolean);
```

Un parámetro `in` (lo que sucede por defecto) actúa como una constante local dentro del subprograma (se asigna un valor al parámetro formal en la entrada al procedimiento o función). Éste es el único modo permitido en las funciones. Dentro de los procedimientos, un parámetro `out` puede ser escrito y leído. Se asigna un valor al parámetro invocador a la terminación del procedimiento. Un parámetro «in out» actúa como una variable dentro del procedimiento. A la entrada se le asigna el valor del parámetro formal, y a la salida del procedimiento el valor que almacena se le pasa al parámetro invocador (real).

C pasa los parámetros (o **argumentos**, como los llama) por valor. Si se tienen que devolver resultados, se deben utilizar apuntadores. C, como Java, sólo soporta funciones; un procedimiento es considerado como una función sin valor de retorno (indicado por un `void` en la definición de la función).

```
void cuadratica(float A, float B, float C,
                float *R1, float *R2, int *OK);
```

Hay que destacar que no existe la palabra clave `function`. También, aun cuando `A`, `B` y `C` han sido copiadas en la función, pueden ser escritas dentro de la misma. Sin embargo, ninguno de sus nuevos valores serán copiados de regreso.

En Java, los argumentos primitivos son pasados mediante copia. Las variables de clase son variables referencia. Por lo tanto, cuando son pasadas como argumentos son copiadas, pero el efecto conseguido es como si hubieran sido pasadas por referencia. Si un argumento no debe ser modificado dentro de una función, debe declararse como `final` en la especificación de la función.

```
public class Raices {
   float R1, R2;
}
```

```
boolean cuadratica (final float A, final float B, final float C,
                    Raices R);
```

Nótese que en el ejemplo de Java el indicador booleano se devuelve mediante el valor retornado por la función. Java también necesita que las raíces de la ecuación sean pasadas como de tipo clase, ya que los tipos primitivos (incluyendo `float`) son pasados por copia, y no existen tipos apuntadores.

En occam2, los parámetros, por defecto, se pasan por referencia, y se utiliza una etiqueta VAL para expresar que el paso es por valor. Dentro de un procedimiento (llamado un PROC en occam2), un parámetro VAL actúa como una constante, y, por lo tanto, los errores que en Pascal podrían darse al omitirse la etiqueta VAR son capturados por el compilador.

```
PROC cuadratica (VAL REAL32 A, B, C,
                 REAL32 R1, R2, BOOL OK)
  -- como en C y Java, el separador de parámetros
  -- no es un punto y coma
```

3.6.2 Procedimientos

Los cuerpos de los procedimientos en los cuatro lenguajes son bastante sencillos, tal y como se muestra al completar las definiciones de «cuadrática» dadas antes. Todos los procedimientos suponen que la función `sqrt` (raíz cuadrada) está disponible.

```
-- Ada
procedure Cuadratica (A, B, C : in Float;
                      R1, R2 : out Float;
                      Ok : out Boolean) is
  Z : Float;
begin
  Z:= B*B - 4.0*A*C;
  if Z < 0.0 or A = 0.0 then
    Ok:= False;
    R1:= 0.0;   -- valores arbitrarios
    R2:= 0.0;
    return;     -- retorno del procedimiento antes
                -- de alcanzar el final lógico
  end if;
  Ok:= True;
  R1:= (-B + Sqrt(Z)) / (2.0*A);
  R2:= (-B - Sqrt(Z)) / (2.0*A);
end Cuadratica;
```

```
/* C */
void cuadratica (float A, float B, float C, float *R1,
                 float *R2, int *OK)
{
   float Z;

   Z= B*B - 4.0*A*C;
   if(Z < 0.0 || A == 0.0) {
      *OK = 0;
      *R1 = 0.0;   /* valores arbitrarios */
      *R2 = 0.0;
      return;      /* retorno del procedimiento antes */
                   /* de alcanzar el final lógico */
   }
   *OK = 1;
   *R1 = (-B + SQRT(Z)) / (2.0*A);
   *R2 = (-B - SQRT(Z)) / (2.0*A);
}

// Java
public class Raices {
   float R1, R2;
}
boolean cuadratica (final float A, final float B, final float C, Raices R) {
   // se requiere la conversión explícita a float
   float Z;
   Z =(float) (B*B - 4.0*A*C);
   if (Z < 0.0 || A == 0.0) {
      R.R1 = 0f; // valores arbitrarios
      R.R2 = 0f;
      return false;
   }
   R.R1 = (float) ((-B + Math.sqrt(Z)) / (2.0*A));
   R.R2 = (float) ((-B - Math.sqrt(Z)) / (2.0*A));
   return true;
};

--occam2
PROC cuadratica (VAL REAL32 A, B, C,
                 REAL32 R1, R2, BOOL OK)
   REAL32 Z:
```

```
  SEQ
    Z:= (B*B) - (4.0*(A*C))     -- parántesis necesarios para
                                -- especificar totalmente la expresión
    IF
      (Z < 0) OR (A = 0.0)
        SEQ
          OK:= FALSE
          R1:= 0.0              -- valores arbitrarios
          R2:= 0.0
      TRUE                      -- sin sentencia return en occam2
        SEQ
          OK:= TRUE
          R1:= (-B + SQRT(Z)) / (2.0*A)
          R2:= (-B - SQRT(Z)) / (2.0*A)
: -- dos puntos para indicar el final de la declaración PROC
```

La invocación de estos procedimientos se hace en los cuatro lenguajes mediante el nombre de los procedimientos acompañado por los parámetros apropiados entre paréntesis.

Además de estas características básicas, existen dos posibilidades extra disponibles en Ada que mejoran la legibilidad. Considere un tipo enumerado `Estado` y un tipo entero que se refiere a 10 válvulas distintas:

```
type Estado is (Abierto, Cerrado);
type Valvula is new Integer range 1 .. 10;
```

La siguiente especificación de procedimiento muestra un subprograma para modificar el estado de una válvula:

```
procedure Modificar_Estado (Num_Valvula : Valvula;
                            Posicion : Estado := Cerrado;
                            );
```

Hay que destacar que uno de los parámetros ha recibido un valor por defecto. La invocación de este procedimiento puede tomar varias formas:

```
Modificar_Estado(6, Abierto);  -- invocación normal
Modificar_Estado(3);           -- se utiliza el valor por defecto 'Cerrado'
Modificar_Estado(Posicion => Abierto,
                 Num_Valvula => 9);  -- notación nombre
Modificar_Estado(Num_Valvula => 4);  -- notación nombre y
                                     -- valor por defecto
```

Los valores por defecto son útiles si alguno de los parámetros recibe casi siempre el mismo valor. La utilización de la notación nombre elimina los errores posicionales e incrementa la legibilidad.

Ada, Java y C permiten llamadas recursivas (y mutuamente recursivas) de procedimientos que no son soportadas por occam2, debido a la sobrecarga dinámica que generan en tiempo de ejecución.

3.6.3 Funciones

Ada proporciona funciones de una forma similar a los procedimientos. Considérese el ejemplo simple de una función que devuelve el menor de dos valores enteros:

```
-- Ada
function Minimo (X, Y : in Integer) return Integer is
begin
   if X > Y then
      return Y;
   else
      return X;
   end if;
end Minimo;

/* C y Java */
int minimo (int X, int Y)
{
   if(X > Y) return Y;
   else return X;
}
```

Ada, Java y C permiten utilizar funciones para devolver cualquier tipo válido, incluyendo tipos estructurados.

La mala utilización de las funciones es la fuente de bastantes errores dentro de los programas; la regla de oro sobre funciones es que no deberían tener efectos laterales. Una expresión debería significar lo que dice:

```
A := B + F(C)
```

El valor de `A` se convierte en el valor de `B` más un valor obtenido de `C` al aplicarle la función `F`. En la ejecución de la expresión anterior, sólo `A` debería cambiar su valor.

Se pueden introducir efectos laterales en la expresión anterior de tres maneras.

(1) `F` podría cambiar el valor de `C` a la vez que devuelve un valor.

(2) `F` podría cambiar el valor de `B` de forma que la expresión tenga un valor diferente dependiendo de si es evaluada de izquierda a derecha o viceversa.

(3) `F` podría cambiar el valor de `D`, donde `D` es cualquier otra variable dentro de su alcance.

Ada y Java restringen los efectos laterales permitiendo que los parámetros sean solamente hacia dentro. Por supuesto, estos parámetros pueden referenciar otros objetos, por lo que los efectos laterales son realmente posibles. Occam2, sin embargo, va más allá, y define la semántica de las funciones de forma que no sean posibles los efectos laterales.

Como en Ada, los parámetros de una función occam2 son pasados por valor. Además, el cuerpo de una función debe ser definido como VALOF. Un VALOF es una secuencia de sentencias necesaria para calcular el valor de un objeto (el cual será devuelto por la función). La propiedad importante del VALOF es que las únicas variables cuyos valores pueden ser cambiados son aquéllas definidas localmente en el VALOF. Esto impide los efectos laterales. La sencilla función para calcular el mínimo, definida antes, podría tener la siguiente forma:

```
INT FUNCTION minimo (VAL INT X, Y)
  INT Z: -- Z será el valor devuelto
  VALOF
    IF
     X > Y
       Z:= Y
     TRUE
       Z:= X
    RESULT Z
:
```

En los lenguajes concurrentes, otra forma de efecto lateral es la concurrencia subyacente, que puede tener una serie de desafortunadas consecuencias. El VALOF de occam2 además es una forma de deshabilitar cualquier concurrencia dentro de él.

Finalmente, hay que indicar que en Java las funciones (y los procedimientos) sólo pueden ser declarados en el contexto de una definición de clase (véase la Sección 4.4.2).

3.6.4 Apuntadores a subprogramas

Tanto Ada como C permiten apuntadores a procedimientos y a funciones. Por ejemplo, la siguiente declaración de tipo Ada (`Informe_Error`) define una variable de acceso a un procedimiento que toma un parámetro de cadena de caracteres. Después se declara un procedimiento que tiene un parámetro de este tipo, y la llamada al procedimiento se hace pasando un apuntador al procedimiento `Aviso_Operador`.

```
type Informe_Error is access procedure (Razon: in String);

procedure Aviso_Operador(Mensaje: in String) is
begin
   -- informar al operador del error
end Aviso_Operador;
```

```
procedure Calculo_Complejo(Error_A : Informe_Error; ... ) is
begin
   -- si se detecta un error durante el cálculo complejo
   Error_A("Abandonando");
end Calculo_Complejo;
...
   Calculo_Complejo(Aviso_Operador'Access, ...);
...
```

En C, el equivalente a un apuntador al informe de error es:

```
void (*informe_error)(char *Mensaje);
```

y la dirección de una función se obtiene mediante

```
informe_error = aviso_operador;
```

Java no permite apuntadores a funciones (procedimientos), ya que sólo pueden ocurrir dentro del contexto de la clase a la cual se accede por referencia, de cualquier modo.

3.6.5 Expansión en línea

Aunque la utilización de subprogramas es claramente beneficiosa en términos de descomposición, reutilización y legibilidad, en el caso de algunas aplicaciones de tiempo real la sobrecarga de la implementación de las llamadas puede ser inaceptablemente alta. Un modo de reducir la sobrecarga es trasladar el código del subprograma al lugar en el que se encontraría la correspondiente llamada. Esta técnica se conoce como **expansión en línea**, y tiene la ventaja de que permite al programador utilizar subprogramas sin añadir sobrecarga alguna en el tiempo de ejecución.

Es interesante destacar que occam2, cuya semántica está especificada formalmente, utiliza la substitución textual cuando especifica qué se entiende por una llamada a procedimiento. Los cuatro lenguajes, sin embargo, permiten que el implementador trate a los subprogramas del modo que considere apropiado. La única excepción a esto se da en Ada cuando el programador puede solicitar, utilizando el pragma `Inline`, que se realice la expansión en línea para el subprograma implicado siempre que sea posible. En Ada estos pragmas sirven para dar instrucciones al compilador, y no son sentencias ejecutables.

Resumen

En el título de su original libro, Wirth expresó el ahora famoso adagio:

```
"Algoritmos + Estructuras de Datos = Programas"
```

Quizás, una vez vistas las dificultades aparecidas en los grandes programas, el adagio podría ser parafraseado como algoritmos + estructuras de datos = módulos, donde un módulo es un componente de un programa diseñado y desarrollado por un ingeniero de software o por un grupo pequeño de ellos.

En este capítulo se han mostrado las características de los lenguajes Ada, Java, C y occam2 necesarias para expresar los algoritmos y representar las estructuras de datos. Aunque estos lenguajes son diferentes, presentan al programador modelos semánticos similares. En efecto, en los lenguajes imperativos las primitivas que soportan la programación de lo pequeño están bien establecidas.

Para expresar los algoritmos, se necesitan bloques, bucles y estructuras de decisión. La sentencia `goto` está totalmente desacreditada. También son necesarios los subprogramas para conseguir una realización concreta de las distintas unidades lógicas que se dan en la mayoría de los módulos no triviales. La semántica de los procedimientos está relativamente libre de discusión (aunque existan diferentes modos para el paso de parámetros), pero las funciones presentan algunos problemas aún, debido a los efectos laterales. Occam2 ha tomado la delantera en el intento de solucionar este problema al considerar una forma de función que no puede tener efectos laterales (occam1 tenía una solución más radical a los efectos laterales: ¡no tenía funciones!).

Ada, Java, C y occam2 proveen una rica variedad de estructuras de datos y de reglas de tipado, aunque las posibilidades en occam2 no son tan completas, y el tipado en C no tan fuerte. El tipado fuerte está universalmente aceptado como una asistencia necesaria para la producción de código fiable. Las restricciones que impone pueden conducir a dificultades, pero pueden ser solventadas mediante la conversiones explícitas controladas. La ausencia de un tipado fuerte en C es una desventaja en cuanto al uso de este lenguaje en sistemas de alta integridad.

Los tipos de datos pueden clasificarse de varias formas. Existe una división clara entre tipos de datos escalares y estructurados. Los tipos escalares pueden ser subdivididos en tipos discretos (tipos enteros y enumerados) y tipos reales. Los tipos de datos estructurados pueden ser clasificados en función de tres parámetros: homogeneidad, tamaño y método de acceso. Un tipo de dato estructurado se dice que es homogéneo si todos sus subcomponentes son del mismo tipo (por ejemplo, los arrays). Si, por el contrario, los subcomponentes pueden ser de distintos tipos (como por ejemplo sucede con los registros), entonces se dice que la estructura de datos es heterogénea. El tamaño puede ser fijo o variable. Tanto los registros como los arrays, en algunos lenguajes, son de tamaño fijo. Las estructuras de datos dinámicas, como las listas enlazadas, los árboles o los grafos, son de tamaño variable, y normalmente son construidas por el programador a partir de un tipo apuntador (o referencia) y de un asignador de memoria. Finalmente, existen varios métodos de acceso a los subcomponentes de una estructura. Los dos más importantes son el directo y el indirecto. El acceso directo, como su propio nombre indica, permite la referencia inmediata de un subcomponente (por ejemplo, un elemento de un array o el campo de un registro). El acceso indirecto implica que el direccionamiento de un subcomponente requiere una cadena de acceso a través de otros componentes. Las estructuras más dinámicas sólo tienen accesos indirectos.

Los atributos (homogeneidad, tamaño y método de acceso) podrían, al menos teóricamente, dar lugar a ocho estructuras diferentes. En la realidad, los atributos están interrelacionados (por ejemplo, un tamaño fijo implica acceso directo), y sólo son necesarias las siguientes categorías:

- Arrays con límites y dimensiones arbitrarias.
- Registros (o clases).
- Apuntadores para construir estructuras de datos dinámicas arbitrarias con direccionamiento indirecto.

Cualquier lenguaje que proporcione las estructuras de control apropiadas y todas estas categorías (como hacen Ada, Java y C), es muy capaz de soportar la programación de lo pequeño. Las funcionalidades extra de programación de lo grande son consideradas en el Capítulo 4.

A pesar de todo esto, los lenguajes de tiempo real pueden tener que restringir las posibilidades del programador. Es difícil, si no imposible, estimar el tiempo requerido para acceder a las estructuras de datos dinámicas. Más aún, es deseable poder garantizar que existe memoria suficiente para el programa antes de que comience la ejecución. Por esta razón, los arrays dinámicos y los apuntadores pueden estar ausentes de la lista «acreditada» de posibilidades del lenguaje en el caso de aplicaciones de tiempo real. Además, también pueden estar restringidos la recursión y los bucles infinitos.

Lecturas complementarias

Barnes, J. G. P. (1998), *Programming in Ada 1995*, Reading, MA: Addison-Wesley.

Bishop, J. (2001), *Java Gently*, 3.ª edición, Reading, MA: Addison-Wesley.

Burns, A. (1988), *Programming in occam2*, Reading, MA: Addison-Wesley.

Galletly, J. (1990), *Occam2*, London: Pitman.

Hatton, L. (1995), *Safer C: Developing Software for High-Integrity and Safety-Critical Systems*, London: McGraw-Hill.

INMOS Limited (1988), *occam2 Reference Manual*, Chichester: Prentice Hall.

Kernighan, B. W., y Ritchie, D. M. (1988), *The C Programming Language*, Englewood Cliffs, NJ: Prentice Hall.

Ejercicios

3.1 Ada finaliza cada construcción con **end** `<nombre construcción>`; C no utiliza ninguna marca de finalización. ¿Cuáles son los pros y los contras de los diseños de estos lenguajes?

3.2 Occam2, Java y C son sensibles a las mayúsculas, mientras que Ada no lo es. ¿Cuáles son los argumentos a favor y en contra de la sensibilidad a las mayúsculas?

3.3 Occam2 ha eliminado la necesidad de las reglas de precedencia entre operadores, al exigir el uso de paréntesis en todas las subexpresiones. ¿Cuál es el efecto de esto sobre la legibilidad y la eficiencia?

3.4 ¿Debería un lenguaje exigir siempre la inicialización de las variables?

3.5 ¿La utilización de la sentencia **`exit`** de Ada hace los programas más legibles y fiables?

3.6 ¿Por qué no está permitida la recursión en occam2?

3.7 Cite las características deseables de los lenguajes para una programación segura.

3.8 ¿Por qué Java no soporta tipos apuntadores?

3.9 ¿Hasta qué punto se puede utilizar C en los sistemas de alta integridad?

Capítulo 4

Programar lo grande

En el Capítulo 3, se indicó que la descomposición y la abstracción son los dos métodos más importantes para gestionar la complejidad característica de los sistemas embebidos grandes. Esta complejidad no se debe simplemente a la cantidad de código, sino a la variedad de actividades y requisitos que corresponden a la interacción del mundo real. Como se apuntó en la Sección 1.3.1, el mundo real también está sujeto a continuos cambios. Más aún, el diseño, implementación y mantenimiento del software están a menudo mal gestionados, lo que da como resultado productos insatisfactorios. Este capítulo aborda aquellas características de los lenguajes que ayudan a incorporar, y soportar, la descomposición y el uso de la abstracción. Estas características se dice que ayudan a la *programación de lo grande*.

La estructura clave que falta en los lenguajes más antiguos, como Pascal, es el módulo. Un módulo puede ser descrito, informalmente, como un conjunto de objetos y operaciones relacionadas lógicamente. La técnica de aislar una función del sistema en un módulo y proporcionar una especificación precisa para la interfaz del módulo se denomina **encapsulamiento**. Consecuentemente, con la estructura de módulo es posible soportar:

- *Ocultación de información.*
- Compilación por separado.
- Tipos abstractos de datos.

En las secciones siguientes se describen las principales motivaciones de la estructura de módulo. Estas necesidades se ilustran con ejemplos codificados en Ada, Java y C. Ada y Java soportan módulos explícitamente bajo la forma de **paquetes**. El soporte de módulos en C es débil; sólo lo proporciona indirectamente al permitir la compilación por archivos separados. Occam2, por su parte, no soporta descomposición modular.

A pesar de que el módulo permite encapsulamiento, esencialmente es un mecanismo estático de estructuración, y no una parte del modelo de tipos del lenguaje. Existe un mecanismo de encapsulamiento más dinámico bajo la forma de clases y objetos (que son soportados por Java).

4.1 Ocultación de información

En los lenguajes sencillos, todas las variables permanentes deben ser globales. Si dos o más procedimientos quieren compartir datos, entonces esos datos deben ser visibles para todas las partes del programa. Incluso si un único procedimiento quiere actualizar alguna variable cada vez que es llamado, esta variable debe ser declarada fuera del procedimiento, y por tanto existe una posibilidad de error y mal uso.

La estructura de módulo soporta una visibilidad reducida al permitir que la información esté oculta dentro del «cuerpo» del módulo. Todas las estructuras de módulo (de las que hay muchos modelos distintos) permiten que el programador controle el acceso a sus variables del módulo. Para ilustrar la ocultación de información, consideremos la implementación de una cola dinámica FIFO. La interfaz del gestor de la cola (para el resto del programa) se realiza a través de tres procedimientos que permiten la adición o eliminación de elementos de la cola, y una comprobación de si la cola está vacía. No es deseable que cierta información interna sobre la cola (como por ejemplo, el puntero de cola) sea visible fuera del módulo. A continuación, se presenta un paquete Ada para esta estructura de lista. Los puntos principales a destacar son:

- Un paquete Ada siempre se declara en dos partes: una *especificación* y un *cuerpo*. Sólo las entidades declaradas en la especificación son visibles externamente.
- Ada utiliza «alcance abierto». Todos los identificadores visibles al declarar el paquete podrán ser accedidos dentro del paquete. No existe lista de importación como en algunos lenguajes, por ejemplo Modula-2 y Java.
- En Ada, todos los identificadores exportados podrán ser accedidos desde fuera del paquete mediante el nombre del paquete como identificador requerido; por ejemplo, `ModCola.Vacia`.
- Se mantiene una única instancia de cola, que se crea en la sección de inicialización del paquete, llamando a `Crear`.

```
package ModCola is
-- asumir que el tipo Elemento está en el alcance
function Vacia return Boolean;
procedure Insertar (E : Elemento);
procedure Eliminar (E : out Elemento);
end ModCola;
package body ModCola is

  type Nodo_Cola_T; -- declaración hacia delante
  type Nodo_Cola_Ptr_T is access Nodo_Cola_T;

  type Nodo_Cola_T is
```

```
   record
      Contenido : Elemento;
      Siguiente : Nodo_Cola_Ptr_T;
   end record;

type Cola_T is
   record
      Frente : Nodo_Cola_Ptr_T;
      Final : Nodo_Cola_Ptr_T;
   end record;

type Cola_Ptr_T is access Cola_T;

Q : Cola_Ptr_T;

procedure Crear is
begin
   Q := new Cola_T;
   Q.Frente := null; -- no es estrictamente necesario
   Q.Final := null;  -- ya que los punteros siempre se inician a null
end Crear;

function Vacia return Boolean is
begin
   return Q.Frente = null;
end Vacia;

procedure Insertar(E : Elemento) is
   Nodo_Nuevo : Nodo_Cola_Ptr_T;
begin
   Nodo_Nuevo := new Nodo_Cola_T;
   Nodo_Nuevo.Contenido := E;
   Nodo_Nuevo.Siguiente := null;
   if Vacia then
      Q.Frente := Nodo_Nuevo;
   else
      Q.Final.Siguiente := Nodo_Nuevo;
   end if;
   Q.Final := Nodo_Nuevo;
end Insertar;

procedure Eliminar(E : out Elemento) is
```

```
    Nodo_Antiguo : Nodo_Cola_Ptr_T;
  begin
    Nodo_Antiguo := Q.Frente;
    E := Nodo_Antiguo.Contenido;
    Q.Frente := Q.Frente.Siguiente;
    if Q.Frente = null then
      Q.Final := null;
      end if;
      -- liberar nodo antiguo, ver Sección 4.5.1
    end Eliminar;
  begin
    Crear; -- crear la cola
  end ModCola;
```

Tanto la especificación como el cuerpo de un paquete, deben estar ubicados en la misma parte declarativa, aunque se pueden definir otras entidades entre las dos partes. De esta forma, dos paquetes pueden invocar mutuamente subprogramas sin necesidad de declaraciones anticipadas.

Cualquier paquete puede ser utilizado si está en el alcance. Se dispone de una sentencia «use» para reducir un exceso de nombres.

```
declare
  use ModCola;
begin
  if not Vacia then
    Eliminar(E);
  end if;
end;
```

Las facilidades de empaquetamiento de Java se explican mejor en el contexto de su modelo de programación orientado al objeto, que se describe en la Sección 4.4.

En C, los módulos no están tan bien formalizados. En su lugar, el programador debe utilizar archivos separados: un archivo cabecera (normalmente con la extensión «.h») y un archivo cuerpo (normalmente con la extensión «.c»). Consideremos de nuevo el módulo de la cola; en primer lugar la cabecera (`modcola.h`):

```
/* asumir que elemento está en el alcance */

int vacia();

void insertarE(elemento E);
void eliminarE(elemento *E);
```

Esto define la interfaz funcional del módulo. El usuario del módulo simplemente utiliza este archivo. El cuerpo del módulo se da a continuación:

```
...
#include "modcola.h" /* hacer visible al cuerpo */
                     /* la especificación del módulo */

  struct nodo_cola_t {
     elemento contenido;
     struct nodo_cola_t *siguiente;
  };

  struct cola_t {
     struct nodo_cola_t *frente;
     struct nodo_cola_t *final;
  } *Q; /* Q es ahora un puntero a la estructura cola_t */

  void crear()
  {
     Q = (struct cola_t *) malloc(sizeof(struct cola_t));
     Q->frente = NULL;
     Q->final = NULL;
  };

  int vacia()
  {
     return(Q->frente == NULL);
  }

  void insertarE(elemento E)
  {
     struct nodo_cola_t *nodo_nuevo;

     nodo_nuevo = (struct nodo_cola_t *)
                  malloc(sizeof(struct nodo_cola_t));
     nodo_nuevo->contenido = E;
     nodo_nuevo->siguiente = NULL;
     if(vacia) {
        Q->frente = nodo_nuevo;
     } else {
        Q->final->siguiente = nodo_nuevo;
     };
     Q->final = nodo_nuevo;
  }
```

```
void eliminarE(elemento *E)
  {
     struct nodo_cola_t *nodo_antiguo;

     nodo_antiguo = Q->frente;
     *E = nodo_antiguo->contenido;
     Q->frente = Q->frente->siguiente;
     if(Q->frente == NULL) {
        Q->final = NULL;
     }
     free(nodo_antiguo);
  }
```

Obsérvese que en el lenguaje C no existe relación formal entre un archivo «.h» y uno «.c». Así, la especificación dada en `modcola.h` no tiene por qué tener relación con el código de `modcola.c`; su uso es una mera covención. Por contra, el paquete es una parte integral del lenguaje Ada, y para toda especificación de subprograma debe haber un cuerpo de subprograma. En Ada, la ausencia de un cuerpo de subprograma provoca un error que será detectado en la compilación. En C, no se detectará la inexistencia del cuerpo de una función hasta el momento del montaje.

La construcción modular en la programación de tiempo real es importantísima. Aun así, los módulos no suelen ser entidades de primera clase de los lenguajes de programación: no pueden definirse tipos de módulo, crearse punteros a módulo, etc. Los partidarios del lenguaje dinámico Simula siempre destacan que el concepto de clase ha estado disponible en el lenguaje ya desde finales de la década de 1960. Más aún, Smalltalk-80 ha probado la potencia de una estructura dinámica de módulo. El tema de los módulos dinámicos está muy relacionado con el de la programación orientada al objeto, que se trata en la Sección 4.4.

4.2 Compilación por separado

Una ventaja obvia de la construcción modular es la compilación separada, por módulos; en este caso se dice que el programa se compila en el contexto de una biblioteca. Los programadores pueden por tanto concentrarse en el módulo actual y ser capaces de construir el programa, al menos en parte, de forma que sea posible probar cada módulo. Una vez probada, y posiblemente acreditada, la nueva unidad puede incluirse, en forma precompilada, en la biblioteca. El ahorro de recursos y el apoyo a la gestión del proyecto mejoran claramente si no hay que recompilar todo el programa por cada pequeño cambio.

En Ada (y en C), la especificación y el cuerpo del paquete (módulo), como ya se comentó en la sección previa, pueden precompilarse de forma inmediata. Si una unidad de biblioteca desea acceder a cualquier otra, debe indicarlo explícitamente utilizando una claúsula **`with`** (`#include`):

```
package Distribuidor is
  -- nuevos objetos visibles
end Distribuidor;

with ModCola;
package body Distribuidor is
  -- objetos ocultos
end Distribuidor;
```

De esta forma se construye una jerarquía de dependencias entre unidades de la biblioteca. El propio programa principal utiliza cláusulas **with** para acceder a las unidades de biblioteca que necesita.

Una característica importante del modelo Ada (y en menor medida del C) es que la especificación y cuerpo de un módulo son contemplados en la biblioteca como entidades distintas. Obviamente, ambos elementos deben estar presentes para la compilación final del programa. (Para ser completamente precisos, algunas especificaciones de módulo no necesitan cuerpo; por ejemplo, si sólo definen tipos o variables.) Sin embargo, puede que una biblioteca en desarrollo sólo contenga especificaciones. Éstas podrán utilizarse para comprobar la consistencia lógica del programa antes de su implementación en detalle. En el contexto de la gestión de un proyecto, las especificaciones serán elaboradas por el personal con mayor experiencia; y esto es así porque las especificaciones representan interfaces entre módulos software. Un error en este código es más importante que uno en el cuerpo del paquete, ya que un cambio en la especificación conlleva, potencialmente, tener que cambiar y recompilar cada usuario del módulo, mientras que un cambio en el cuerpo sólo requiere la recompilación del cuerpo.

La compilación por separado permite la programación de «abajo a arriba». Las unidades de biblioteca se construyen a partir de otras unidades; así hasta que se pueda codificar el programa final. La programación «de abajo a arriba», en el contexto del diseño descendente, es ascendente en cuanto a las especificaciones (definiciones), no a las implementaciones (cuerpos), por lo que resulta bastante conveniente. No obstante, Ada ha incluido una característica adicional para la compilación separada que soporta más directamente el diseño descendente. Dentro de una unidad de programa, puede dejarse un «resguardo» (stub), que será substituido posteriormente utilizando la construcción **is separate**. El siguiente ejemplo de esquema muestra cómo puede dejarse sin implementar el procedimiento `Convierte` hasta después de haber definido el programa principal:

```
procedure Main is
  type Lectura is ...
  type Valor_Control is ...
  procedure Convierte (L : Lectura; Vc : out Valor_Control)
            is separate;
begin
  loop
    Entrada (Lc);
```

```
      Convierte(Lc, Vc);
      Salida (Vc);
  end loop;
end;
```

Más tarde se añade el cuerpo del procedimiento:

```
separate (Main)
procedure Convierte (L : Lectura; Vc : out Valor_Control) is
  -- código real requerido
end Convierte;
```

En Ada, la compilación por separado está integrada en la especificación del lenguaje. Más importante es que las reglas de tipado fuerte que se aplicarían si el programa fuera construido como una única unidad, se aplican igualmente sobre unidades de biblioteca. Éste es un mecanismo mucho más fiable que el montaje conjunto de unidades precompiladas de C (y de algunas implementaciones de FORTRAN y Pascal). Con este último enfoque, no es posible una comprobación completa de los tipos. Aun así, C dispone de una herramienta de apoyo específica, lint, que comprueba la consistencia de unidades de compilación.

4.3 Tipos abstractos de datos

En el Capítulo 3 se indicó que una de las mayores ventajas de los lenguajes de alto nivel es que los programadores no tienen que preocuparse de la representación física de los datos en el computador. La idea de los tipos de datos proviene de esta separación. Los tipos abstractos de datos (ADT; abstract data type) son una ampliación de este concepto. Para definir un ADT, un módulo dará nombre a un nuevo tipo, y después proporcionará todas las operaciones que pueden ser aplicadas a ese tipo. La estructura del ADT se oculta dentro del módulo. Al estar permitida más de una instancia de cada tipo, es preciso incluir en la interfaz del módulo una rutina de creación.

Los ADT se complican por el hecho de requerir la compilación por separado de la especificación de un módulo de la de su cuerpo. Dado que la estructura de un ADT debe estar oculta, el lugar lógico de definición es el cuerpo. Pero entonces el compilador no conocerá el tamaño del tipo cuando compila código que utiliza la especificación. Una forma de solucionar el problema es forzar al programador a utilizar un nivel de indirección. Por ejemplo, en C la interfaz del módulo `modcola` quedaría así:

```
typedef struct cola_t *cola_ptr_t;

cola_ptr_t crear();

int vacia(cola_ptr_t Q);

void insertarE(cola_ptr_t Q, elemento E);
void eliminarE(cola_ptr_t Q, elemento *E);
```

Aunque este es un enfoque aceptable, no siempre es apropiado. Por este motivo, Ada también permite que parte de la implementación aparezca en la especificación, pero que sólo sea accesible desde el cuerpo del paquete. A esta parte se le denomina parte **privada** (private) de la especificación.

Para establecer una comparación, siguiendo con el ejemplo de la cola, la definición Ada de un ADT para la cola es de la forma siguiente:

```
package ModCola is
  type Cola is limited private;
  procedure Crear (Q : in out Cola);
  function Vacia (Q : Cola) return Boolean;
  procedure Insertar (Q : in out Cola; E : Elemento);
  procedure Eliminar (Q : in out Cola; E : out Elemento);
private
  -- ninguna de las siguientes declaraciones es visible externamente
  type Nodocola;
  type Colaptr is access Nodocola;
  type Nodocola is
    record
      Contenido : Elemento;
      Siguiente : Colaptr;
    end record;
  type Cola is
    record
      Frente : Colaptr;
      Final : Colaptr; e
    end record;
end ModCola;

package body ModCola is
  -- esencialmente igual que el código original
end ModCola;
```

Las palabras clave **`limited private`** indican que sólo se pueden aplicar sobre el tipo aquellos subprogramas definidos en este paquete. Un tipo *privado limitado* es, por tanto, un verdadero tipo abstracto de dato. Ada, sin embargo, tiene en cuenta que muchos ADT precisan de un operador de asignación y de comprobación de igualdad. En vez de definirlos donde se necesite, el tipo puede ser declarado simplemente como **`private`**. Si este es el caso, entonces, además de los subprogramas definidos, el usuario dispone de la asignación y de la comprobación de igualdad. A continuación, se da un ejemplo de ADT habitual en Ada, que proporciona un paquete de aritmética compleja. Observe que los subprogramas definidos con el tipo `Complejo` toman la forma de operaciones sobrecargadas, y por tanto permiten la escritura «usual» de expresiones aritméticas:

```
package Aritmetica_Compleja is
  type Complejo is private;
  function "+" (X,Y : Complejo) return Complejo;
  function "-" (X,Y : Complejo) return Complejo;
  function "*" (X,Y : Complejo) return Complejo;
  function "/" (X,Y : Complejo) return Complejo;
  function Comp (A,B : Float) return Complejo;
  function Parte_Real (X: Complejo) return Float;
  function Parte_Imaginaria (X: Complejo) return Float;
private
  type Complejo is
    record
      Parte_Real: Float;
      Parte_Imaginaria: Float;
    end record;
end Aritmetica_Compleja;
```

4.4 Programación orientada al objeto

Se acostumbra a llamar *objetos* a las variables de cierto ADT, y paradigma de *programación orientada al objeto* (POO) a la consecuencia de su utilización. No obstante, hay una definición más estricta de la abstracción de objeto (Wegner, 1987) que establece una útil distinción entre objetos y ADT. En general, los ADT carecen de cuatro propiedades que les harían adecuados para la programación orientada al objeto. Éstas son:

(1) Extensibilidad de tipos (herencia).

(2) Inicialización automática de objetos (constructores).

(3) Finalización automática de objetos (destructores).

(4) Selección de operaciones en tiempo de ejecución (polimorfismo).

Todas ellas están soportadas, de una forma u otra, por Ada y Java. En la cola del ejemplo anterior, es necesario declarar una variable cola y después llamar a un procedimiento de inicialización. En POO, esta inicialización (rutina constructora) se realiza automáticamente para cada objeto cola que se declara. De igual modo, se ejecuta un procedimiento destructor cuando un objeto queda fuera de alcance.

Las propiedades (2) y (3) son útiles, pero el concepto clave de la abstracción de objeto es la extensibilidad. Ésta permite definir un tipo mediante la extensión de otro tipo definido previamente. El nuevo tipo hereda del tipo «base», y puede incluir nuevos campos y nuevas operaciones. Una vez que el tipo ha sido extendido, es preciso seleccionar sobre la marcha cada una de las operaciones solicitadas para asegurar que se invoca a la operación apropiada de una instancia particular de la familia de tipos.

C no permite la POO, aunque una extensión de C, C++, se ha hecho muy popular, y es ya un estándar ISO.

Las implicaciones generales del desarrollo de software orientado al objeto de software quedan fuera del alcance de este libro. No obstante, es necesario dominar los principios de la POO para poder comprender las funcionalidades que ofrece Java para la concurrencia y el tiempo real.

4.4.1 Ada y la POO

Ada permite la programación orientada al objeto a través de dos mecanismos complementarios que proporcionan extensiones de tipo y polimorfismo dinámico: tipos etiquetados y tipos de clases generales. Los paquetes hijos y los tipos controlados son también características importantes del soporte que proporciona Ada.

Tipos etiquetados

En Ada, se puede *derivar* un nuevo tipo a partir de un tipo existente, y utilizar subtipado para cambiar algunas de las propiedades del tipo. El ejemplo siguiente declara un nuevo tipo y un subtipo, denominado `Ajuste`, que tiene las mismas propiedades que el tipo `Integer` pero con un rango restringido. `Ajuste` e `Integer` son distintos, y no pueden intercambiarse:

```
type Ajuste is new Integer range 1 .. 100;
```

Se pueden definir nuevas operaciones para manipular `Ajuste`; sin embargo, no se pueden añadir nuevos componentes. Los tipos etiquetados (tagged) eliminan esta restricción, y permiten la adición de componentes adicionales al tipo. Cualquier tipo que potencialmente pueda ser extendido de esta forma, debe declararse como tipo etiquetado. Debido a que la extensión de un tipo implica que el tipo sea un registro, sólo podrán etiquetarse los tipos registro (o los tipos privados implementados como registros). Consideremos, por ejemplo, el siguiente tipo y una operación primitiva:

```
type Coordenadas is tagged
  record
     X : Float;
     Y : Float;
  end record;

procedure Dibujar(P: Coordenadas);
```

Este tipo puede ser extendido:

```
type Tres D is new Coordenadas with
  record
     Z : Float;
  end record;
```

```
procedure Dibujar(P: Tres D); -- reemplaza al subprograma Dibujar
Punto : Tres D := (X => 1.0, Y => 1.0, Z => 0.0);
```

Todos los tipos derivados de esta forma (incluyendo el inicial **root**) se dice que pertenecen a la misma jerarquía de **clases**. Al extender un tipo, éste hereda automáticamente cualquier operación primitiva (aquéllas definidas en el tipo) disponible en el tipo padre.

Los campos de la clase `Tres_D` del ejemplo anterior son directamente visibles, para los usuarios del tipo. Ada 95 también permite que estos atributos estén completamente encapsulados mediante tipos privados:

```
package Clase_Coordenada is
  type Coordenadas is tagged private;

  procedure Dibujar(P: Coordenadas);

  procedure Establecer_X(P: Coordenadas; X: Float);
  function Obtener_X(P: Coordenadas) return Float;
  -- similar para Y
private
  type Coordenadas is tagged
  record
    X : Float;
    Y : Float;
  end record;
end Clase_Coordenada;
```

Otros mecanismos son la noción de tipo etiquetado abstracto y la operación primitiva abstracta. Éstos son similares a los de Java que se verán en la Sección 4.4.2. Tenga en cuenta que Ada sólo permite la herencia de un único padre, aunque se puede conseguir herencia múltiple utilizando la genericidad del lenguaje.

Tipos de clases generales

Los tipos etiquetados proporcionan el mecanismo para extender incrementalmente los tipos. Como resultado, el programador podrá crear una jerarquía de tipos. Otras partes del programa podrían querer manipular para sus propios fines a cualquier miembro de esa jerarquía, pero sin importar qué miembro está procesando en cada momento. Ada es un lenguaje fuertemente tipado, por lo que es necesario un mecanismo que permita que pueda ser pasado como parámetro un objeto de cualquier miembro de la jerarquía. Por ejemplo, un subprograma podría querer tomar como parámetro una coordenada, sin importar demasiado que vea bidimensional o tridimensional.

La programación de clases generales es la técnica que permite escribir programas que manipulan familias de tipos. Para cada tipo etiquetado como `T`, existe un tipo asociado `T'Class` que comprende todos los tipos de la familia que comienzan con `T`. Así, el siguiente subprograma aceptará coordenadas bidimensionales o tridimensionales.

```
procedure Dibujar_General(P : Coordenadas'Class);
```

Cualquier llamada a una operación primitiva sobre un tipo de clase general, dará como resultado la llamada a la operación correcta para el tipo real utilizado; este proceso se conoce como **selección en tiempo de ejecución**.

```
procedure Dibujar_General(P : Coordenadas'Class) is
begin
   -- hacer alguna tarea
   Dibujar(P);
   -- dependiendo del valor real de P, se llamará a
   -- uno de los procedimientos Dibujar ya definidos
end Dibujar_General;
```

Aunque la selección en tiempo de ejecución es un mecanismo potente, origina algunos problemas para los sistemas de tiempo real. En particular, no es posible conocer por análisis estático del código qué operaciones son invocadas. Esto dificulta el análisis estático de la temporización (véase la Sección 13.7).

Paquetes hijo

La principal función de los paquetes hijo es añadir más flexibilidad al empaquetamiento simple. Sin ellos, los cambios en un paquete que requirieran modificaciones en la especificación, implicarían la recompilación de todos los clientes que utilizaran este paquete. Esto está reñido con la programación orientada al objeto, que facilita los cambios incrementales. Más aún, no es posible extender los tipos etiquetados privados sin añadidos adicionales al lenguaje, ya que en los tipos privados sólo se puede acceder a los datos desde dentro del cuerpo del paquete.

Considere el siguiente ejemplo de la sección anterior:

```
package Clase_Coordenada is
   type Coordenadas is tagged private;

   procedure Dibujar(P: Coordenadas);

   procedure Establecer_X(P: Coordenadas; X: Float);
   function Obtener_X(P: Coordenadas) return Float;
   -- similar para Y
private
   type Coordenadas is tagged
   record
      X : Float;
      Y : Float;
   end record;
end Clase_Coordenada;
```

Para extender esta clase y poder ver los atributos del padre sería necesario editar el paquete.

Los paquetes hijo permiten acceder directamente a los componentes privados del padre sin tener que acceder a través de sus interfaces. Por tanto, el siguiente código,

```
package Clase_Coordenada.Tres_D is
   -- "." indica que el paquete Tres_D es un hijo de Clase_Coordenada

   type Tres_D is new Coordenadas with private;

   -- operaciones primitivas nuevas
   procedure Establecer_Z(P: Coordenadas; Z: Float);
   function Obtener_Z(P: Coordenadas) return Float;

   procedure Dibujar(P: Tres_D); -- reemplaza al subprograma Dibujar
private

   type Tres_D is new Coordenadas with
   record
      Z : Float;
   end record;
end Clase_Coordenada.Tres_D;
```

permite implementar una nueva operación primitiva que accede a los atributos de datos de la clase original.

Tipos controlados

Los tipos *controlados* (controlled) proporcionan un soporte adicional para la programación orientada al objeto. Mediante ellos es posible definir subprogramas a los que se invoca automáticamente cuando se da alguna de las siguientes circunstancias con estos tipos:

- Se crea un objeto: *initialize* (inicializa).
- Deja de existir un objeto: *finalize* (finalizar).
- Se asigna un nuevo valor al objeto: *adjust* (ajusta).

Para acceder a estas características, el tipo debe ser derivado de `Controlled`, un tipo predefinido declarado en el paquete de biblioteca `Ada.Finalization`; esto es, debe formar parte de la jerarquía de la clase `Controlled`. El paquete `Ada.Finalization` define los procedimientos `Initialize`, `Finalize` y `Adjust`. Cuando se deriva un tipo de `Controlled`, es posible remplazar los procedimientos por defecto. Dado que los objetos dejan de existir cuando quedan fuera del alcance, la salida de un bloque puede implicar cierto número de llamadas a `Finalize`.

4.4.2 Java y la POO

En general, existen al menos dos enfoques a la hora de incluir mecanismos de programación orientada al objeto en un lenguaje. Uno es el propuesto por el lenguaje de programación Oberon (Wirth, 1988), que proporciona un mecanismo de extensión de tipos; éste fue el enfoque elegido por Ada. El enfoque alternativo (y el más popular) consiste en introducir la noción de clase directamente en el lenguaje.

En el Capítulo 3 se presentó una sencilla clase para la fecha en Java:

```
class Fecha
{
  int dia, mes, año;
}
```

Este ejemplo solamente muestra cómo se agrupan elementos de datos. Las posibilidades completas de la clase permiten el encapsulamiento de elementos de datos (variables de instancia o campos, como son llamados en Java), y por tanto que se pueda construir fácilmente un tipo abstracto de datos.

A continuación, se desarrollará en Java la clase para la abstracción de la cola. En primer lugar, se puede declarar un paquete que contenga la cola (si no se da nombre al paquete, el sistema asume un paquete sin nombre), donde también se pueden inportar otros paquetes. A continuación vienen las declaraciones de las clases dentro del paquete. Una de estas clases, (`Cola`), se declara «pública», y ésta es la única a la que se podrá acceder desde fuera del paquete. En Java, la palabra clave **`public`** se denomina «modificador». Otros modificadores para clases son **`abstract`** y **`final`**. Una clase abstracta es aquélla de la que no se pueden crear objetos, y que por tanto deberá ser extendida (para producir una subclase) y convertida en no abstracta para que los objetos puedan existir. El modificador final indica que de la clase no se puede obtener una subclase. La ausencia de modificador indica que la clase es accesible sólo dentro del paquete.

Cada clase puede declarar variables de instancia locales (o campos), métodos constructores, y *métodos (miembros) ordinarios*. Los métodos constructores tienen el mismo nombre que sus clases asociadas. Cuando se crean objetos de una clase se llama al método constructor apropiado. Todos los métodos de un objeto pueden también tener modificadores asociados. Éstos determinan la accesibilidad del método; **`public`**, **`protected`** y **`private`** son los modificadores permitidos: **`public`** permite acceso total, **`protected`** permite el acceso desde el mismo paquete o desde una subclase de la clase en la que se define, y **`private`** permite el acceso solamente desde la clase de la definición. Las variables de instancia también pueden tener estos modificadores. Los métodos miembro y las variables de instancia pueden tener otros modificadores que serán presentados a lo largo del libro.

A continuación viene el código para el tipo abstracto de datos cola.

```
import algunpaquete.Elemento; // importar tipo elemento
package colas; // nombre del paquete
```

```
class NodoCola // clase local al paquete
{
  Elemento dato;
  NodoCola siguiente;
}

public class Cola // clase accesible desde fuera del paquete
{

  NodoCola frente, final; // variables de instancia

  public Cola() // constructor público
  {
    frente = null;
    final = null;
}

public void insertar(Elemento E) // método visible
{
  NodoCola nodoNuevo = new NodoCola();

  nodoNuevo.dato = E;
  nodoNuevo.siguiente = null;
  if(vacia()) {
    frente = nodoNuevo;
  } else {
    final.siguiente = nodoNuevo;
  }
  final = nodoNuevo;
}

public Elemento eliminar () // método visible
{
  if(!vacia()) {
    Elemento ImpE = frente.dato;
    frente = frente.siguiente;
    if(vacia()) final = null;
  }
  // la recolección de basura liberará el objeto NodoCola
  // que está ahora colgando
  return tmpE;
```

```
  public boolean vacia() // método visible
  {
     return (frente == null);
  }
}
```

Obsérvese que no existe el concepto de especificación de paquete, como en Ada. Sin embargo, se puede proporcionar una funcionalidad similar utilizando interfaces (véase la Sección 4.5.2).

4.4.3 Java y la herencia

La herencia en Java se obtiene derivando unas clases de otras. La herencia múltiple no está soportada, aunque se pueden alcanzar efectos similares utilizando interfaces (véase la Sección 4.5.2).

Consideremos de nuevo el ejemplo de las coordenadas; una estructura de clase para este tipo abstracto es:

```
package coordenadas;

public class Coordenadas // recuérdese que Java distingue mayúsculas
{
  float X;
  float Y;

  public Coordenadas(float X_inicial, float Y_inicial) // constructor
  {
     X = X_inicial;
     Y = Y_inicial;
  };

  public void establecer(float F1, float F2)
  {
     X = F1;
     Y = F2;
  };

  public float obtenerX()
  {
     return X;
  };

  public float obtenerY()
  {
```

```
    return Y;
  };
};
```

Esta clase puede ahora ser extendida para producir una nueva clase, poniendo el nombre de la clase base en la declaración de la clase derivada junto con la palabra clave «extends». La nueva clase se encuentra en el mismo paquete.[1]

```
package coordenadas;
public class TresDimensiones extends Coordenadas {
  // subclase de Coordenadas

  float Z; // nuevo campo

  public TresDimensiones(float XInicial, float YInicial,
         float ZInicial) // constructor
  {
    super(XInicial, YInicial); // llamada al constructor de la superclase
    Z = ZInicial;
  };

  public void establecer(float F1, float F2, float F3) //método redefinido
  {
    establecer(F1, F2); // llamada a establecer de la superclase
    Z = F3;
  };

  public float obtenerZ() // nuevo método
  {
    return Z;
  };
};
```

Se ha añadido un nuevo campo `Z` y se ha definido el constructor de la clase, se ha redefinido la función `establecer`, y se ha proporcionado una nueva operación. El constructor llama al constructor de la clase base (mediante la palabra clave **`super`**) e inicializa la última dimensión. De forma similar, la función `establecer` llama a la clase base.

A diferencia de Ada, todas las llamadas a métodos en Java son potencialmente despachables (vinculadas dinámicamente). Por ejemplo, consideremos otra vez el ejemplo anterior:

[1] En Java, las clases públicas deben residir en su propio archivo. Por tanto, un paquete puede estar distribuido en uno o más archivos, y puede agrandarse a voluntad.

```
package coordenadas;

public class Coordenadas
{
  float X;
  float Y;

  public Coordenadas(float X_inicial, float Y_inicial)
  {
    X = X_inicial;
    Y = Y_inicial;
  };

  public void establecer(float F1, float F2)
  {
    X = F1;
    Y = F2;
  };

  public float obtenerX()
  {
    return X;
  };

  public float obtenerY()
  {
    return Y;
  };

  public void dibujar()
  {
  // dibujar un punto bidimensional
  };
};
```

Aquí se ha añadido el método `dibujar`. Ahora, si se redefine `dibujar` en una (sub)clase hija:

```
package coordenadas;
public class TresDimensiones extends Coordenadas
{
```

```
  public TresDimensiones(float XInicial, float YInicial,
                         float ZInicial)
  {
    super(XInicial, YInicial);
    Z = ZInicial;
  };

  float Z;

  void establecer(float F1, float F2, float F3)
  {
    establecer(F1, F2);
    Z = F3;
  };

  float obtenerZ()
  {
    return Z;
  };

  public void dibujar()
  {
    // dibujar un punto tridimensional
  };
};
```

Entonces el siguiente código:

```
{
  Coordenadas A = new Coordenadas(0f, 0f);
  A.dibujar();
}
```

dibujaría unas coordenadas de dos dimensiones, mientras que

```
{
  Coordenadas A = new Coordenadas(0f, 0f);
  TresDimensiones B = new TresDimensiones(0f, 0f, 0f);

  A =B;
  A.dibujar();
}
```

dibujaría unas coordenadas de tres dimensiones, a pesar de que A fue declarada originalmente del tipo Coordenadas. Esto es así porque A y B son de tipo referencia. La asignación de B a A cambia solamente la referencia, pero no el propio objeto.

4.4.4 La clase Object

Todas las clases en Java son subclases implícitas de una clase raíz llamada Object. En el Programa 4.1 se da la definición de esta clase (throw se discutirá en la Sección 6.3.2).

Hay seis métodos en la clase Object que son de interés para este libro. Los tres métodos wait y los dos notify se utilizan para el control de la concurrencia, y se considerarán en la Sección 8.8. El sexto método es finalize. Se trata del método que se invoca justo antes de que el objeto sea destruido. Por tanto, reemplazando este método, una clase hijo puede proporcionar

Programa 4.1. La clase *Object* de Java.

```
public class Object {
   public final class getClass();

   public String toString();
   public boolean equals(Object obj);
   public int hashCode();

   protected Object clone()
      throws CloneNotSupportedException;

   public final void wait()
      throws IllegalMonitorStateException,
         InterruptedException;
   public final void wait(long millis)
      throws IllegalMonitorStateException,
         InterruptedException;
   public final void wait(long millis, int nanos)
      throws IllegalMonitorStateException,
         InterruptedException;
   public final void notify()
      throws IllegalMonitorStateException;
   public final void notifyAll()
      throws IllegalMonitorStateException;

   protected void finalize()
   throws Throwable;
}
```

la finalización de los objetos que se crean a partir de ella. Por supuesto, cuando la clase redefine el método `finalize`, la última acción debería ser llamar al método `finalize` de su clase padre. Sin embargo, debe tenerse en cuenta que `finalize` sólo se invoca cuando va a tener lugar la recolección de basura, y esto puede ocurrir después de que el objeto esté en uso; no existen métodos destructores como, por ejemplo, en C++. De ahí que esta facilidad no deba ser utilizada para recuperar recursos. Es preferible utilizar las facilidades del manejo de excepciones como alternativa (véase la Sección 6.3.2).

4.5 Reusabilidad

La producción de software es un negocio caro, cuyo coste crece ano tras año inexorablemente. Una de las razones de ello es que parece que el software siempre se construye «desde cero». En comparación, el ingeniero hardware dispone de una amplia elección de componentes probados y validados con los que construir los sistemas. El disponer de un suministro similar de componentes software ha sido una demanda constante de los ingenieros de software. Sin embargo, aparte de alguna área concreta (por ejemplo, el análisis numérico), esto no ha sido así, aunque la reutilización del código tendría un claro efecto beneficioso, tanto en la fiabilidad como en la productividad de la construcción del software.

Las técnicas de los lenguajes de programación modernos, como la programación modular y la programación orientada al objeto, proporcionan las bases sobre las que se pueden construir bibliotecas de software reutilizable.

Un impedimento para la reutilización es el modelo de tipado fuerte recomendado en el Capítulo 3. Con este modelo, el módulo de ordenación de enteros, por ejemplo, no sirve para ordenar reales o registros, aunque los algoritmos básicos sean idénticos. Este tipo de restricción, aunque necesario por otros motivos, limita severamente la reutilización. Los diseñadores de Ada y Java afrontaron este problema, y proporcionaron ayudas que permiten la reutilización sin perjuicio del modelo de tipos. En Ada este recurso se basa en el concepto de módulo genérico; en C++ se proporciona un recurso similar que permite definir clases genéricas, denominadas **plantillas** (templates); y en Java, se utiliza un método diferente que se sirve de la noción de interfaz. En las dos secciones siguiente se comentarán dichos conceptos.

4.5.1 Programación genérica en Ada

Un **genérico** es una plantilla de la que se pueden **instanciar** componentes reales. En esencia, un genérico manipula objetos sin tener en cuenta su tipo, y sólo al instanciarlo se especifica su tipo real. El modelo del lenguaje asegura que al instanciar el genérico se comprueba cualquier condición hecha sobre el tipo dentro del genérico. Por ejemplo, si en un genérico se supone que el tipo de un **parámetro genérico** es discreto, al realizar la instanciación el compilador comprobará que el tipo del parámetro actual lo es también.

Una medida de la reusabilidad de un genérico se relaciona con las restricciones impuestas sobre los parámetros genéricos. En un extremo, si el tipo instanciado tiene que ser, por ejemplo, un

array de enteros de una dimensión, entonces el genérico no es particularmente reutilizable. En el otro, si se puede instanciar sobre cualquier tipo, puede obtenerse un alto nivel de reutilización.

El modelo de parámetros para los genéricos de Ada es exhaustivo, y no se describirá en detalle; en lugar de ello, se mostrarán dos ejemplos que ilustran una alta reusabilidad.

En el cuerpo del genérico es donde se definen las operaciones que se aplican sobre el parámetro genérico. Si no se aplica ninguna operación, se dice que los parámetros son **privados limitados** (limited private), y si sólo se realizan asignaciones y comprobaciones de igualdad, se dice que son **privados** (private). Los componentes con alta reutilización tienen parámetros privados o privados limitados. Como primer ejemplo, consideremos el paquete `ModCola`. En lo descrito hasta el momento, y a pesar de que se ha proporcionado un tipo abstracto de datos para cola, todas las colas deben manejar objetos del tipo `Elemento`. Queda clara la necesidad de un genérico donde se maneje el tipo del objeto como parámetro. En el cuerpo del paquete `ModCola` solamente se introducían y eliminaban objetos del tipo `Elemento` en las colas. Este parámetro, por tanto, puede ser privado.

```
generic
  type Elemento is private;
package ModCola_Plantilla is
  type Cola is limited private;
  procedure Crear (Q : in out Cola);
  function Vacia (Q : Cola) return Boolean;
  procedure Insertar (Q : in out Cola; E : Elemento);
  procedure Eliminar (Q : in out Cola; E : out Elemento);
private
  type Nodocola;
  type Colaptr is access Nodocola;
  type Nodocola is
    record
      Contenido : Elemento;
      Siguiente : Colaptr;
    end record;
  type Cola is
    record
      Frente : Colaptr;
      Final : Colaptr;
    end record;
end ModCola_Plantilla;

package body ModCola_Plantilla is
  -- lo mismo que antes
end ModCola_Plantilla;
```

Al instanciar este genérico se crea un paquete real:

```
declare
  package Colas_Enteros is new ModCola_Plantilla(Integer);
  type ProcesoId is
    record
      ...
    end record;
  package Colas_Procesos is new ModCola_Plantilla(ProcesoId);
  Q1, Q2 : Colas_Enteros.Cola;
  Pid : Colas_Procesos.Cola;
  P : ProcesoId;
  use Colas_Enteros;
  use Colas_Procesos;
begin
  Crear(Q1);
  Crear(Pid);
  ...
  Insertar(Pid,P);
  ...
end;
```

Cada uno de estos paquetes define un tipo abstracto de dato para colas, siendo cada uno de ellos diferente, puesto que los tipos de los elementos son distintos.

La discusión anterior se ha centrado en los parámetros genéricos como tipos, pero se dispone de otras tres formas: constantes, subprogramas y paquetes. Los búferes son construcciones muy frecuentes en los programas de tiempo real, y se diferencian de las colas en que su tamaño es fijo. A continuación se muestra la especificación de un paquete genérico para un tipo abstracto de datos de búferes. De nuevo, el tipo del elemento es un parámetro genérico, al que se añade el tamaño del búfer como parámetro constante genérico (su valor por defecto es 32):

```
generic
  Tamaño : Natural := 32;
  type Elemento is private;
package Bufer_Plantilla is
  type Bufer is limited private;
  procedure Crear(B : in out Bufer);
  function Vacia(B : Buffer) return Boolean;
  procedure Poner(B : in out Bufer; E : Elemento);
  procedure Tomar(B : in out Bufer; E : out Elemento);
private
  subtype Rango_Bufer is Natural range 0..Tamaño-1;
  type Buf is array(Bufer_Rango) of Elemento;
  type Bufer is
```

```
    record
      Bf : Buf;
      Top : Rango_Bufer := 0;
      Base: Rango_Bufer := 0;
    end record;
end Bufer_Plantilla;
```

Un búfer de enteros de tamaño 32 se instancia de la siguiente manera:

```
package Bufer_Enteros is new Bufer_Plantilla(Integer);
```

Un búfer de 64 elementos de cierto tipo registro `Reg` se construye con una instanciación similar:

```
package Bufer_Reg is new Bufer_Plantilla(64,Reg);
```

Igual que ocurría con los parámetros de los subprogramas, se alcanza una mayor legibilidad mediante la asociación de nombres:

```
package Bufers_Reg is new Bufer_Plantilla(Tamaño => 64, Elemento => Reg);
```

A continuación se muestra un ejemplo de parámetro de subprograma genérico. El paquete genérico define dos procedimientos que actúan sobre un array de `Elementos`: uno de ellos encuentra el `Elemento` mayor, y el otro ordena el array. Para implementar dichos procedimientos es preciso poder comparar pares de `Elementos` para ver cuál es mayor. Para tipos escalares se dispone del operador «>», pero no para cualquier tipo privado. En consecuencia, el paquete genérico debe importar la función «>».

```
generic
  Tamaño : Natural;
  type Elemento is private;
  with function ">"(E1, E2 : Elemento) return Boolean;
package Ordenar_Array is
  type Vector is array(1..Tamaño) of Elemento;
  procedure Ordenar(V: in out Vector);
  function Mayor(V: Vector) return Elemento;
end Ordenar_Array;
```

La implementación de este genérico se deja como ejercicio para el lector (Ejercicio 4.8).

Se pueden crear paquetes genéricos más sofisticados utilizando paquetes (posiblemente ellos mismos instanciaciones de genéricos) como parámetros genéricos, pero no se verán, por no ser necesarios para el material que se presentará en los capítulos restantes.

4.5.2 Interfaces en Java

Las interfaces en Java se añaden a las clases para aumentar la reutilización de código. Una interfaz es una forma de clase especial que especifica un conjunto de métodos y constantes. Por definición son abstractas, y no se pueden instanciar directamente, aunque pueden ser implementadas por otras clases. Los objetos que implementan interfaces pueden ser pasados como argumentos de métodos, definiendo el parámetro como de tipo interfaz. Las interfaces sirven para permitir construir relaciones entre clases al margen de la jerarquía de clases.

Considere un algoritmo «genérico» que ordena arrays de objetos. Es común a todas las clases que pueden ser ordenadas el admitir el operador «<» o «>». Dicha característica puede encapsularse en una interfaz.

La interfaz `Ordenado` define un único método, `menorQue`, que toma como argumento un objeto cuya clase implementa la interfaz `Ordenado`. Cualquier clase que implemente la interfaz `Ordenado` debe comparar el objeto de ordenación con el argumento que se pase, y devolver si es menor que éste[2].

```
package ejemplosInterfaz;

public interface Ordenado {
  boolean menorQue (Ordenado O);
};
```

La clase de números complejos es:

```
import ejemplosInterfaz.*;
class NumeroComplejo implements Ordenado
{
  // la clase implementa la interfaz Ordenado

  protected float parteReal;
  protected float parteImag;

  public boolean menorQue(Ordenado O) // la implementación de la interfaz
  {
    NumeroComplejo NC = (NumeroComplejo) O; // coerción del parámetro

    if((parteReal*parteReal + parteImag*parteImag) <
       (NC.obtenerReal()*NC.obtenerReal() + NC.obtenerImag()*NC.obtenerImag()))
    {
```

[2] Compárese esto con el estilo Ada, donde la comparación de dos objetos se realiza en una función que recibe **ambos** objetos como parámetros.

```
        return true;
      }
      return false;
    };

    public NumeroComplejo (float I, float J) // constructor
    {
      parteReal = I;
      parteImag = J;
    };

    public float obtenerReal()
    {
      return parteReal;
    };

    public float obtenerImag()
    {
      return parteImag;
    };

}
```

Ahora es posible escribir algoritmos que ordenen ésta y cualquier otra clase que implemente la interfaz. Por ejemplo, en la Sección 4.5.1 se da un genérico Ada para `Ordenar_Array`. El paquete equivalente en Java sería:

```
package ejemplosInterfaz;

public class OrdenarArray
{
  public static void ordenar (Ordenado oa[], int tamaño) // método ordenar
  {
    Ordenado tmp;
    int pos;

    for (int i = 0; i < tamaño - 1; i++) {
      pos = i;
      for (int j=i +1;j <tamaño; j++) {
        if (oa[j].menorQue(oa[pos])) {
          pos = j;
        }
      }
```

```
            tmp = oa[pos];
            oa[pos] = oa[i];
            oa[i] = tmp;
        };
    };

    public static Ordenado mayor(Ordenado oa[], int tamaño) // método mayor
    {
        Ordenado tmp;
        int pos;
        pos =0;
        for (int i = 1; i < tamaño; i++) {
            if (! oa[i].menorQue(oa[pos])) {
                pos = i;
            };
        };
        return oa[pos];
    };
}
```

El método `ordenar` toma dos argumentos: el primero es un array de objetos que implementa la interfaz `Ordenado`, y el segundo es el número de elementos del array. La implementación realiza una ordenación por intercambio. Un detalle importante del ejemplo es que al intercambiar dos objetos, sólo se intercambian los valores de referencia, por lo que no importa el tipo del objeto (siempre que extienda la interfaz `Ordenado`).

La utilización de la clase e interfaz anteriores simplemente requiere lo siguiente:

```
{ OrdenarArray AR = new OrdenarArray();
  NumeroComplejo arrayComplejo[] = { // say
          new NumeroComplejo(6f,1f),
          new NumeroComplejo(1f, 1f),
          new NumeroComplejo(3f,1f),
          new NumeroComplejo(1f, 0f),
          new NumeroComplejo(7f,1f),
          new NumeroComplejo(1f, 8f),
          new NumeroComplejo(10f,1f),
          new NumeroComplejo(1f, 7f)
  };
  // array desordenado
  AR.ordenar(arrayComplejo, 8);
  // array ordenado
}
```

De hecho, el paquete `java.lang` ya define una interfaz, llamada `Comparable`, con una función denominada `compareTo`, que podría ser utilizada en vez de `Ordenado`. Más aún, hay un método estático en `java.util.Arrays`, llamado `sort`, que implementa una ordenación de objetos por mezcla.

Las interfaces tienen tres propiedades adicionales a considerar: en primer lugar, al igual que las clases, pueden participar en relaciones de herencia simple; en segundo lugar, una clase puede implementar más de una interfaz, con lo que se puede conseguir gran parte de la funcionalidad de la herencia múltiple; finalmente, las interfaces también proporcionan los mecanismos con los que implementar devoluciones de llamada. Esto ocurre cuando un servidor necesita invocar a un método definido en la clase del que hace la invocación.

Resumen

En la evolución de los lenguajes de programación, una de las construcciones más importantes que ha aparecido es el módulo. Esta estructura permite manejar y gestionar la complejidad inherente a los sistemas de tiempo real. En particular, soporta:

(1) La ocultación de información.

(2) La compilación por separado.

(3) Los tipos abstractos de datos.

Tanto Ada como C tienen una estructura estática de módulo. Ada utiliza «alcance abierto», con lo que todos los objetos que están en el alcance en la declaración del módulo, son visibles en él. C permite módulos sólo de manera informal, mientras que Java soporta una estructura dinámica de módulo en forma de clases. Tanto los paquetes en Ada (y Java) como las clases en Java poseen especificaciones bien definidas que sirven de interfaz entre el módulo y el resto del programa.

La compilación por separado permite la construcción de bibliotecas de componentes precompilados. Esto favorece la reutilización, y proporciona un repositorio para el software. Este repositorio, sin embargo, debe gestionarse adecuadamente, de forma que asuntos como el control de versiones no se convierta en una fuente de falta de fiabilidad.

La descomposición de un programa grande en módulos o clases es la esencia de la *programación de lo grande*. De cualquier forma, es importante que este proceso de descomposición produzca módulos bien definidos.

La utilización de los tipos abstractos de dato (ADT), o programación orientada al objeto, proporciona una de las principales herramientas que los programadores pueden utilizar para gestionar grandes sistemas software. De nuevo, es la construcción módulo, tanto en Ada como en Java, la que permite construir y utilizar ADT.

Los lenguajes fuertemente tipados restringen la facilidad de reutilización de los módulos, ya que su comportamiento viene ligado a los tipos de sus parámetros y subcomponentes. Esta dependecia es a menudo mayor de lo que sería necesario. Ada y C++ proporcionan una primitiva de genéricos que intenta mejorar la reusabilidad del software: es posible definir paquetes y procedimientos genéricos que sirven de plantillas con las que instanciar código real. En Java se consigue el mismo efecto mediante interfaces. El uso adecuado de estos recursos debe reducir costes, así como aumentar la fiabilidad, de los programas de tiempo real.

Lecturas complementarias

Ben-Ari, M. (1998), *Ada for Software Engineers,* Chichester: Wiley.

Darnell, P. A., y Margolis, P. E. (1996), C: *A Software Engineering Approach,* London: Springer-Verlag.

Meyer, B. (1992), *Eiffel: The Language,* New York: Prentice Hall.

Sommerville, I. (2001), *Software Engineering,* 6th Edition, Reading, MA: Addison-Wesley.

Horstmann, C. S., y Cornell, G. (1999), *Core Java Fundamentals,* Sun Microsystems.

Schach, S. R. (1997), *Software Engineering with Java,* Chicago, IL: Richard Irwin.

Ejercicios

4.1 ¿Por qué no es suficiente el procedimiento como módulo de programa?

4.2 Distinga entre compilación por separado, compilación independiente y multiprogramación.

4.3 Evalúe la aproximación de C a la programación modular.

4.4 ¿Qué ventajas y desventajas supondría el que los paquetes Ada fueran elementos de primera clase del lenguaje?

4.5 Compare y contraste los mecanismos de Ada y C para el paso de funciones como parámetros a otras funciones.

4.6 Compare y contraste los recursos de POO en Ada y Java.

4.7 Defina un tipo abstracto de datos para el tiempo. ¿Qué operaciones son aplicables a los valores del tiempo?

4.8 Implemente el paquete genérico Ada dado en la Sección 4.5.1. Ilustre su uso mostrando cómo ordenar un array de búferes (teniendo en cuenta que un búfer es «mayor que» otro si tiene más elementos).

4.9 La POO permite despachar los métodos en tiempo de ejecución. ¿Debería la programación de tiempo real utilizar únicamente vinculación estática?

4.10 En Java, todos los objetos se asignan en memoria dinámicamente. Discuta qué implicaciones tiene la recolección de basura para la programación de tiempo real.

4.11 ¿Hasta qué punto Ada 95 y Java permiten la herencia múltiple?

4.12 Ada 95 soporta la noción de paquetes hijos. Discuta el papel que juegan en el soporte general de Ada a la POO.

Capítulo 5

Fiabilidad y tolerancia a fallos

Los requisitos de fiabilidad y seguridad de los sistemas de tiempo real y embebidos habitualmente son mucho más estrictos que los de otros sistemas informáticos. Por ejemplo, si una aplicación que calcula la solución a algún problema científico falla, entonces la finalización del programa podría ser una solución razonable, ya que sólo se habrá perdido tiempo de cálculo. Sin embargo, en el caso de un sistema embebido esto puede no ser una acción aceptable. Un computador de control de procesos, por ejemplo, responsable de una gran caldera de gas, no puede decidir apagar la caldera tan pronto como se produzca un defecto. En lugar de esto, debe intentar proporcionar un servicio reducido y prevenir una costosa operación de apagado. En casos más dramáticos, los sistemas informáticos de tiempo real pueden poner en peligro vidas humanas si abandonan su objetivo de control. Un computador embebido que controla un reactor nuclear no debe dejar al reactor fuera de control, ya que esto produciría la fusión del núcleo y la emisión de radiación. Un sistema de navegación aérea debería posibilitar que el piloto saltase fuera del avión antes de permitir que el avión se estrellara.

En los tiempos actuales, cada vez más funciones de control antes realizadas por operadores humanos o mediante probados métodos analógicos, están siendo administradas por computadores digitales. En 1955, sólo el 10 por ciento de los sistemas de armamento de EEUU necesitaba programas de computador. Al principio de la década de 1980, este número creció hasta el 80 por ciento (Leveson, 1986). Existen muchos ejemplos en los que defectos en los programas han producido fallos en misión. En los primeros años de la década de 1970, el programa responsable de controlar los globos meteorológicos de gran altura de un satélite meteorológico francés emitió una solicitud de «autodestrucción de emergencia» en lugar de una petición de «lectura de datos». El resultado fue la destrucción de 72 de los 141 globos (Leveson, 1986). Muchos otros ejemplos como éste han sido documentados, y es razonable pensar que muchos más no lo fueron. En 1986, Hecht y Hecht (1986b) estudiaron programas grandes, y concluyeron que, normalmente, por cada millón de líneas de código se introducían 20.000 imperfecciones en el software. Normalmente, el 90 por ciento de éstas se encontraban mediante pruebas. Durante el primer año de funcionamiento, podían salir a la luz unos 200 defectos, mientras que otros 1.800 permanecían sin detectar. Las rutinas de mantenimiento normalmente solían arreglar unos 200 agujeros, ¡dejando otros 200 nuevos!

A medida que la sociedad siga confiando el control de sus funciones vitales a los computadores, más necesario será que estos sistemas no fallen. Sin pretender definir de forma precisa (por el momento) qué significa un fallo de un sistema o un defecto, existen, en general, cuatro causas de defectos que pueden propiciar el fallo de un sistema embebido.

(1) Especificación inadecuada. Se ha sugerido que la gran mayoría de los defectos parten de una especificación inadecuada (Leveson, 1986).

(2) Defectos provocados por errores de diseño en los componentes de software.

(3) Defectos provocados por fallos en uno o más componentes del procesador de los sistemas embebidos.

(4) Defectos provocados por una interferencia transitoria o permanente en el subsistema de comunicaciones subyacente.

Estos tres últimos tipos de defectos son los que plagan los lenguajes de programación utilizados en la implementación de los sistemas embebidos. Los errores introducidos por defectos de diseño son, en general, no previstos (en cuanto a sus consecuencias), mientras que los debidos a fallos en el procesador o en la red son, de alguna manera, predecibles. Uno de los principales requisitos, por lo tanto, para cualquier lenguaje de programación de tiempo real es que debe facilitar la construcción de sistemas altamente fiables. En este capítulo se tratarán algunas de las técnicas que pueden ser utilizadas para mejorar la fiabilidad general de los sistemas de computadores embebidos. El Capítulo 6 mostrará cómo se pueden utilizar herramientas de **gestión de excepciones** para ayudar a implementar algunas de estas filosofías de diseño, en particular aquéllas basadas en **tolerancia a fallos**. El tratamiento de los problemas debidos a fallos del procesador o de las comunicaciones se estudia en el Capítulo 14.

5.1 Fiabilidad, fallos y defectos

Antes de proseguir, se hacen necesarias definiciones más precisas de fiabilidad, fallo y defecto. Randell et al. (1978) define la **fiabilidad** de un sistema como

> ... una medida del éxito con el que el sistema se ajusta a alguna especificación definitiva de su comportamiento.

De manera ideal, esta especificación debería ser completa, consistente, comprensible y no ambigua. También debería tenerse en cuenta que los *tiempos de respuesta* son una parte importante de la especificación, aunque la discusión del cumplimiento de los tiempos límite se pospone hasta el Capítulo 13. La anterior definición de fiabilidad también se puede utilizar para definir un **fallo** del sistema. Recurriendo de nuevo a las palabras de Randell et al.:

> Cuando el comportamiento de un sistema se desvía del especificado para él, se dice que es un fallo.

La Sección 5.9 tratará las medidas de fiabilidad, pero, mientras tanto, *altamente fiable* se considerará como sinónimo de *tasa baja de fallos*.

El atento lector se habrá dado cuenta de que nuestras definiciones, al final, están relacionadas con el *comportamiento* de un sistema, esto es, con su apariencia *externa*. Los fallos son el resultado de problemas internos no esperados que el sistema manifiesta eventualmente en su comportamiento externo. Estos problemas se llaman **errores**, y sus causas mecánicas o algorítmicas se denominan **defectos**. Un componente defectuoso de un sistema es, por lo tanto, un componente que producirá un error bajo un conjunto concreto de circunstancias durante la vida del sistema. Visto en términos de transición de estados, un sistema puede ser considerado como un número de estados *externos* e *internos*. Un estado externo no especificado en el comportamiento del sistema se considerará un fallo del sistema. El sistema en sí mismo consta de un número de componentes (cada uno con sus propios estados), contribuyendo todos ellos al comportamiento externo del sistema. La combinación de los estados de estos componentes se denomina estado interno del sistema. Un estado interno no especificado se considera un error, y el componente que produjo la transición de estados ilegal se dice que es defectuoso.

Por supuesto, un sistema habitualmente está compuesto de componentes, cada uno de los cuales se puede considerar como un sistema en sí mismo. Por lo tanto, un fallo en un sistema puede inducir un defecto en otro, el cual puede acabar en un error y en un fallo potencial de ese sistema. Esto puede continuar y producir un defecto en cualquier sistema relacionado, y así sucesivamente (según se ilustra en la Figura 5.1).

Se pueden distinguir tres tipos de fallos.

(1) **Fallos transitorios** Un fallo transitorio comienza en un instante de tiempo concreto, se mantiene en el sistema durante algún periodo, y luego desaparece. Ejemplos de este tipo de fallos se dan en componentes hardware en los que se produce una reacción adversa a una interferencia externa, como la producida por un campo eléctrico o por radiactividad. Después de que la perturbación desaparece, lo hace también el fallo (aunque no necesariamente el error inducido). Muchos de los fallos de los sistemas de comunicación son transitorios.

(2) **Fallos permanentes** Los fallos permanentes comienzan en un instante determinado y permanecen en el sistema hasta que son reparados; es el caso, por ejemplo, de un cable roto o de un error de diseño del software.

(3) **Fallos intermitentes** Son fallos transitorios que ocurren de vez en cuando. Un ejemplo es un componente hardware sensible al calor, que funciona durante un rato, deja de funcionar, se enfría, y entonces comienza a funcionar de nuevo.

Para crear sistemas fiables, se deben tener en cuenta todos estos tipos de fallos, de forma que se pueda evitar que causen un comportamiento erróneo del sistema (esto es, un fallo). La dificultad que esto conlleva se agrava por la utilización indirecta de los computadores en la construcción de

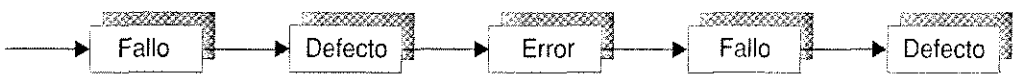


Figura 5.1. Fallo, error, defecto, y fallos encadenados.

sistemas en los que el aspecto de la seguridad es crítico. Por ejemplo, en 1979 fue descubierto un error en un programa utilizado en el diseño de reactores nucleares y de sus sistemas de refrigeración. El defecto producido en el diseño del reactor no se detectó durante las pruebas de instalación, y estaba relacionado con la resistencia estructural de los soportes de las tuberías y las válvulas. El programa supuestamente había garantizado el cumplimiento de los estándares de seguridad contra terremotos de los reactores en funcionamiento. El descubrimiento del agujero provocó que tuvieran que ser apagadas cinco plantas nucleares (Leveson, 1986).

5.2 Modos de fallo

Un sistema puede fallar de varias maneras diferentes. Un diseñador que utiliza un sistema *X* para implementar un sistema *Y*, usualmente realiza alguna suposición sobre los modos de fallo esperados de *X*. Si *X* falla de forma diferente a lo esperado, entonces el sistema *Y* puede fallar como consecuencia de ello.

Un sistema proporciona servicios. Resulta posible, por lo tanto, clasificar los modos de fallo de un sistema de acuerdo con el impacto que tendrán en los servicios que se proporciona. Se pueden identificar dos dominios generales de modos de fallo:

- Fallos de valor: el valor asociado con el servicio es erróneo.
- Fallos de tiempo: el servicio se completa a destiempo.

Las combinaciones de fallos de valor y de tiempo normalmente se denominan fallos **arbitrarios**.

En general, un error de valor podría estar aún dentro del rango correcto de valores, o podría encontrarse fuera del rango esperado para ese servicio. Esto último equivale a un error de escritura en los lenguajes de programación, y se denomina **error de límites**. Normalmente, estos tipos de fallos se reconocen fácilmente.

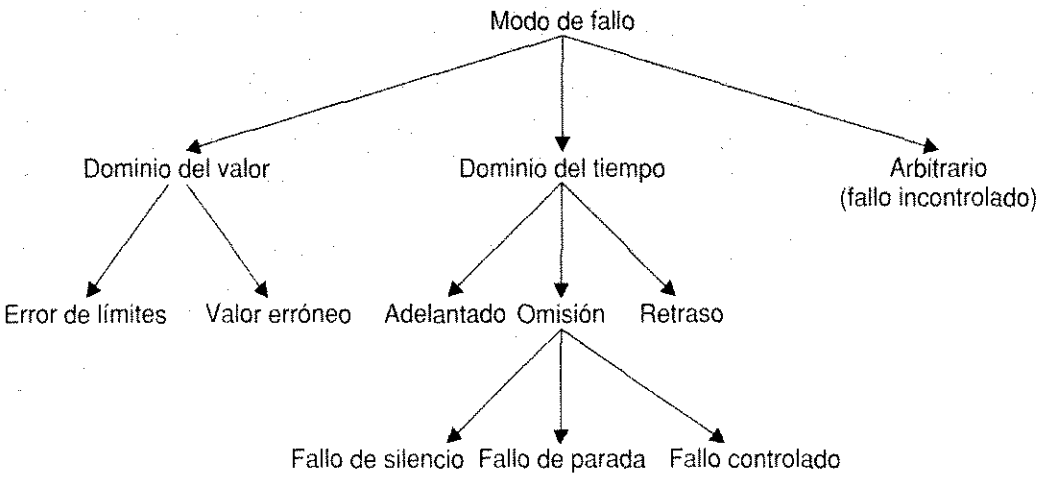


Figura 5.2. Clasificación de los modos de fallo.

Los fallos en el dominio del tiempo pueden hacer que el servicio sea entregado:

- Demasiado pronto: el servicio es entregado antes de lo requerido.
- Demasiado tarde: el servicio se entrega después de lo requerido (en estos casos se suele hablar de **error de prestaciones**).
- Infinitamente tarde: el servicio nunca es entregado (en estos casos se suele hablar de un **fallo de omisión**).

Aún se puede identificar un modo de fallo más, que se produce cuando un servicio es entregado sin ser esperado. A este tipo de fallo habitualmente se le denomina fallo de **encargo** o **improvisación**. Por supuesto que, a menudo, resulta difícil distinguir entre un fallo en el dominio del valor y del tiempo de un fallo de encargo seguido de un fallo de omisión. La Figura 5.2 ilustra la clasificación de los modos de fallo.

Dada la clasificación anterior de los modos de fallo, se pueden realizar algunas suposiciones sobre el modo en que un sistema puede fallar:

- **Fallo descontrolado**: un sistema que produce errores arbitrarios, tanto en el domino del valor como en el del tiempo (incluyendo errores de improvisación).
- **Fallo de retraso**: un sistema que produce servicios correctos en el dominio del valor, pero que sufre errores de retraso en el tiempo.
- **Fallo de silencio**: un sistema que produce servicios correctos tanto en el dominio del valor como en el del tiempo, hasta que falla. El único fallo posible es un fallo de omisión, y cuando ocurre todos los servicios siguientes también sufrirán fallos de omisión.
- **Fallo de parada**: un sistema que tiene todas las propiedades de un fallo silencioso, pero que permite que otros sistemas puedan detectar que ha entrado en el estado de fallo de silencio.
- **Fallo controlado**: un sistema que falla de una forma especificada y controlada.
- **Sin fallos**: un sistema que siempre produce los servicios correctos, tanto en el dominio del valor como en el del tiempo.

Se pueden realizar más suposiciones, pero la lista anterior resulta suficiente para este libro.

Prevención de fallos y tolerancia a fallos

Se pueden considerar dos formas diferentes de ayudar a los diseñadores a mejorar la fiabilidad de sus sistemas (Anderson y Lee, 1990). La primera es conocida como **prevención de fallos**, y se refiere al intento de impedir que cualquier posibilidad de fallo se cuele en el sistema antes de que esté operativo. La segunda se denomina **tolerancia a fallos**, y hace posible que el sistema

continúe funcionando incluso ante la presencia de fallos. Ambas aproximaciones intentan producir sistemas con modos de fallo bien definidos.

5.3.1 Prevención de fallos

Existes dos fases en la prevención de fallos: **evitación** y **eliminación**.

Con la evitación se intenta limitar la introducción de componentes potencialmente defectuosos durante la construcción del sistema. En lo que respecta al hardware, esto puede implicar (Randell et al., 1978):

- La utilización de los componentes más fiables dentro de las restricciones de coste y prestaciones dadas.
- La utilización de técnicas exhaustivamente refinadas para la interconexión de componentes y el ensamblado de subsistemas.
- El aislamiento del hardware para protegerlo de formas de interferencia esperadas.

Los componentes software de los grandes sistemas embebidos actualmente son mucho más complejos que sus correspondientes elementos hardware. Aunque el software no se deteriora con el uso, resulta en cualquier caso virtualmente imposible escribir programas libres de fallos. Esto ya se mencionó en los Capítulos 2 y 4; sin embargo, la calidad del software se puede mejorar mediante:

- Especificaciones de requisitos rigurosas, si no formales.
- Utilización de probadas metodologías de diseño.
- Utilización de lenguajes que faciliten la abstracción de datos y la modularidad.
- Uso de herramientas de ingeniería de software para ayudar en la manipulación de los componentes software y en la gestión de la complejidad.

A pesar de utilizar técnicas para evitar fallos, éstos se encontrarán inevitablemente en el sistema una vez construido. En concreto, pueden existir errores de diseño tanto en los componentes de software como en los de hardware. La segunda fase de la prevención de fallos, por lo tanto, es la *eliminación de fallos*. Esto consiste normalmente en procedimientos para encontrar y eliminar las causas de los errores. Aunque se pueden utilizar técnicas como los revisores de diseño, la verificación de programas y las inspecciones del código, normalmente se pone más énfasis en la prueba del sistema. Desafortunadamente, la prueba del sistema nunca puede ser exhaustiva y eliminar todos los potenciales fallos. En concreto, existen los siguientes problemas:

- Una prueba sólo se puede utilizar para demostrar la presencia de fallos, no su ausencia.
- A menudo resulta imposible realizar pruebas bajo condiciones reales. Una de las mayores causas de preocupación en relación con la Iniciativa Americana de Defensa Estratégica (American Strategic Defense Initiative) es la imposibilidad de probar cualquier sistema de

forma realista excepto bajo condiciones de combate real. La mayoría de las pruebas son realizadas con el sistema en modo de simulación, y es difícil garantizar que la simulación sea precisa. La última prueba nuclear francesa en el Pacífico fue realizada, supuestamente, para recoger datos que hicieran más precisas las simulaciones de las futuras pruebas.

- Los errores que han sido introducidos en la etapa de requisitos del sistema puede que no se manifiesten hasta que el sistema esté operativo. Por ejemplo, en el diseño del avión F18 se realizó una suposición errónea acerca del tiempo que tardaba en liberarse un misil montado en el ala. El problema fue descubierto cuando un misil que no se había separado del lanzador en el momento de su ignición produjo una violenta pérdida de control de la aeronave (Leveson, 1986).

A pesar de todas las técnicas de prueba y verificación, como los componentes hardware pueden fallar, la aproximación basada en la prevención de fallos resultará inapropiada cuando la frecuencia o la duración de los tiempos de reparación resulten inaceptables, o cuando no se pueda acceder al sistema para actividades de mantenimiento y reparación. Un ejemplo extremo de esto último es la sonda no tripulada Voyager.

5.3.2 Tolerancia a fallos

Debido a las inevitables limitaciones de la prevención de fallos, los diseñadores de sistemas embebidos deben considerar recurrir a la tolerancia a fallos. Por supuesto que esto no significa que deba abandonarse todo esfuerzo para evitar encontrarse con sistemas operativos defectuosos. Sin embargo, este libro se centra más en la tolerancia a fallos que en la prevención de fallos.

Un sistema puede proporcionar diferentes niveles de tolerancia a fallos

- **Tolerancia total frente a fallos**: el sistema continua en funcionamiento en presencia de fallos, aunque por un periodo limitado, sin una pérdida significativa de funcionalidad o prestaciones.

- **Degradación controlada** (o caída suave): el sistema continua en operación en presencia de errores, aceptándose una degradación parcial de la funcionalidad o de las prestaciones durante la recuperación o la reparación.

- **Fallo seguro**: el sistema cuida de su integridad durante el fallo aceptando una parada temporal de su funcionamiento.

El nivel de tolerancia a fallos requerido dependerá de la aplicación. Aunque, al menos teóricamente, los sistemas más críticos respecto a la seguridad exigirán una tolerancia total frente a fallos, en la práctica muchos se conformarán con una degradación controlada. En concreto, aquellos sistemas que puedan sufrir un daño físico, como los aviones de combate, pueden proporcionar varios grados de degradación controlada. También se hace necesaria la degradación controlada en aquellas aplicaciones altamente complejas que tengan que funcionar de forma continua (con requisitos de alta disponibilidad), ya que la tolerancia total frente a fallos no es alcanzable para periodos indefinidos. Por ejemplo, el sistema de automatización avanzada de la

administración de aviación federal de Estados Unidos, que proporciona servicios automatizados de control del tráfico tanto en ruta como en la terminal, tiene tres niveles de degradación controlada para los acoplamientos de los computadores de control de área (Avizienis y Ball, 1987). Esto se ilustra en la Figura 5.3.

En algunas situaciones, simplemente resulta necesario apagar el sistema en un estado seguro. Estos sistemas seguros frente a fallos intentan limitar el alcance del daño causado por un fallo. Por ejemplo, cuando el computador de control de los variadores del ángulo de ataque y salida del ala (*slats* y *flaps*, respectivamente) de un Airbus A310 detecta un error en el aterrizaje, lleva el sistema a un estado seguro y luego se apaga. En esta situación, un estado seguro es aquél en el que ambas alas tienen la misma configuración; sólo las configuraciones asimétricas son peligrosas en el aterrizaje (Martin, 1982).

Las primeros acercamientos al diseño de sistemas tolerantes a fallos realizaban tres suposiciones:

(1) Los algoritmos del sistema han sido diseñados correctamente.

(2) Se conocen todos los posibles modos de fallos de los componentes.

(3) Se han tenido en cuenta todas las posibles interacciones entre el sistema y el entorno.

Sin embargo, la creciente complejidad del software y la introducción de componentes hardware VLSI han hecho que no se puedan seguir haciendo esas suposiciones (en el caso de que alguna

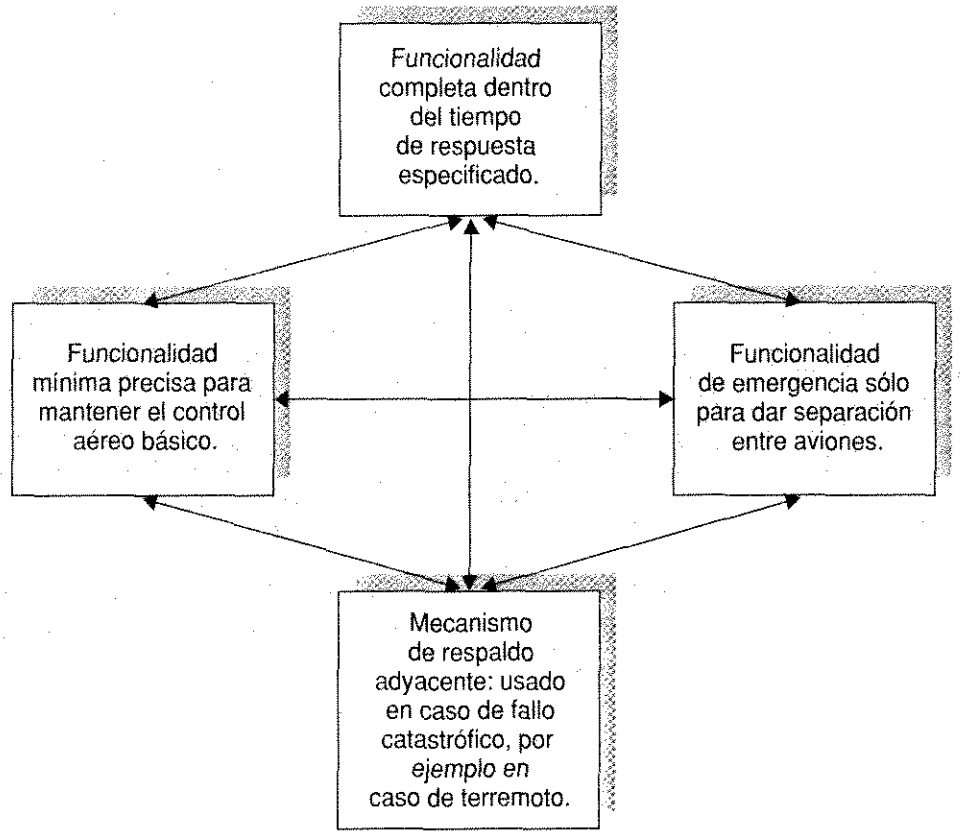


Figura 5.3. Degradación controlada y recuperación en el sistema de control del tráfico aéreo.

vez se hayan podido hacer). Consecuentemente, se debe considerar tanto los fallos previstos como los imprevistos, incluyendo los fallos de diseño software y hardware.

5.3.3 Redundancia

Todas las técnicas utilizadas para conseguir tolerancia a fallos se basan en añadir elementos extra al sistema para que detecte y se recupere de los fallos. Estos componentes son redundantes, en el sentido de que no son necesarios para el normal funcionamiento del sistema. Esto se llama a menudo **redundancia protectora**. La intención última de la tolerancia a fallos es minimizar la redundancia maximizando la fiabilidad proporcionada, siempre bajo las restricciones de coste y tamaño del sistema. Se debe tener cuidado al estructurar los sistemas tolerantes a fallos, ya que los distintos componentes incrementan inevitablemente la complejidad de todo el sistema. Esto, en sí mismo, puede conducir a sistemas menos fiables. Por ejemplo, el primer lanzamiento de la lanzadera espacial fue abortado debido a un problema en la sincronización en los sistemas de computadores replicados (Garman, 1981). Para ayudar a reducir los problemas asociados con la interacción entre los componentes redundantes, resulta aconsejable separar los componentes tolerantes a fallos del resto del sistema.

Existen varias clasificaciones diferentes de redundancia, dependiendo de los componentes considerados y de la terminología utilizada. La tolerancia a fallos de software es el principal tema de este capítulo, y por lo tanto sólo se realizarán referencias de pasada a las técnicas de redundancia hardware. En el caso del hardware, Anderson y Lee (1990) distinguen entre redundancia *estática* (o enmascarada) y *dinámica*. En la redundancia estática, los componentes redundantes son utilizados dentro de un sistema (o subsistema) para ocultar los efectos de los fallos. Un ejemplo de redundancia estática es la *redundancia triple modular* (TMR; Triple Modular Redundancy). La TMR consiste en tres subcomponentes idénticos y en circuitos de *votación por mayoría*. Los circuitos comparan la salida de todos los componentes, y si alguna difiere de las otras dos es bloqueada. La suposición hecha aquí es que el fallo no se debe a un aspecto común de los subcomponentes (como un error de diseño), sino a un error transitorio o al deterioro de algún componente. Resulta evidente que se necesita más redundancia para enmascarar fallos de más de un componente. Para denominar a este enfoque se utiliza el término general de *redundancia N modular* (N Modular Redundancy; NMR).

La redundancia dinámica es la redundancia aportada dentro de un componente que hace que el mismo indique, explícita o implícitamente, que la salida es errónea. De este modo, se proporciona la capacidad de *detección de errores*, más que la de enmascaramiento del error; la recuperación debe ser proporcionada por otro componente. Las sumas de comprobación (checksums) en las transmisiones de comunicaciones y las paridades de bits en las memorias son ejemplos de redundancia dinámica.

Se pueden identificar dos enfoques distintos en la búsqueda de tolerancia a fallos debidos a errores de diseño. El primero es análogo a la redundancia enmascarada del hardware, y se llama programación de *N*-versiones. El segundo se basa en la detección y la recuperación de errores, y es análogo a la redundancia dinámica, en el sentido de que se activan los procedimientos de recuperación después de haberse detectado un error. El tema de la tolerancia a fallos hardware será tratado de nuevo en la Sección 14.5, donde se utilizarán técnicas software.

5.4 Programación de *N*-versiones

El éxito de TMR y NMR en hardware han motivado un enfoque similar en el tratamiento de la tolerancia a fallos software. Sin embargo, como el software no se deteriora con el uso, se utilizan para la detección de fallos de diseño. De hecho, este enfoque (que se conoce como programación de *N*-versiones) fue recomendado originalmente por Babbage en 1837 (Randell, 1982):

> Cuando la fórmula resulta muy complicada, puede ser algebraicamente dispuesta para la computación de dos o más formas distintas, con lo que se pueden conseguir dos o más conjuntos de tarjetas. Si se emplean las mismas constantes con cada conjunto, y si bajo estas circunstancias coinciden los resultados, podremos estar bastantes seguros de la precisión de todos ellos.

La programación de *N*-versiones se define como la generación independiente de *N* programas funcionalmente equivalentes (con *N* mayor o igual a 2) a partir de la misma especificación inicial (Chen y Avizienis, 1978). La generación independiente de *N* programas significa que *N* individuos o grupos producen las *N* versiones necesarias del software *sin interacción* (por esta razón la programación de *N*-versiones se denomina frecuentemente **diversidad en el diseño**). Una vez diseñados y escritos, los programas se ejecutan de forma concurrente con las mismas entradas, y un **programa director** compara sus resultados. En principio, los resultados deberían ser idénticos, pero en la práctica existirán algunas diferencias, en cuyo caso se toma como correcto, por consenso, uno de ellos, suponiendo que exista alguno.

La programación de *N*-versiones se basa en la suposición de que se puede especificar completamente un programa de forma consistente y sin ambigüedad, y que los programas que han sido desarrollados independientemente fallarán de forma independiente. Esto es, que no existe relación entre los fallos de una versión y los fallos de otra. Esta suposición puede no resultar válida si cada versión ha sido escrita en el mismo lenguaje de programación, ya que los errores asociados con la implementación del lenguaje son comunes a las distintas versiones. Consecuentemente, se debe utilizar lenguajes y entornos de programación diferentes. Si se utiliza el mismo lenguaje, los compiladores y los entornos utilizados deberían ser de distintos fabricantes. En cualquier caso, para protegernos de fallos físicos, las *N*-versiones deben ser distribuidas entre diferentes máquinas con líneas de comunicación tolerantes a fallos. Para el sistema de control de vuelo del Boeing 777, se escribió un único programa en Ada, pero para obtener diversidad se utilizaron tres procesadores diferentes y tres compiladores distintos.

El programa con *N*-versiones es controlado por un proceso director el cual es responsable de:

- Invocar a cada una de las versiones.
- Esperar a que las versiones realicen su trabajo.
- Comparar los resultados y actuar basándose en los mismos.

Anteriormente se ha asumido de forma implícita que los programas o procesos deben acabar su ejecución antes de proceder a la comparación de sus resultados, pero a menudo éste no es el caso en los sistemas embebidos, ya que los procesos nunca finalizan. El director y las *N* versiones deben, por lo tanto, comunicarse durante el transcurso de sus ejecuciones.

Se concluye que las versiones, aunque independientes, deben interactuar con el programa director. Esta interacción se especifica en los requisitos de las versiones, y se compone de tres elementos (Chen y Avizienis, 1978):

(1) Vectores de comparación.

(2) Indicadores de estatus de la comparación.

(3) Puntos de comparación.

La forma en la que las versiones se comunican y se sincronizan con el director dependerá del lenguaje de programación utilizado y de su modelo de concurrencia (véanse los Capítulos 7, 8 y 9). Si se utilizan lenguajes diferentes en las diferentes versiones, entonces normalmente los medios de comunicación y de sincronización serán proporcionados por un sistema operativo de tiempo real. El diagrama de la Figura 5.4 muestra la relación entre las N versiones y el director, con $N = 3$.

Los vectores de comparación son las estructuras de datos que representan las salidas (o los votos) producidos por las versiones, además de cualquier atributo asociado con su cálculo; éstos deben ser comparados por el director. Por ejemplo, en un sistema de control de una aeronave, si los valores que se están comparando son las posiciones de la aeronave, un atributo puede indicar si los valores son el resultado de una lectura de radar reciente o han sido calculados a partir de lecturas anteriores. Los indicadores de estatus de la comparación son comunicados por el director a las versiones, e indican las acciones que cada versión debe ejecutar como resultado de la comparación realizada por el director. Tales acciones dependerán del resultado de la comparación: si los votos coinciden o si han sido entregados a tiempo. Los posibles resultados son:

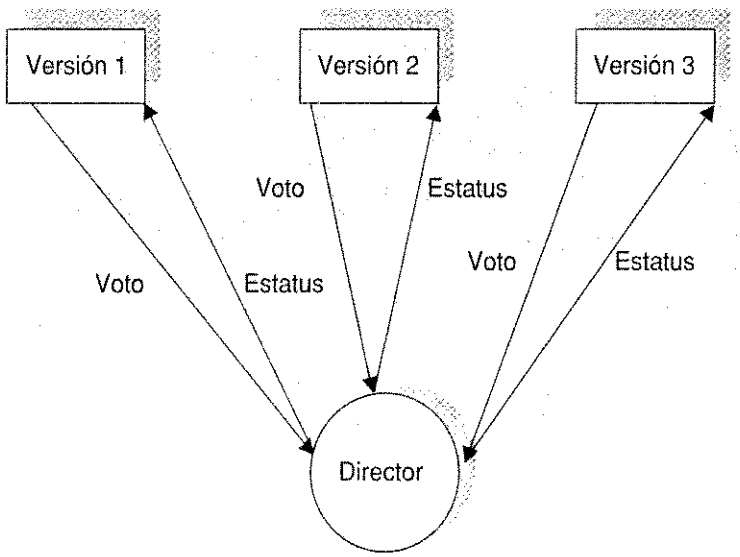


Figura 5.4. Programación de *N*-versiones.

- Continuación.
- Terminación de una o más versiones.
- Continuación después de modificar uno o más votos respecto al valor de la mayoría.

Los puntos de comparación son puntos dentro de las versiones donde deben comunicar sus votos al proceso director. Como Hecht y Hecht (1986a) destacan, la frecuencia con la cual se realizan las comparaciones es una decisión de diseño importante. Ésta es la **granularidad** de la tolerancia a fallos proporcionada. La tolerancia a fallos de gran granularidad (esto es, con comparaciones infrecuentes) minimizará la penalización de las prestaciones inherente a las estrategias de comparación, y permitirá una gran independencia en el diseño de las versiones. Sin embargo, una granularidad grande probablemente producirá una gran divergencia en los resultados obtenidos, debido al gran número de pasos ejecutados entre comparaciones. Los problemas de comparación de votos, o votación (como se suele denominar), se considerarán en la siguiente subsección. La tolerancia a fallos de granularidad pequeña requiere una semejanza en los detalles de la estructura de los programas, y por lo tanto reduce el grado de independencia entre las versiones. Un número frecuente de comparaciones también incrementa la sobrecarga asociada con esta técnica.

5.4.1 Comparación de votos

En la programación de *N*-versiones resulta crucial la eficiencia y la facilidad con la que el programa director compara los votos y decide cuándo existe un desacuerdo. En aquellas aplicaciones en las que se manipula texto o se realizan operaciones aritméticas de enteros, normalmente existirá un único resultado correcto. El director puede comparar fácilmente los votos de las distintas versiones y elegir la decisión mayoritaria.

Desafortunadamente, no todos los resultados son de una naturaleza exacta. En concreto, en aquellos votos que requieren cálculos con números reales, será difícil que diferentes versiones produzcan exactamente el mismo resultado. Esto podría deberse a la inexactitud de la representación hardware de los números reales, o a la sensibilidad de los datos respecto a un algoritmo concreto. Las técnicas utilizadas en la comparación de estos tipos de resultados se denominan **votación inexacta**. Una técnica simple es acabar comprobando un rango utilizando una estimación previa o un valor medio de los *N* resultados. Sin embargo, puede que sea difícil encontrar un enfoque general para la votación inexacta.

Otra dificultad asociada a la aritmética de precisión finita es el llamado **problema de la comparación consistente** (Brilliant et al., 1987). Este problema se da cuando una aplicación tiene que realizar una comparación basada en un valor finito dado en la especificación; el resultado de la comparación determina entonces el curso de la acción. Como ejemplo, considérese un sistema de control de procesos que monitoriza los sensores de temperatura y de presión y toma las decisiones apropiadas de acuerdo con sus valores para asegurar la integridad del sistema. Supóngase que cuando cualquiera de las lecturas sobrepasa un valor umbral se debe tomar algún tipo de acción correctora. Considérese ahora un sistema software de 3-versiones (V_1, V_2, V_3), cada una de las cuales debe leer ambos sensores, decidir alguna acción, y entonces votar el resultado (no existe comunicación entre las versiones hasta que votan). Como resultado de una aritmética

de precisión finita, cada versión calculará diferentes valores (digamos T_1, T_2, T_3 para el sensor de temperatura, y P_1, P_2, P_3 para el sensor de presión). Suponiendo que el valor umbral para la temperatura es T_u y para la presión P_u, el problema de comparación consistente se da cuando ambas lecturas están alrededor de sus valores umbral.

La situación se podría producir cuando T_1 y T_2 están justo por debajo de T_u, y T_3 justo por encima, de modo que V_1 y V_2 seguirán con su ejecución normal, mientras que V_3 tomará alguna acción correctora. Ahora bien, si las versiones V_1 y V_2 llegan a otro punto de comparación, esta vez con el sensor de presión, resulta posible que P_1 se encuentre justo por debajo de P_u, mientras que P_2 estará justo por encima. El resultado final será que las tres versiones tomarán caminos de ejecución diferentes, cada uno de los cuales será válido. El proceso está representado mediante un diagrama en la Figura 5.5.

En un primer acercamiento, parecería apropiado utilizar técnicas de comparación inexacta y asumir que los valores son iguales si difieren en un Δ de tolerancia, pero como Brilliant et al. (1987) destacan, el problema reaparecería cuando los valores estuvieran cerca del valor umbral $\pm\Delta$.

Todavía existen más problemas con la comparación de votos cuando existen múltiples soluciones naturales a un mismo problema. Por ejemplo, una ecuación cuadrática puede tener más de una solución. De nuevo el desacuerdo es posible, incluso cuando no se haya producido fallo alguno (Anderson y Lee, 1990).

5.4.2 Aspectos principales de la programación de *N*-versiones

Se ha comentado que el éxito de la programación de *N*-versiones depende de varios aspectos, los cuales vamos a repasar a continuación.

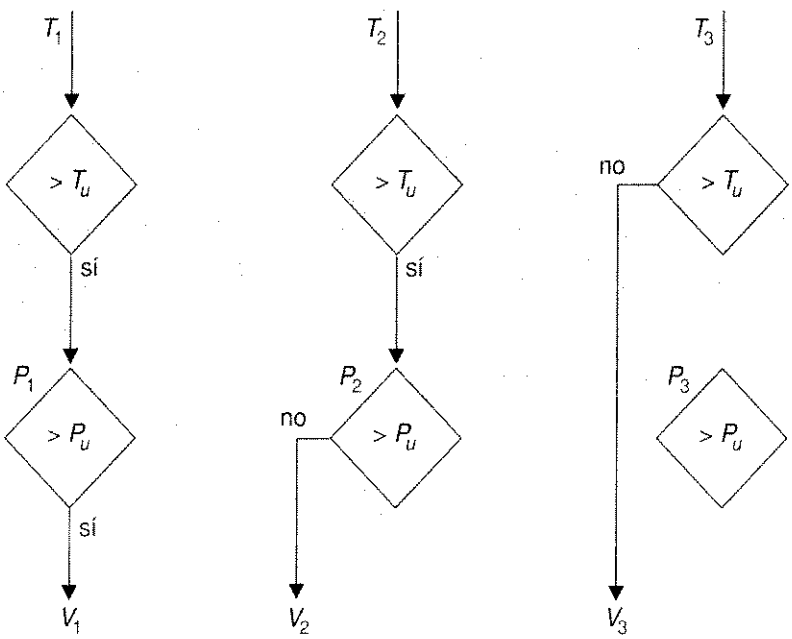


Figura 5.5. El problema de la comparación consistente con tres versiones.

(1) **Especificación inicial**: se ha sugerido que la gran mayoría de los fallos en el software provienen de una especificación inadecuada (Leveson, 1986). Las técnicas actuales están lejos de producir especificaciones completas, consistentes, comprensibles y no ambiguas, aunque los métodos de especificación formal proporcionan una fructífera línea de investigación. Resulta evidente que un error de especificación se manifestará en todas las *N* versiones de la implementación.

(2) **Independencia en el diseño**: se han realizado algunos experimentos (Knight et al., 1985; Avizienis et al., 1988; Brilliant et al., 1990; Eckhardt et al., 1991; Hatton, 1997) para comprobar la hipótesis de que el software producido independientemente mostrará fallos distintos. Sin embargo, se obtienen resultados conflictivos. Knight et al. (1985) han demostrado que para un problema particular con una especificación concienzudamente refinada, la hipótesis tuvo que ser rechazada con un nivel de confianza del 99 por ciento. Por contra, Avizienis et al. (1988) afirmaron que era muy raro encontrar fallos idénticos en dos versiones de un sistema de seis versiones. Comparando sus resultados y los obtenidos por Knight et al., concluyeron que el problema estudiado por Knight et al. tenía un potencial de diversidad limitado, y que los test de aceptación eran totalmente inadecuados de acuerdo con los estándares industriales habituales. Avizienis et al. afirmaron que la aplicación rigurosa del paradigma de la programación de *N*-versiones habría conducido a la eliminación de todos los errores encontrados por Knight et al. antes de la aceptación del sistema. Sin embargo, existe la certeza de que allí donde la especificación resulte compleja, inevitablemente se producirá una distorsión de la comprensión de los requisitos entre los equipos independientes. Si estos requisitos también se refieren a datos de entrada poco frecuentes, entonces los errores de diseño comunes pueden no ser capturados en la fase de pruebas. En años más recientes, estudios de Hatton (1997) han encontrado que un sistema de tres versiones es, a pesar de todo, entre 5 y 9 veces más fiable que un sistema de alta calidad de versión única.

(3) **Presupuesto adecuado**: con la mayoría de los sistemas embebidos, el principal coste se debe al software. Un sistema de tres versiones casi triplicará el presupuesto, y causará problemas de mantenimiento. En un entorno competitivo, es raro que un cliente potencial proponga la técnica de la programación de *N*-versiones, a menos que sea obligatorio. Además, no está claro que no se pueda producir un sistema más fiable si los recursos potencialmente disponibles para construir las *N* versiones fueran utilizados para construir una única versión.

También se ha demostrado que en algunos casos es difícil encontrar algoritmos de votación inexacta, y que a menos que se tenga cuidado con el problema de la comparación consistente, los votos pueden diferir incluso en ausencia de fallos.

Aunque la programación de *N*-versiones puede jugar un papel en la producción de software fiable, debería ser utilizada con cuidado junto con otras técnicas, como por ejemplo las que se ven a continuación.

5.5 Redundancia de software dinámica

La programación de *N*-versiones es el equivalente software a la redundancia estática o enmascarada, en la que los fallos dentro de un componente son ocultados respecto al exterior. Esto es estático, ya que cada versión del software tiene una relación fija con cada versión y con el director, además de que siempre entra en funcionamiento, se hayan producido fallos o no. Con la redundancia **dinámica**, los componentes redundantes sólo entran en juego *cuando* se ha detectado un error.

Esta técnica de tolerancia a fallos tiene cuatro fases constituyentes (Anderson y Lee, 1990).

(1) **Detección de errores**: la mayoría de fallos se manifestarán bajo la forma de un error, de modo que no se utilizará ningún esquema de tolerancia a fallos hasta que se haya detectado un error.

(2) **Confinamiento y valoración de daños**: cuando se detecte un error, debe estimarse la extensión del sistema que ha sido corrompida (esto se suele denominar *diagnosis de errores*); el intervalo de tiempo entre el fallo y la manifestación del error asociado es debido a que la información errónea tiene que difundirse a través del sistema.

(3) **Recuperación del error**: éste es uno de los aspectos más importantes de la tolerancia a fallos. Las técnicas de recuperación de errores deberían dirigir al sistema corrupto a un estado a partir del cual pueda continuar con su normal funcionamiento (quizás con una degradación funcional).

(4) **Tratamiento del fallo y continuación del servicio**: un error es un síntoma de un fallo; aunque el daño pudiera haber sido reparado, el fallo continua existiendo, y por lo tanto el error puede volver a darse a menos que se realice algún tipo de mantenimiento.

Aunque estas cuatro fases de la tolerancia a fallos se tratan en relación con las técnicas de redundancia dinámica del software, se pueden aplicar a la programación de *N*-versiones. Como Anderson y Lee (1990) han destacado: la detección de errores es realizada por el director, el cual realiza la comprobación de los votos (no se requiere la valoración de daños, ya que las versiones son independientes); la recuperación de errores implica descartar el resultado erróneo, y el tratamiento del fallo se reduce a ignorar la versión que ha producido el valor erróneo. Sin embargo, si todas las versiones producen votos diferentes, entonces se tiene que llevar a cabo la detección de errores, aunque no existen posibilidades de recuperación. Las siguientes secciones tratan brevemente las fases de la tolerancia a fallos. Para una discusión más profunda, se recomienda al lector consultar Anderson y Lee (1990).

5.5.1 Detección de errores

La efectividad de cualquier sistema tolerante a fallos depende de la efectividad de sus técnicas de detección de errores. Se pueden identificar dos clases de técnicas de detección de errores:

- **Detección en el entorno**. Los errores se detectan en el entorno en el cual se ejecuta el programa. Se incluyen aquellos detectados por el hardware, como los de «ejecución de instrucción ilegal», «desbordamiento aritmético», o «violación de protección». También son considerados los errores detectados en tiempo de ejecución por el sistema soporte del lenguaje de programación de tiempo real; por ejemplo, los de «error en los límites del array», «referencia a apuntador nulo», o «valor fuera de rango». Estos tipos de error se verán en el Capítulo 6 dentro del contexto de los lenguajes de programación Ada y Java.

- **Detección en la aplicación**. Los errores se detectan por la aplicación misma. La mayoría de las técnicas que se pueden utilizar en la aplicación corresponden a alguna de las siguientes categorías.

  - **Comprobación de réplicas**. Se ha demostrado que la programación de *N*-versiones puede ser utilizada para tolerar fallos software, y también como técnica para la detección de errores (utilizando una redundancia de 2-versiones).

  - **Comprobaciones temporales**. Existen dos tipos de comprobaciones temporales. El primer tipo implica un proceso **temporizador guardián**, que si no es puesto a cero por un cierto componente dentro de un cierto periodo de tiempo, se supone que dicho componente está en un estado de error. El componente software debe poner a cero el temporizador de forma continua para indicar que está funcionando correctamente. En los sistemas embebidos, donde los tiempos de respuesta son importantes, se necesita un segundo tipo de comprobación. De esta manera se detectan fallos asociados con el incumplimiento de tiempos límite (deadlines). Cuando el sistema de soporte de ejecución planifique tiempos límite, la detección de los fallos asociados con el incumplimiento de esos límites se puede considerar como parte del entorno. (Algunos de los problemas asociados con la planificación de los tiempos límite se tratarán en el Capítulo 13.)

    Por supuesto que las comprobaciones temporales *no* aseguran que un componente esté funcionando correctamente, sino sólo que está funcionando a tiempo. Las comprobaciones temporales deberían ser utilizadas, por lo tanto, junto con otras técnicas de detección de errores.

  - **Comprobaciones inversas**. Éstas son posibles en componentes donde exista una relación uno a uno (isomórfica) entre la entrada y la salida. Una comprobación de este tipo toma la salida, calcula la entrada correspondiente, y la compara con el valor de la entrada real. Por ejemplo, en un componente que calcule la raíz cuadrada de un número, la comprobación inversa es simplemente el cálculo del cuadrado de la salida y la comparación de éste con la entrada. (Hay que destacar que en el caso de los número reales se tendrán que utilizar técnicas de comparación inexacta.)

  - **Códigos de comprobación**. Los códigos de comprobación se utilizan para comprobar la corrupción de los datos. Se basan en la redundancia de la información contenida en los datos. Por ejemplo, se puede calcular un valor (checksum) y enviarlo con los datos para su transmisión a través de una red de comunicación. Cuando se reciben los datos, el valor es recalculado y comparado con el *checksum* recibido.

- **Comprobaciones de racionalidad**. Se basan en el conocimiento del diseño y de la construcción del sistema. Comprueban que el estado de los datos o el valor de un objeto es razonable basándose en su supuesto uso. Normalmente, en los lenguajes de programación modernos la mayoría de la información necesaria para ejecutar estas comprobaciones puede ser aportada por los programadores, como información asociada con los objetos. Por ejemplo, los objetos enteros restringidos a estar comprendidos entre ciertos valores se pueden representar como subtipos de enteros con rangos explícitos. La violación de rango se puede detectar por el sistema de soporte en tiempo de ejecución.

  Algunas veces, las comprobaciones de racionalidad se incluyen en los componentes software, llamándose comúnmente **aserciones**, bajo la forma de expresiones lógicas que al ser evaluadas como verdaderas en tiempo de ejecución indican que no se ha detectado error alguno.

- **Comprobaciones estructurales**. Las comprobaciones estructurales son utilizadas para comprobar la integridad de los objetos de datos tales como listas o colas. Podrían consistir en contar el número de elementos en el objeto, en apuntadores redundantes, o en información extra sobre su estatus.

- **Comprobaciones de racionalidad dinámica**. En la salida producida por algunos controladores digitales, habitualmente existe una relación entre cualesquiera dos salidas consecutivas. Por lo tanto, se podrá detectar un error si el valor de una salida nueva difiere considerablemente del valor de la salida anterior.

Hay que destacar que muchas de las técnicas anteriores se pueden aplicar también en el nivel hardware, y que por lo tanto pueden ser consideradas dentro de los «errores en el entorno».

5.5.2 Confinamiento y valoración de los daños

Como puede ocurrir que transcurra un tiempo entre que se produce un defecto y se detecta el error, será necesario valorar cualquier daño que se haya podido producir en ese intervalo de tiempo. Aunque el tipo de error detectado dará alguna idea sobre el daño a la rutina de tratamiento del error, podrían haber sido diseminadas informaciones erróneas por el sistema y por su entorno. Por lo tanto, la valoración de los daños estará estrechamente relacionada con las precauciones que hayan sido tomadas por los diseñadores del sistema para el confinamiento del daño. El confinamiento del daño se refiere a la estructuración del sistema de modo que se minimicen los daños causados por un componente defectuoso. También se conoce como **construcción de cortafuegos**.

Se pueden utilizar dos técnicas para estructurar los sistemas de modo que se facilite el confinamiento del daño: **descomposición modular** y **acciones atómicas**. Los méritos de la descomposición modular fueron descritos en el Capítulo 4. Aquí, se mencionará simplemente que el sistema debería ser descompuesto en componentes, cada uno de los cuales se representa por uno o más módulos. La interacción entre los componentes se produce a través de interfaces bien definidas, y los detalles internos de los módulos están ocultos y no son accesibles directamente desde el exterior. Esto hace más difícil que un error en un componente pase indiscriminadamente a otro.

La descomposición modular proporciona al sistema software una estructura *estática* que en su mayor parte se pierde en el tiempo de ejecución. Igualmente importante para el confinamiento de los daños es la estructura dinámica del sistema, ya que facilita el razonamiento sobre el comportamiento en el tiempo de ejecución del software. Una técnica de estructuración dinámica importante es la que se basa en el uso de acciones atómicas.

> Se dice que la actividad de un componente es atómica si *no* existen interacciones entre la actividad y el sistema durante el transcurso de la acción.

Esto es, para el resto del sistema una acción atómica aparece como indivisible, y se lleva a cabo de modo *instantáneo*. No se puede pasar información desde la acción atómica hacia el resto del sistema y viceversa. Las acciones atómicas a menudo reciben el nombre de **transacciones** o **detransacciones atómicas**. Se utilizan para mover el sistema de un estado consistente a otro, y para restringir el flujo de información entre componentes. Cuando dos o más componentes comparten un recurso, el confinamiento de daños debe implicar restricciones de acceso a ese recurso. La implementación de este aspecto de las acciones atómicas utilizando las primitivas de comunicación y de sincronización presentes en los lenguajes de tiempo real modernos, será considerada en el Capítulo 10.

Otras técnicas encaminadas a restringir el acceso a recursos están basadas en **mecanismos de protección**, algunos de los cuales pueden ser soportados por el hardware. Por ejemplo, cada recurso puede tener uno o más modos de operación, cada uno de ellos con una lista de acceso asociada (por ejemplo, lectura, escritura y ejecución). La actividad de un componente o proceso también tendrá asociada un modo. Cada vez que un proceso accede a un recurso se puede comparar la operación deseada con sus **permisos de acceso**, y, si fuera necesario, se le denegará el acceso.

5.5.3 Recuperación de errores

Una vez que ha sido detectada una situación de error y el daño ha sido valorado, se deben iniciar los procedimientos de recuperación de errores. Probablemente, ésta es la fase más importante de cualquier técnica de tolerancia a fallos. Se debe transformar un estado erróneo del sistema en otro desde el cual el sistema pueda continuar con su funcionamiento normal, aunque quizás con una cierta degradación en el servicio. Respecto a la recuperación de errores, se han propuesto dos estrategias: recuperación **hacia adelante** (forward) y **hacia atrás** (backward).

La recuperación de errores hacia adelante intenta continuar desde el estado erróneo realizando correcciones selectivas en el estado del sistema. En los sistemas embebidos, esto puede implicar proteger cualquier aspecto del entorno controlado que pudiera ser puesto en riesgo o dañado por el fallo. Aunque la recuperación de errores hacia adelante puede resultar eficiente, es específica de cada sistema, y depende de la precisión de las predicciones sobre la localización y la causa del error (esto es, de la valoración de los daños). Ejemplos de técnicas de recuperación hacia adelante son los punteros redundantes en las estructuras de datos, y el uso de códigos autocorrectores, como el Código de Hamming. Durante la acción de recuperación resultará necesario disponer de la posibilidad de abortar (excepción asíncrona) si más de un proceso se encontraba involucrado en la realización del servicio cuando ocurrió el error.

La recuperación hacia atrás se basa en restaurar el sistema a un estado seguro previo a aquél en el que se produjo el error, para luego ejecutar una sección alternativa del programa. Ésta tendrá la misma funcionalidad que la sección que produjo del defecto, pero utilizando un algoritmo diferente. Como en la programación de *N*-versiones, se espera que esta alternativa *no* provoque el mismo defecto de forma recurrente. Al punto al que el proceso es restaurado se le denomina **punto de recuperación** (*checkpoint*), y al proceso de su establecimiento normalmente se le denomina **checkpointing**. Para establecer un punto de recuperación, es necesario salvar en tiempo de ejecución la información apropiada sobre el estado del sistema.

La restauración de estado tiene la ventaja de que el estado erróneo ha sido eliminado y no se tiene que buscar la localización o la causa del defecto. La recuperación hacia atrás de errores se puede usar, por lo tanto, para recuperar defectos no considerados, incluyendo los errores de diseño. Sin embargo, su desventaja es que no puede deshacer ninguno de los efectos que el defecto pudiera haber producido en el entorno del sistema embebido; es difícil deshacer el lanzamiento de un misil, por ejemplo. Además, la recuperación de errores hacia atrás puede consumir tiempo de ejecución, lo cual puede excluir su uso en algunas aplicaciones de tiempo real. Por ejemplo, las operaciones que involucran información de sensores pueden ser dependientes del tiempo, por lo que puede que las técnicas de restauración de estado costosas simplemente no sean factibles. Consecuentemente, para mejorar las prestaciones se han considerado aproximaciones de establecimiento de **puntos de recuperación incrementales** (incremental checkpointing). Un ejemplo de estos sistemas es la **caché de recuperación** (Anderson y Lee, 1990). Otras aproximaciones son los diarios, registros, o logs. En estos casos, el sistema soporte subyacente debe deshacer los efectos del proceso revertiendo las acciones indicadas en el registro.

La restauración no resulta tan simple como se ha descrito cuando se trata de procesos concurrentes que interacciona entre sí. Considérense los dos procesos mostrados en la Figura 5.6. El proceso P_1 establece los puntos de recuperación R_{11}, R_{12} y R_{13}. El proceso P_2 fija los puntos de recuperación R_{21} y R_{22}. Además, los dos procesos se comunican y sincronizan sus acciones a través de IPC_1, IPC_2, IPC_3 e IPC_4. La abreviatura *IPC* se utiliza para indicar una comunicación entre procesos (Interprocess Communication).

Si P_1 detecta un error en T_e, entonces simplemente debe volver atrás hasta el punto de recuperación R_{13}. Sin embargo, consideremos el caso en el que P_2 detecta un error en T_e. Si P_2 es retrotraído hasta R_{22}, entonces se debe deshacer la comunicación IPC_4 con P_1; esto requiere que P_1 sea retrotraído hasta R_{21}. Pero si esto se lleva a cabo, P_2 debe ser llevado hacia atrás hasta R_{21} para deshacer la comunicación IPC_3, y así sucesivamente. El resultado será que ambos procesos serán retrotraídos al comienzo de su interacción conjunta. ¡En la mayoría de los casos, esto puede implicar tener que abortar ambos procesos! Este fenómeno se conoce como **efecto dominó**.

Obviamente, si los dos procesos no interactúan entre sí, entonces no se producirá el efecto dominó. Cuando interactúan más de dos procesos, se incrementa la posibilidad de que se dé el efecto dominó. En este caso, los puntos de recuperación deben ser diseñados consistentemente, de forma que un error detectado en uno de los procesos no produzca que todos los procesos con los que interactúa sean revertidos. En lugar de esto, los procesos pueden ser reiniciados desde un conjunto consistente de puntos de recuperación. Estás **líneas de recuperación**, que es como suelen ser llamadas, están íntimamente relacionadas con la noción de acción atómica. El problema

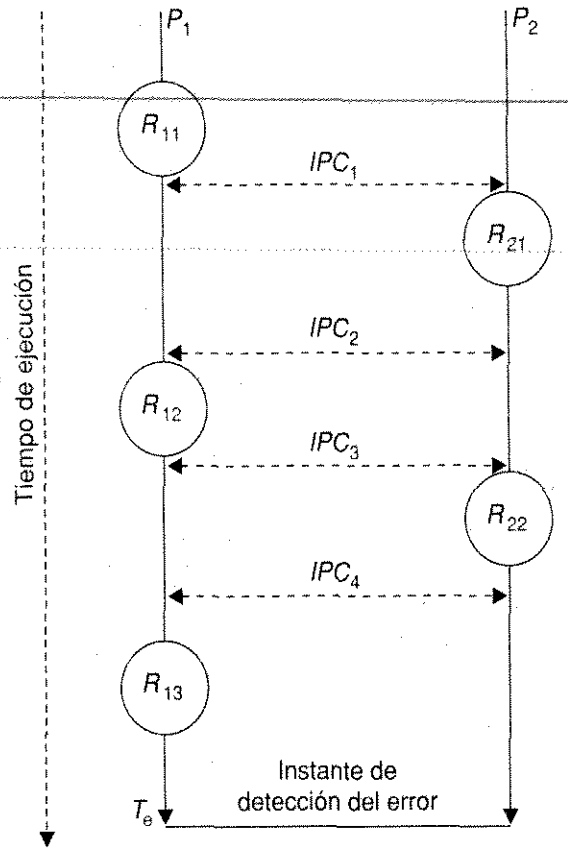


Figura 5.6. El efecto dominó.

de la recuperación de errores en procesos concurrentes será retomado en el Capítulo 10. En lo que resta de capítulo, sólo se considerarán sistemas secuenciales.

Se han presentado los conceptos de recuperación de errores hacia adelante y hacia atrás; cada uno con sus ventajas y desventajas. Los sistemas embebidos no sólo tienen que ser capaces de recuperarse de errores no previstos, sino que tienen que hacerlo en un tiempo finito; por lo que requerirán ambas técnicas de recuperación de errores. En la siguiente sección se considerará la forma en que se plasma la recuperación de errores hacia atrás en los lenguajes experimentales de programación secuencial. En este capítulo no se considerarán más que los mecanismos para la recuperación de errores hacia adelante, ya que resultan difíciles de proveer independientemente de la aplicación. Sin embargo, en el capítulo siguiente se considera la implementación de ambas formas de recuperación de errores dentro del marco común del manejo de excepciones.

5.5.4 Tratamiento de los fallos y servicio continuado

Un error es una manifestación de un defecto, y aunque la fase de recuperación del error puede haber llevado al sistema a un estado libre de error, el error se puede volver a producir. Por lo tanto, la fase final de la tolerancia a fallos es erradicar el fallo del sistema, de forma que se pueda continuar con el servicio normal.

El tratamiento automático de los fallos resulta difícil de implementar, y tiende a ser específico de cada sistema. Consecuentemente, algunos sistemas no disponen de tratamiento de fallos, asumiendo que todos los fallos son transitorios. Otros suponen que las técnicas de recuperación de errores son lo suficientemente potentes para tratar los fallos recurrentes.

El tratamiento de fallos se divide en dos fases: la localización del fallo y la reparación del sistema. Las técnicas de detección de errores pueden ayudar a realizar un seguimiento del fallo de un componente. En el caso de componentes hardware, esto puede ser suficientemente preciso, y el componente puede ser simplemente reemplazado. Un fallo software puede ser eliminado en una nueva versión del código. Sin embargo, en la mayoría de las aplicaciones que no se pueden parar será necesario modificar el programa mientras se está ejecutando. Esto representa un problema técnico bastante importante, por lo que estos casos no serán considerados en adelante.

5.6 La estrategia de bloques de recuperación en la tolerancia a fallos software

Los bloques de recuperación (Horning et al., 1974) son **bloques** en el sentido normal de los lenguajes de programación, excepto que en la entrada del bloque se encuentra un **punto de recuperación** automático, y en la salida un **test de aceptación**. El test de aceptación se utiliza para comprobar que el sistema se encuentra en un estado aceptable después de la ejecución del bloque (o **módulo primario**, como se le llama comúnmente). El fallo en el test de aceptación provoca que el programa sea restaurado al punto de recuperación del principio del bloque, y que se ejecute un **módulo alternativo**. Si el módulo alternativo también falla en el test de aceptación, entonces el programa es restaurado otra vez al punto de recuperación, y entonces se ejecuta otro módulo, y así sucesivamente. Si todos los módulos fallan entonces el bloque falla, y la recuperación debe llevarse a cabo a un nivel superior. La ejecución de un bloque de recuperación se muestra en la Figura 5.7.

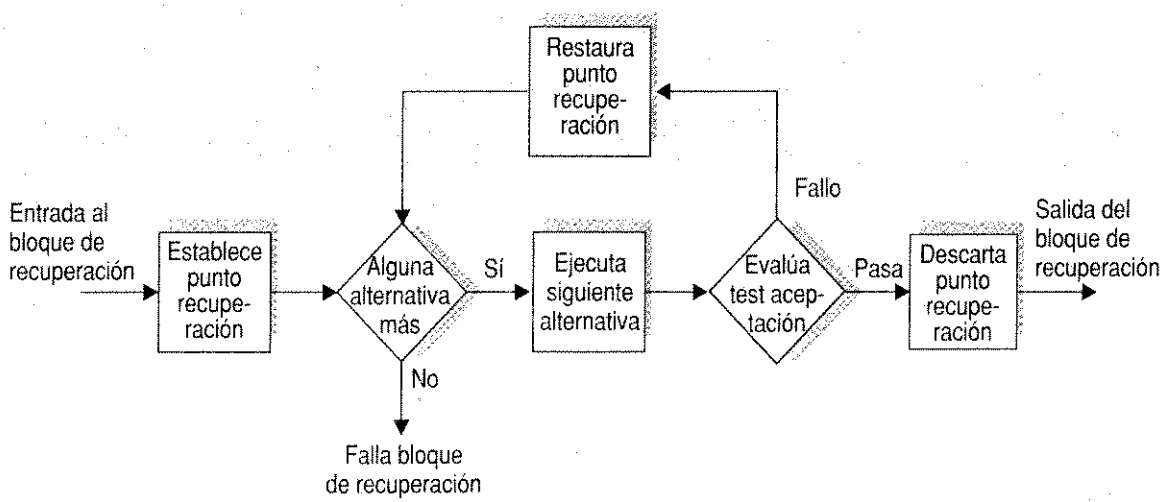


Figura 5.7. Mecanismo del bloque de recuperación.

En relación con las cuatro fases de la tolerancia a fallos software, tenemos que: la detección de errores es asumida por el test de aceptación; la evaluación de daños no es necesaria, ya que se supone que la recuperación de errores hacia atrás limpia los estados erróneos; y el tratamiento del fallo es realizado utilizando un recambio que estaba a la espera.

Aunque ningún lenguaje de programación en tiempo real disponible comercialmente tiene las características necesarias para explotar los bloques de recuperación, se han desarrollado algunos sistemas experimentales (Shrivastava, 1978; Purtilo y Jalote, 1991). A continuación, se ilustra una posible sintaxis para bloques de recuperación.

```
ensure <test aceptación>
by
  <modulo primario>
else by
  <modulo alternativo>
else by
  <modulo alternativo>
  ...
else by
  <modulo alternativo>
else error
```

Como los bloques ordinarios, los bloques de recuperación se pueden anidar. Si un bloque anidado en un bloque de recuperación falla, su test de aceptación y todas sus alternativas también fallan; en este caso, se restaurará el punto de recuperación del nivel exterior y se ejecutará el módulo alternativo de dicho módulo.

Para mostrar la utilización de los bloques de recuperación, se van a considerar varios métodos para encontrar la solución numérica de un sistema de ecuaciones diferenciales. Como dichos métodos no dan soluciones exactas, y están sujetos a diferentes errores, se puede encontrar que algunas aproximaciones resultarán más apropiadas para ciertas clases de ecuaciones que para otras. Desafortunadamente, los métodos que obtienen resultados precisos en un amplio rango de ecuaciones son caros de implementar (en cuanto al tiempo necesario para completar la ejecución del método). Por ejemplo, un método **Kutta explícito** será más eficiente que un método **Kutta implícito**. Sin embargo, sólo dará una tolerancia de error aceptable en algunos problemas concretos. Existe una clase de ecuaciones llamadas **rígidas** cuya solución utilizando un Kutta explícito conduce a la acumulación de errores de redondeo; se puede tratar el problema más adecuadamente con el método Kutta implícito, aunque resulte más costoso. A continuación se ofrece un ejemplo de utilización de bloques de recuperación que usa el método más barato en el caso de ecuaciones no rígidas, pero que no falla cuando se encuentra ecuaciones rígidas.

```
ensure error_redondeo_dentro_tolerancia_aceptable
by
  Método Kutta Explícito
else by
```

```
    Método Kutta Implícito
  else error
```

En este ejemplo, normalmente se utilizará el método explícito, que es más barato. Sin embargo, cuando falle se empleará el método explícito, que es más costoso. Aunque este error ha sido previsto, esta aproximación también permite una tolerancia respecto a un error en el diseño del algoritmo explícito. Si el algoritmo, en sí mismo, está en un error, y el test de aceptación es lo suficientemente general como para detectar ambos tipos de error en el resultado, se utilizará el algoritmo explícito. Cuando el test de aceptación no se puede hacer lo suficientemente general, se pueden utilizar en su lugar bloques de recuperación anidados. En el siguiente ejemplo, se proporciona una redundancia total de diseño. Al mismo tiempo, el algoritmo más barato será utilizado siempre que sea posible.

```
ensure error_redondeo_dentro_tolerancia_aceptable
by
  ensure valor_adecuado
  by
    Método Kutta Explícito
  else by
    Método Predictor-Corrector de K-Incrementos
  else error
else by
  ensure valor_adecuado
  by
    Método Kutta Implícito
  else by
    Método de K-Incrementos de Orden Variable
  else error
else error
```

En el ejemplo anterior, se dan dos métodos explícitos, y cuando ambos fallan en cuanto a producir un resultado adecuado, se ejecuta el método Kutta implícito. Por supuesto, también se ejecutará el método Kutta implícito si el valor producido por los métodos explícitos resulta adecuado pero no se encuentra dentro de la tolerancia requerida. Sólo si los cuatro métodos fallan las ecuaciones quedarán sin resolver.

Los bloques de recuperación podrían haber sido anidados de otro modo, según se muestra a continuación. En este caso, se producirá un comportamiento diferente cuando un valor no adecuado también esté fuera de la tolerancia aceptable. En el primer caso, después de ejecutar el algoritmo Kutta explícito, se podría haber intentado el método predictor corrector. En el segundo, debería ejecutarse el algoritmo Kutta implícito.

```
ensure valor_adecuado
by
  ensure error_redondeo_dentro_margen_aceptable
```

```
    by
      Método Kutta Explícito
    else by
      Método Kutta Implícito
    else error
  else by
    ensure error_redondeo_dentro_margen_aceptable
    by
      Método Predictor-Corrector de K-Incrementos
    else by
      Método de K-Incrementos de Orden Variable
    else error
  else error
```

5.6.1 El test de aceptación

El test de aceptación proporciona el mecanismo de detección de errores que hace posible explotar la redundancia en el sistema. El diseño del test de aceptación es crucial para la eficiencia del esquema de bloques de recuperación. Como ocurre dentro de todos los mecanismos de detección de errores, existe una relación entre proporcionar un test de aceptación exhaustivo y mantener al mínimo la sobrecarga que esto implica, de modo que la ejecución normal libre de errores se vea afectada lo menos posible. Hay que destacar que se utiliza el término **aceptación**, y no el término **corrección**, lo que permite que un componente pueda proporcionar un servicio degradado.

Todas las técnicas de detección de errores discutidas en la Sección 5.5.1 se pueden utilizar para el test de aceptación. Sin embargo, hay que tener mucho cuidado al diseñarlo, ya que un test de aceptación defectuoso puede permitir que errores residuales permanezcan sin ser detectados.

5.7 Una comparación entre la programación de *N*-versiones y los bloques de recuperación

Se han descrito dos aproximaciones para proporcionar software tolerante a fallos: la programación de *N*-versiones y los bloques de recuperación. Está claro que ambas comparten algunos aspectos de su filosofía básica, y al mismo tiempo son bastante diferentes. En esta sección se revisarán brevemente y se compararán ambas.

- **Redundancia estática frente a dinámica**

  La programación de *N*-versiones se basa en la redundancia estática: todas las versiones corren en paralelo, independientemente de si se producen o no fallos. Por contra, los bloques

de recuperación son dinámicos, ya que los módulos alternativos sólo se ejecutan cuando ha sido detectado algún error.

- **Sobrecargas asociadas**

  Tanto la programación de *N*-versiones como los bloques de recuperación implican cierto coste extra de desarrollo, ya que ambos requieren la creación de algoritmos alternativos. Además, en la programación de *N*-versiones se debe diseñar el proceso conductor, y en los bloques de recuperación el test de aceptación.

  En tiempo de ejecución, la programación de *N*-versiones necesita en general *N* veces los recursos de una única versión. Aunque los bloques de recuperación sólo necesitan un conjunto de recursos en un instante de tiempo, el establecimiento de los puntos de recuperación y la restauración del estado del proceso son caros. Sin embargo, es posible proporcionar un soporte hardware para el establecimiento de los puntos de recuperación (Lee et al., 1980), y la restauración del estado sólo será necesaria cuando se produzca un fallo.

- **Diversidad en el diseño**

  Ambas aproximaciones explotan la diversidad en el diseño para conseguir la tolerancia de errores no previstos. Ambas son, por lo tanto, susceptibles de contener errores originados en la especificación de requisitos.

- **Detección de errores**

  La programación de *N*-versiones utiliza la comparación de votos para detectar los errores, mientras que en los bloques de recuperación se utiliza un test de aceptación. Dependiendo de si las votaciones son exactas o inexactas, es probable que exista una sobrecarga menor que en los test de aceptación. Sin embargo, cuando sea difícil encontrar una técnica de votación inexacta, cuando existan múltiples soluciones, o cuando exista un problema de comparación consistente, los test de aceptación pueden dar más flexibilidad.

- **Atomicidad**

  La recuperación de errores hacia atrás es criticada debido a que no se pueden deshacer los errores que se hayan podido producir en el entorno. La programación de *N*-versiones elude este problema debido a que se supone que las versiones no interfieren entre ellas: son atómicas. Esto requiere que cada versión se comunique con el conductor en lugar de hacerlo directamente con el entorno. Sin embargo, resulta totalmente posible estructurar un programa de forma que las operaciones no recuperables no aparezcan en los bloques de recuperación.

Quizás deba destacarse que, aunque la programación de *N*-versiones y los bloques de recuperación han sido descritos como aproximaciones en competencia, también pueden ser consideradas como complementarias. Por ejemplo, no existe razón alguna para censurar a un diseñador que utilice bloques de recuperación dentro de cada versión de un sistema de *N*-versiones.

5.3 Redundancia dinámica y excepciones

En esta sección se va a presentar un marco de trabajo para implementar la tolerancia a fallos software basado en la redundancia dinámica y en la noción de excepciones y los gestores de excepciones.

A lo largo de este capítulo se ha utilizado el término «error» para referirse a la manifestación de un defecto, mientras que un defecto se considera la desviación de un componente de su especificación. Dichos errores pueden ser previstos (como ocurre en el caso de una lectura fuera de rango de un sensor debida a un mal funcionamiento del hardware) o imprevistos (como en el caso de un error en el diseño de un componente). Una **excepción** se puede definir como la ocurrencia de un error. Al hecho de mostrar la condición de excepción al que invocó la operación que ha causado dicha excepción, se le denomina **generar** (o **señalar** o **lanzar**) la excepción, mientras que a la respuesta del invocador se le denomina **gestión**, **manejo** o **captura** de la excepción. La gestión de excepciones se puede considerar como un mecanismo de recuperación hacia adelante, ya que cuando la excepción ha sido lanzada, el sistema no es retrotraído al estado anterior, sino que se pasa el control al gestor de forma que se puedan iniciar los procedimientos de recuperación. Sin embargo, según será mostrado en la Sección 6.5, las posibilidades de la gestión de excepciones también se pueden utilizar para proporcionar una recuperación de errores hacia atrás.

Aunque se ha definido una excepción como la ocurrencia de un error, existe cierta controversia acerca de la verdadera naturaleza de las excepciones y de cómo deberían ser utilizadas. Por ejemplo, considere un componente o módulo software que mantiene una tabla de símbolos del compilador. Una de las operaciones que proporciona es la búsqueda de un símbolo. Esto produce dos posibles resultados: *símbolo presente* o *símbolo ausente*. Cualquiera de los resultados es una respuesta prevista, y puede o no representar una condición de error. Si la operación *consulta* se utiliza para determinar la interpretación de un símbolo en el cuerpo de un programa, la *ausencia de un símbolo* se corresponde con un «identificador no declarado», lo que provoca una condición de error. Si, por el contrario, la operación *consulta* es utilizada durante el proceso de declaración, el resultado *símbolo ausente* será probablemente el caso normal, y el resultado *símbolo presente* (esto es, una «definición duplicada») será la excepción. Decidir qué constituye un error, por lo tanto, depende del contexto en el que se dé el evento. Sin embargo, en cualquiera de los casos anteriores se podría haber argumentado que el error no es un error del componente tabla de símbolos o del compilador, ya que cualquiera de las consecuencias es un resultado previsto y forma parte de la funcionalidad del módulo tabla de símbolos. Por lo tanto, ninguna de las consecuencias debería ser considerada como una excepción.

Las posibilidades de gestión de excepciones *no* fueron incorporadas a los lenguajes de programación para atender a los errores de diseño de los programadores; sin embargo, según se mostrará en la Sección 6.5, ahora se pueden utilizar para hacer exactamente eso. La motivación original de las excepciones vino de la necesidad de gestionar las condiciones anormales que surgían en los entornos en los que se ejecutaban los programas. Estas excepciones podrían ser consideradas eventos raros en el entorno de funcionamiento, y podría ser o no posible recuperar los programas después de su aparición. Una válvula defectuosa o una alarma de temperatura podrían

causar una excepción. Éstos son eventos raros que bien podrían ocurrir considerando un periodo de tiempo suficiente, y que deben ser tolerados.

A pesar de lo anterior, las excepciones y sus gestores se utilizarán inevitablemente como un mecanismo de gestión de errores de propósito general. Para concluir, las excepciones y la gestión de excepciones se pueden utilizar para:

- Enfrentarse a condiciones anormales que surgen en el entorno.
- Tolerar los defectos en el diseño del programa.
- Proporcionar unas capacidades de propósito general para la detección de errores y la recuperación.

Las excepciones se verán con más detalle en el Capítulo 6.

5.8.1 Componente ideal de un sistema tolerante a fallos

La Figura 5.8 muestra el componente ideal a partir del cual se construirán sistemas tolerantes a fallos (Anderson y Lee, 1990). Los componentes aceptan peticiones de servicio y, si fuera necesario, efectúan llamadas a servicios de otros componentes antes de producir una respuesta. Ésta podría ser una respuesta normal o una respuesta excepcional. En un componente ideal se pueden dar dos tipos de fallos: aquéllos debidos a una petición de servicio ilegal, llamados **excepciones de interfaz**, y aquéllos debidos a un mal funcionamiento del componente mismo, o de los componentes necesarios para servir la petición original. Cuando los componentes no pueden tolerar estos fallos mediante una recuperación de errores hacia adelante o hacia atrás, se lanzan excepciones de fallo en el componente invocador. Antes de lanzar cualquier excepción, el componente debe volver, si es posible, a un estado consistente, de forma que pueda servir una petición futura.

5.9 Medida y predicción de la fiabilidad del software

Las métricas de fiabilidad en los componentes hardware han sido establecidas hace tiempo. Tradicionalmente, cada componente es considerado como un representante de la población de miembros idénticos cuya fiabilidad es estimada como la proporción de una muestra que falla durante una prueba. Por ejemplo, se ha observado que, después de un periodo inicial de establecimiento, ciertos componentes electrónicos fallan en una tasa constante, por lo que su fiabilidad en el instante t puede ser modelada por la expresión

$$R(t) = Ge^{-\lambda t}$$

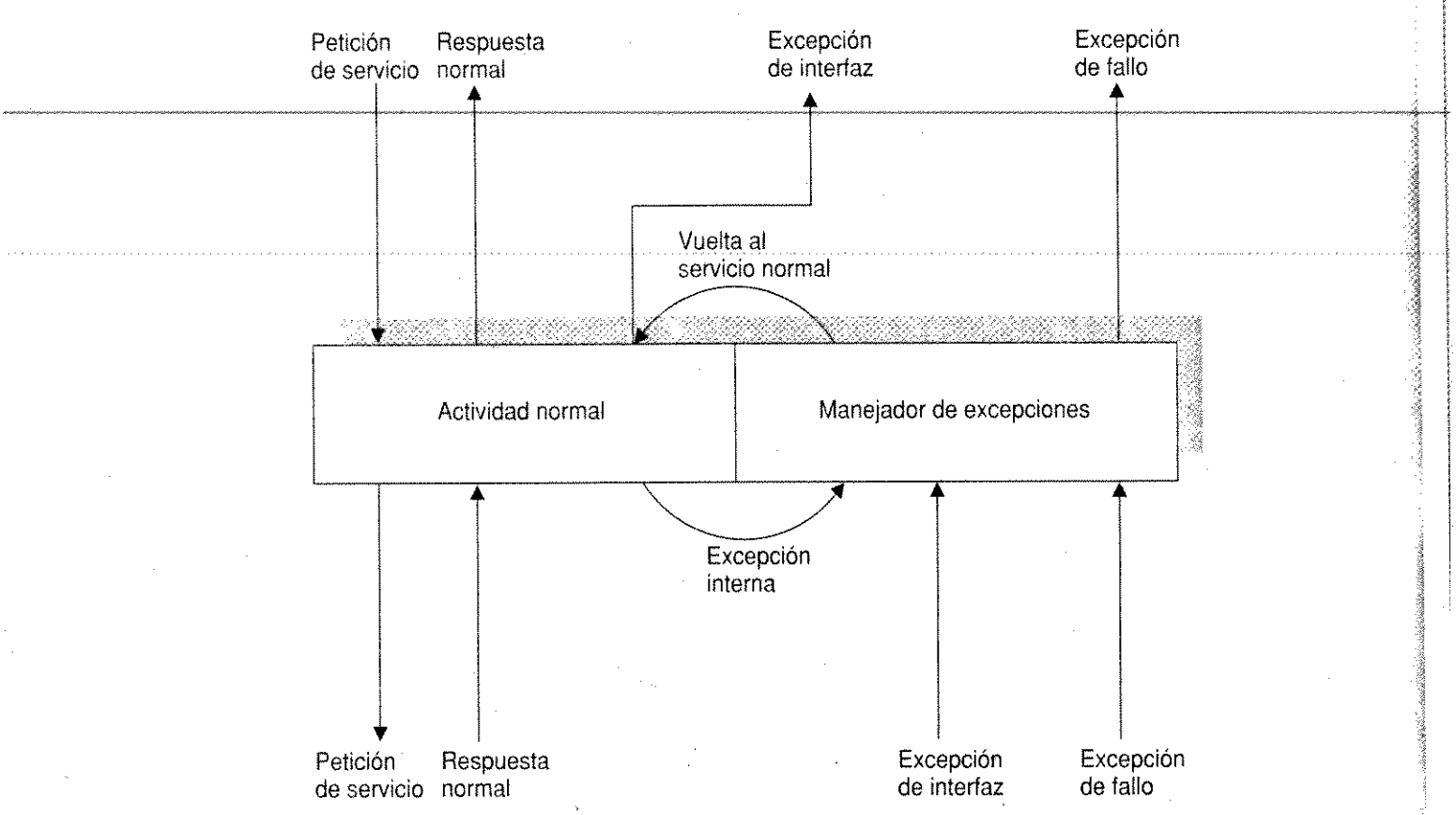


Figura 5.8. Un componente ideal tolerante a fallos.

donde G es una constante, y λ es la suma de las tasas de fallo de todos los componentes participantes. Una métrica utilizada normalmente es el tiempo medio entre fallos (Mean Time Between Failures; MTBF), el cual, para un sistema sin redundancia, es igual a $1/\lambda$.

La predicción y medida de la fiabilidad del software no es aún una disciplina bien establecida. Ha sido ignorada durante muchos años por aquellas industrias que necesitaban sistemas extremadamente fiables debido a que el software no se deteriora con el uso (simplemente se considera fiable o no fiable). Además, en el pasado determinados componentes software se utilizaban una única vez en aquellos sistemas para los cuales fueron desarrollados, por lo que aunque se eliminara cualquier error encontrado durante la fase de pruebas, esto no conducía al desarrollo de componentes más fiables que hubieran podido ser utilizados en cualquier otro caso. Esto se puede comparar con lo que sucede con los componentes hardware, los cuales son producidos en masa. Podrá corregirse cualquier error encontrado en la fase de prueba, haciendo que el siguiente lote sea más fiable. Sin embargo, ahora se considera que se puede mejorar la fiabilidad del software y reducir los costes asociados con el mismo mediante la reutilización de componentes. Desafortunadamente, nuestro conocimiento sobre cómo construir componentes software reutilizables esta lejos de ser completo (véase la Sección 4.5.1).

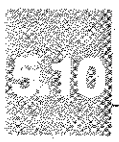
Aún es común la visión del software como correcto o no. Si no es correcto, la prueba de un programa indicará la localización de los defectos, que entonces podrán ser corregidos. Este capítulo ha intentado ilustrar que el enfoque tradicional de la prueba del software, aunque resulte indispensable, nunca puede asegurar que los programas se encuentren libres de errores, especialmente en el caso de sistemas muy grandes y complejos donde existirán especificaciones residuales o

errores de diseño. Más aún, a pesar de los rápidos avances en el campo de la demostración de la corrección, la aplicación de estas técnicas a sistemas no triviales, particularmente a aquéllos que implican el concepto tiempo, está todavía lejos del *estado actual del arte*. Por todas estas razones es por lo que se aboga por los métodos que incrementen la fiabilidad a partir del uso de la redundancia. Desafortunadamente, incluso con este enfoque no se puede garantizar que los sistemas que contengan software no vayan a fallar. Por lo tanto, resulta esencial que se desarrollen técnicas para predecir o medir su fiabilidad.

La fiabilidad del software puede ser considerada como la *probabilidad de que un programa dado funcione correctamente en un entorno concreto durante un determinado periodo de tiempo*. Se han propuesto varios modelos para intentar estimar la fiabilidad del software. Éstos pueden ser clasificados en general como (Goel y Bastini, 1985):

- Modelos de crecimiento de la fiabilidad del software.
- Modelos estadísticos.

Los modelos de crecimiento intentan predecir la fiabilidad de un programa basándose en su historial de errores. Los modelos estadísticos abordan la estimación de la fiabilidad de un programa mediante la determinación de la respuesta exitosa o fallida a algunos casos de prueba aleatorios, sin corregir ningún error encontrado. Los modelos de crecimiento se encuentran bastante maduros, y existe un amplio *corpus* de literatura y experiencia industrial respecto a su aplicación (Littlewood y Strigini, 1993; Bennett, 1994; Lytz, 1995). Sin embargo, Littlewood y Strigini (1993) han aducido que la prueba sólo puede proporcionar una evidencia de fiabilidad como mucho de un 10^{-4} (esto es, 10^{-4} errores por hora de funcionamiento). Esto debería ser comparado con el requisito de fiabilidad habitualmente exigido en las aplicaciones nucleares y de aeronáutica, que es de 10^{-9}.



Seguridad, fiabilidad y confiabilidad

El término «seguridad» (que fue definido informalmente en el Capítulo 2) puede ampliar su significado para incorporar los problemas analizados en este capítulo. Por lo tanto se puede definir la seguridad como *la ausencia de condiciones que puedan causar muerte, lesión, enfermedad laboral, daño (o pérdida) de equipos (o propiedades), o nocividad en el medio ambiente* (Leveson, 1986). Sin embargo, como esta definición podría considerar inseguros a la mayoría de los sistemas que tengan un elemento de riesgo asociado con su uso, la seguridad software se considera a menudo en términos de **percances** (Leveson, 1986). Un percance es un **evento no planeado** o **secuencias de eventos** que pueden producir muerte, lesión, enfermedad laboral, daño (o pérdida) de equipos (o propiedades) o nocividad en el medio ambiente.

Aunque la fiabilidad y la seguridad suelen considerarse como sinónimos, existe una diferencia en el énfasis. La fiabilidad ha sido definida como la medida del éxito con el cual un sistema se ajusta a la especificación de su comportamiento. Esto se expresa habitualmente en términos

de probabilidad. La seguridad, sin embargo, es la improbabilidad de que se den las condiciones que conducen a un percance, *independientemente de si se realiza la función prevista*. Estas dos definiciones pueden entrar en conflicto. Por ejemplo, aquellas medidas tendentes a incrementar la probabilidad de que un arma de fuego dispare cuando se le requiera, pueden incrementar la posibilidad de una detonación accidental. Desde muchos puntos de vista, el único aeroplano seguro es aquél que nunca despega; sin embargo, no es nada fiable.

De la misma manera que sucede con la fiabilidad, para cumplir con los requisitos de seguridad de un sistema embebido se debe realizar un análisis de seguridad a lo largo de todas las etapas de desarrollo de su ciclo de vida. Entrar en los detalles del análisis de la seguridad está fuera del alcance de este libro. Para un tratamiento general del análisis del árbol de defectos (una técnica utilizada para analizar la seguridad del diseño del software), el lector debe consultar Leveson y Harvey (1983).

5.10.1 Confiabilidad

En la última década se ha realizado mucho trabajo de investigación sobre la llamada fiabilidad o computación tolerante a fallos. Consecuentemente, estos términos se han sobrecargado, y los investigadores han buscado nuevas palabras y expresiones para expresar el aspecto particular que querían enfatizar. Los distintos términos asociados con el concepto de seguridad son ejemplos de esta nueva terminología. Llegados a este punto, se ha producido un intento de obtener definiciones claras y ampliamente aceptadas para los conceptos básicos en este campo. Por ello, se ha introducido la noción de **confiabilidad** (Laprie, 1985). *La confiabilidad de un sistema es la propiedad del sistema que permite calificar, justificadamente, como fiable al servicio que proporciona.* La confiabilidad, por lo tanto, incluye como casos especiales las nociones de fiabilidad y seguridad. La Figura 5.9, basada en otra dada por Laprie (1995), ilustra estos y otros aspectos de la confiabilidad (donde la seguridad es vista en términos de integridad y confidencialidad). En esta figura, el término «fiabilidad» se utiliza para medir la entrega continuada del servicio adecuado. La disponibilidad es la medida de la frecuencia de los periodos de servicio incorrecto. La confiabilidad puede ser descrita en relación con tres componentes (Laprie, 1995).

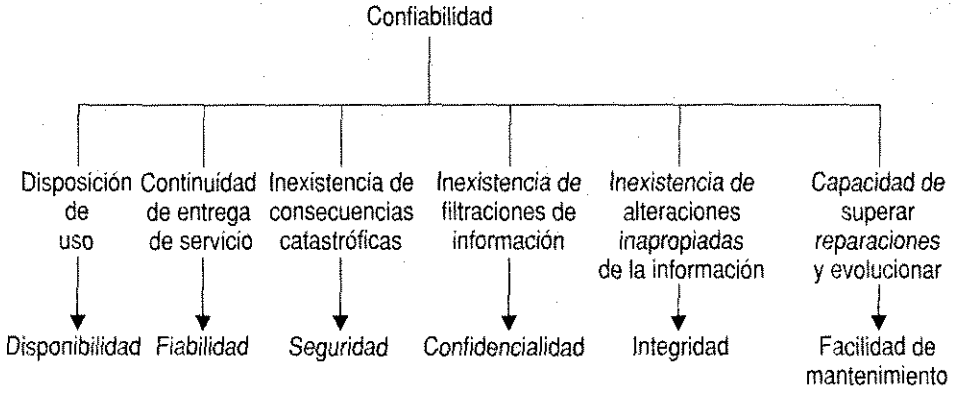


Figura 5.9. Aspectos de la confiabilidad.

- *Deficiencias*: circunstancias que causan o son producto de la no confiabilidad.
- *Medios*: los métodos, herramientas y soluciones requeridas para entregar un servicio confiable con la confianza requerida.
- *Atributos*: el modo y las medidas mediante las cuales se puede estimar la calidad de un servicio confiable.

Aunque existe un cierto acuerdo respecto a la terminología relacionada con la confiabilidad, está lejos de existir unanimidad, y la terminología aún está siendo refinada. Este libro, por lo tanto, continuará manejando aquellos nombres establecidos utilizados en la mayor parte de este capítulo.

Resumen

Este capítulo ha definido la fiabilidad como un requisito principal de los sistemas de tiempo real. La fiabilidad de un sistema ha sido definida como la medida del éxito con el que el sistema cumple con alguna de las especificaciones obligatorias de su comportamiento. Cuando el comportamiento de un sistema se desvía del especificado, se dice que se está produciendo un fallo. Los fallos son el resultado de defectos. Los defectos pueden ser introducidos accidentalmente o intencionadamente en el sistema. Pueden ser transitorios, permanentes o intermitentes.

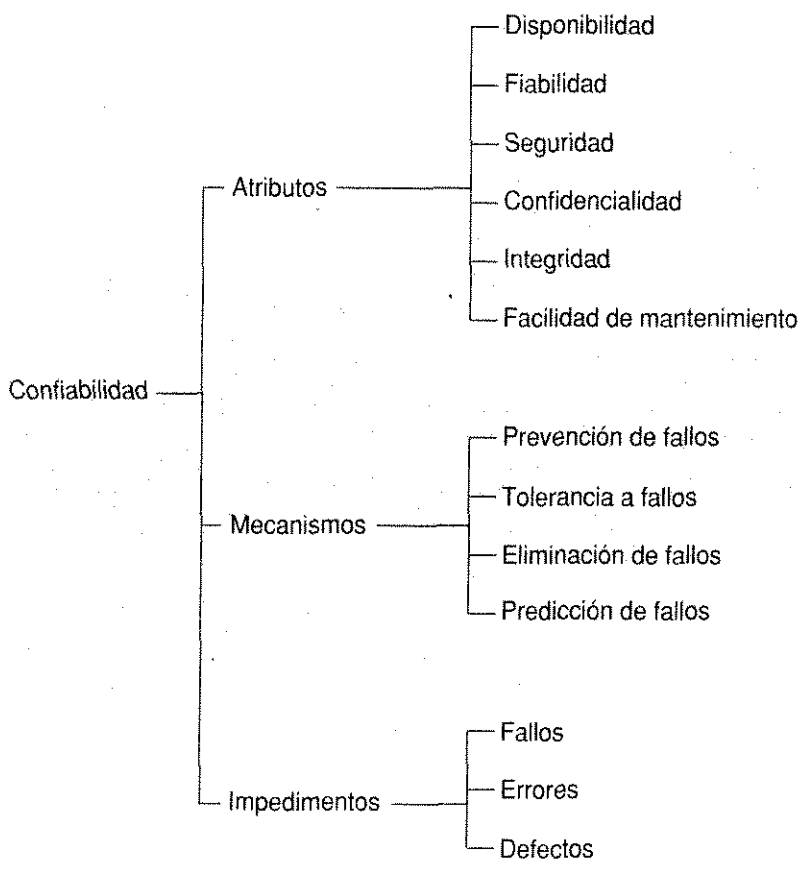


Figura 5.10. Terminología relacionada con la confiabilidad.

Existen dos enfoques en el diseño de un sistema que pueden ayudar a asegurar que los potenciales defectos no causen el fallo del sistema: la prevención de fallos y la tolerancia a fallos. La prevención de fallos consiste en evitar los fallos (intentando limitar la introducción de componentes defectuosos en el sistema) y en eliminarlos (el proceso de encontrar y arreglar los defectos). La tolerancia a fallos implica la introducción de componentes redundantes en el sistema, de forma que se puedan detectar y tolerar los fallos. En general, un sistema proporcionará tolerancia total a fallos, degradación controlada, o un comportamiento totalmente seguro. Se han discutido dos aproximaciones a la tolerancia a fallos en el software: la programación de *N*-versiones (redundancia estática), y la redundancia dinámica, que utiliza recuperación de errores hacia adelante y hacia atrás. La programación de *N*-versiones se define como la generación independiente de *N* programas funcionalmente equivalentes para la misma especificación inicial (donde *N* es mayor o igual que 2). Una vez diseñados y escritos, los programas son ejecutados concurrentemente con las mismas entradas, y se comparan sus resultados. En principio, los resultados deberían ser idénticos, pero en la práctica quizás exista alguna diferencia, en cuyo caso se consensúa un resultado, que es tomado como correcto. La programación de *N*-versiones se basa en la suposición de que un programa puede ser completo, consistente y no ambiguamente especificado, y que los programas que han sido desarrollados independientemente fallarán de forma independiente. Estas suposiciones puede que no siempre sean válidas, y aunque la programación de *N*-versiones tiene influencia en la producción de software fiable, debería ser utilizada con cuidado y junto con técnicas basadas en redundancia dinámica.

Las técnicas de redundancia dinámica tienen cuatro fases constituyentes: detección de errores, confinamiento y valoración de los daños, recuperación del error y tratamiento del fallo, y continuación del servicio. Las acciones atómicas han sido presentadas como una técnica estructural importante para ayudar en el confinamiento del daño. Una de las fases más importantes es la recuperación del error, para la que se han propuesto dos enfoques: la recuperación hacia adelante, y la recuperación hacia atrás. En la recuperación hacia atrás, es necesario que los procesos en comunicación alcancen estados de recuperación consistentes para evitar el efecto dominó. En los sistemas secuenciales, se ha presentado el bloque de recuperación como un concepto del lenguaje apropiado para expresar la recuperación hacia atrás. Los bloques de recuperación son bloques en el sentido normal de los lenguajes de programación, excepto por el hecho de que en la entrada del bloque existe un punto de recuperación automático y en la salida una prueba de aceptación. La prueba de aceptación se utiliza para ver si el sistema está en un estado aceptable después de la ejecución del módulo primario. El fallo en la prueba de aceptación propuesta hace que el programa sea devuelto al punto de recuperación del principio del bloque, y que se ejecute un módulo alternativo. Si el módulo alternativo también fallara en la prueba de aceptación, el programa será restituido otra vez al punto de recuperación y será ejecutado otro módulo, continuando de esta forma el proceso. Si todos los módulos fallaran, entonces el bloque fallará. Una comparación entre la programación de *N*-versiones y los bloques de recuperación ha ilustrado las similitudes y las diferencias entre estos enfoques.

Aunque la recuperación de errores hacia adelante es específica de cada sistema, la gestión de excepciones se ha mostrado como un marco de trabajo apropiado para su implementación. Se presentó el concepto de un componente tolerante a fallos ideal que utiliza excepciones.

Finalmente, se han presentado en este capítulo las nociones de seguridad y confiabilidad del software.

Lecturas complementarias

Anderson, T., y Lee, P. A. (1990), *Fault Tolerance, Principles and Practice*, 2.ª ed., Englewood Cliffs, NJ: Prentice Hall.

Laprie J.-C. y otros (1995), *Dependability Handbook*, Toulouse: Cépaduès (en francés).

Leveson, N. G. (1995), *Safeware: System Safety and Computers*, Reading, MA: Addison-Wesley.

Mili, A. (1990), *An Introduction to Program Fault Tolerance*, New York: Prentice Hall.

Neumann, P. G. (1995), *Computer-Related Risks*, Reading, MA: Addison-Wesley.

Powell, D. (ed.) (1991), *Delta-4: A Generic Architecture for Dependable Distributed Computing*, Londres: Springer-Verlag.

Randell, B., Laprie, J.-C., Kopetz, H., y Littlewood, B. (eds) (1995), *Predictable Dependable Computing Systems*, Londres: Springer.

Redmill, F., y Rajan, J. (eds) (1997), *Human Factors in Safety-Critical Systems*, Oxford: Butterworth-Heinemann.

Storey, N. (1996), *Safety-Critical Computer Systems*, Reading, MA: Addison-Wesley.

Existe también una serie de libros publicados por Kluwer Academic Publishers y editados por G. M. Koob y C. G. Lau sobre los «fundamentos de los sistemas confiables» («Foundations of Dependable Systems»).

Ejercicios

5.1 ¿Es fiable un programa que es conforme a una especificación errónea de su comportamiento?

5.2 ¿Cuáles deberían ser los niveles de degradación de servicio apropiados para un automóvil controlado por computador?

5.3 Escriba un bloque de recuperación para la ordenación de un array de enteros.

5.4 ¿Hasta qué punto es posible detectar lineas de recuperación en tiempo de ejecución? (Véase Anderson y Lee (1990), Capítulo 7.)

5.5 La Figura 5.11 muestra la ejecución concurrente de cuatro procesos en comunicación ($P1$, $P2$, $P3$ y $P4$) y sus puntos de recuperación asociados (por ejemplo, $R11$ es el primer punto de recuperación para el proceso $P1$).

Explique qué ocurre cuando:

- El proceso $P1$ detecta un error en el instante t.
- El proceso $P2$ detecta un error en el instante t.

5.6 ¿Debería ser señalada al programador como una excepción la condición de fin de fichero en una lectura secuencial de un fichero?

5.7 La diversidad de datos es una estrategia de tolerancia a fallos que complementa la diversidad de diseño. ¿Bajo qué condiciones debería ser más apropiada la diversidad de datos que la diversidad de diseños? (Consejo: véase Ammann y Knight (1988).)

5.8 ¿Debería ser juzgada la dependencia de un sistema por parte de un asesor independiente?

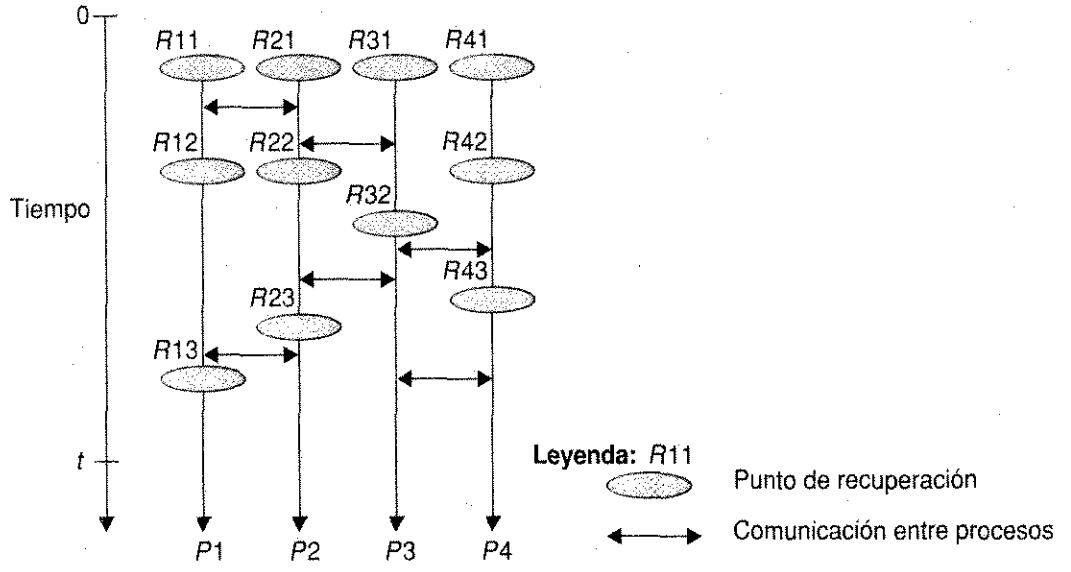


Figura 5.11. Ejecución concurrente de cuatro procesos para el Ejercicio 5.5.

Capítulo 6

Excepciones y manejo de excepciones

El Capítulo 5 trataba de cómo podían hacerse sistemas más fiables, y presentaba las excepciones como un marco de trabajo para implementar la tolerancia a fallos. En este capítulo, se consideran en más detalle las excepciones y el manejo de excepciones, y se discute cómo se desarrollan en lenguajes de programación concretos para tiempo real.

Existen ciertos requisitos generales para cualquier mecanismo de manejo de excepciones:

(R1) Como ocurre con cualquier aspecto de un lenguaje, el mecanismo debe ser fácil de comprender y utilizar.

(R2) El código de manejo de excepciones no debiera complicar el aspecto del programa hasta el punto de obscurecer la comprensión de la operación normal del programa (sin errores). Un mecanismo donde se funde el código para el procesamiento normal y el procesamiento excepcional resultará difícil de comprender y de mantener, y bien pudiera llevarnos a un sistema menos fiable.

(R3) El mecanismo deberá diseñarse de modo que sólo suponga una sobrecarga en la ejecución cuando se maneja una excepción. Aunque la mayoría de las aplicaciones requieren que las prestaciones de un programa que use excepciones no se vea afectada negativamente bajo condiciones normales, no tiene por qué ser siempre así. Bajo ciertas circunstancias, en concreto cuando sea primordial la rapidez en la recuperación, una aplicación puede estar preparada para tolerar cierta pequeña sobrecarga en una operación normal libre de errores.

(R4) El mecanismo deberá permitir un tratamiento uniforme de las excepciones detectadas tanto por el entorno como por el programa. Por ejemplo, una excepción como el **desbordamiento aritmético**, que es detectada por el hardware, debiera ser manejada exactamente de la misma forma que una excepción generada por el programa como resultado de un fallo en una aserción.

(R5) Como se indicó en el Capítulo 5, el mecanismo de excepciones deberá permitir la programación de acciones de recuperación.

6.1 Manejo de excepciones en los lenguajes de tiempo real primitivos

Aunque los términos «excepción» y «manejo de excepciones» se han puesto de moda sólo recientemente, se refieren simplemente a una forma de programar para contener y manejar las situaciones de error. En consecuencia, la mayoría de los lenguajes de programación disponen de medios para manejar al menos algunas excepciones. Esta sección describe brevemente estos mecanismos en relación con los requisitos mencionados anteriormente.

6.1.1 Retorno de un valor inusual

Una de las formas más primitivas de mecanismo de manejo de excepciones es el *retorno de un valor inusual* o *retorno de error* desde un procedimiento o una función. Su principal ventaja es que es simple, y que para su implementación no precisa de ningún mecanismo nuevo en el lenguaje. C soporta este modo, y se usa así:

```
if (llamada_funcion(parametros) == UN_ERROR) {
   /* código para el manejo de errores */
} else {
   /* código para el retorno normal */
};
```

Como puede verse, aunque presenta el requisito de simplicidad (R1) y permite programar acciones para la recuperación (R5), no satisface R2, R3 y R4. El código se complica, supone una sobrecarga, y no queda claro cómo manejar los errores detectados desde el entorno.

A lo largo de este libro, C se utiliza junto con POSIX. Las condiciones de error en POSIX se indican con valores de retorno distinto de cero (cada uno con un nombre simbólico). Por seguridad, al llamar a una función del sistema debiera comprobarse el valor devuelto para ver si se ha producido un error inesperado; aunque, como se puede ver, se obscurece la estructura del código. Por esto, y con fines pedagógicos, en este libro se emplea una interfaz de POSIX estilizada. Para cada llamada POSIX al sistema, se supone definida una macro que efectuará la comprobación del error. Por ejemplo, una llamada al sistema como `sys_call` que tome un parámetro, tendrá definida la siguiente macro.

```
#define SYS_CALL(A) if(sys_call(A) != 0) error()
```

donde `error` es una función que efectúa el procesamiento del error. Así, en la llamada sólo aparecerá `SYS_CALL (param)`.

6.1.2 Bifurcación forzada

En los lenguajes de ensamblado, la forma usual de manejar excepciones en subrutinas es *sobrepasar el retorno*. Es decir, se sobrepasa la instrucción inmediatamente posterior a la llamada a la subrutina para indicar la presencia (o ausencia) de errores. Para ello, la subrutina incrementa su dirección de retorno (contador de programa) en el tamaño de una instrucción de salto simple con el fin de indicar un retorno sin (o con) error. En caso de que sea posible más de un retorno excepcional, la subrutina presupondrá que tras la llamada hay varias instrucciones de salto, y manipulará convenientemente el contador de programa.

Por ejemplo, partiendo de que hay dos condiciones de error posibles, con el siguiente código se llama a cierta subrutina que envía un símbolo a un dispositivo.

```
jsr pc, IMPRIME_SIMB
jmp ERROR_ES
jmp DISPOSITIVO_NO_PREPARADO
# procesamiento normal
```

En el caso de un retorno normal, la subrutina incrementará la dirección de retorno en dos instrucciones `jmp`.

Si bien esta práctica implica poca sobrecarga (R3) y permite programar acciones de recuperación (R5), obscurece la estructura del programa, violando así los requisitos R1 y R2. Tampoco puede satisfacerse R4.

6.1.3 `goto` no local

Una versión de la bifurcación forzada sobre un lenguaje de alto nivel requeriría pasar como parámetros las diferentes etiquetas, o disponer de variables de etiqueta estándar (una variable de etiqueta es un objeto al que se le puede asignar una dirección del programa y que puede ser usado para transferir el control). RTL/2 es un caso de lenguaje de tiempo real primitivo que proporciona esta posibilidad bajo la forma de un `goto` no local. RTL/2 emplea *bricks* (ladrillos) para estructurar sus programas: un brick pueden ser datos (delimitado por `data...enddata`), un procedimiento (delimitado por `proc...endproc`), o una pila (indicada por `stack`). El sistema define un tipo especial de brick de datos llamado `data svc`. Uno de estos bricks (`rrerr`) proporciona un mecanismo estándar de manejo de errores que incluye una variable de etiqueta de error llamada `erl`.

El siguiente ejemplo muestra cómo puede usarse la etiqueta de error en RTL/2

```
svc data rrerr
  label erl; % una variable de etiqueta %
  ...
enddata
proc DondeSeDetectaElError();
  ...
```

```
    goto erl;
    ...
endproc;
proc Invocador();
    ...
    DondeSeDetectaElError();
    ...
endproc;
proc main();
    ...
reinicio:
    ...
    erl := reinicio;
    ...
    Invocador();
    ...
end proc;
```

Vemos que cuando se emplea de esta forma, el `goto` es más que un simple salto: implica un retorno anormal de un procedimiento, por lo que habrá que desapilar datos de la pila hasta restaurar el entorno del procedimiento que contiene la declaración de la etiqueta. La penalización por el desapilamiento se aplica sólo cuando aparece un error, por lo que se satisface el requisito R3. Aunque el empleo de `goto` es muy flexible (satisfaciendo R4 y R5), puede desembocar en programas muy obscuros. En consecuencia, no satisface los requisitos R1 y R2.

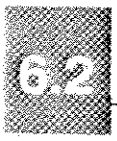
6.1.4 Variable de procedimiento

Aunque el ejemplo en RTL/2 muestra cómo recuperar los errores mediante una etiqueta de error, se pierde el dominio sobre el flujo del programa. En RTL/2, la etiqueta de error suele emplearse con los errores irrecuperables, recurriéndose a una *variable de procedimiento de error* cuando el control debe retornar al punto donde se originó el error. El siguiente ejemplo ilustra este método.

```
svc data rrerr;
    label erl;
    proc(int) erp; % erp es una variable de procedimiento %
enddata;

proc recupera(int);
    ...
    ...
endproc;
proc DondeSeDetectaElError();
    ...
```

```
    if recuperable then
      erp(n)
    else
      goto erl
    end;
    ...
endproc;

proc Invocador();
    ...
    DondeSeDetectaElError();
    ...
endproc;

proc main();
    ...
    erl := fallo;
    erp := recupera;
    ...
    Invocador();
    ...
fallo:
    ...
end proc;
```

De nuevo, el principal reproche que puede hacerse a esta aproximación es que los programas pueden volverse muy difíciles de entender y de mantener.



6.2 Manejo de excepciones moderno

En las aproximaciones tradicionales al manejo de excepciones suele ocurrir que el código de manejo se entremezcla con el flujo normal de ejecución del programa. Actualmente, se tiende a incluir funcionalidades de manejo de excepciones directamente en el lenguaje, resultando un mecanismo más estructurado. La naturaleza exacta de estas facilidades varía de un lenguaje a otro, aunque se pueden identificar algunas formas de proceder comunes. En las secciones siguientes se discuten estos mecanismos.

6.2.1 Excepciones y su representación

En la Sección 5.5.1, se vió que había dos técnicas principales para detectar errores: detección desde el entorno y desde la aplicación. También, según el retraso en la detección del error, puede ser

preciso generar la excepción síncrona o asíncronamente. La excepción síncrona se genera como respuesta inmediata a una operación inapropiada desde un fragmento de código. La excepción asíncrona se genera cierto tiempo después de que ocurra la operación que da lugar a la aparición del error. Puede generarse desde el proceso que ejecutó la operación originalmente, o en otro proceso. En consecuencia, existen cuatro clases de excepciones:

(1) Detectada por el entorno y generada síncronamente, como por ejemplo una violación de los límites de un array o una división por cero.

(2) Detectada por la aplicación y generada síncronamente, como el fallo en la comprobación de un aserto definido en el programa.

(3) Detectada por el entorno y generada asíncronamente, como en las excepciones generadas como resultado de un fallo de alimentación o en algún mecanismo de monitorización vital.

(4) Detectada por la aplicación y generada asíncronamente, como cuando un proceso verifica una condición de error que ocasionará que un proceso no cumpla los plazos o no termine correctamente.

Las excepciones asíncronas suelen recibir el nombre de notificaciones asíncronas o señales, y se suelen emplear en el contexto de la programación concurrente. Este capítulo se centrará en el manejo de excepciones síncronas, y se dejará el tema del manejo de excepciones asíncronas hasta el Capítulo 10.

Existen diversas formas de declarar las excepciones síncronas. Por ejemplo:

- Mediante un nombre constante que necesita ser declarado explícitamente.
- Mediante un objeto de cierto tipo que podrá o no ser declarado explícitamente.

Ada precisa que las excepciones se declaren como constantes; por ejemplo, las excepciones que puede generar el entorno de ejecución de Ada se declaran en el paquete `Standard`:

```
package Standard is
   ...
   Constraint_Error : exception;
   Program_Error : exception;
   Storage_Error : exception;
   Tasking_Error : exception;
   ...
end Standard;
```

Este paquete es visible para todos los programas Ada.

Java y C++ dan una visión de las excepciones más orientada al objeto. En Java, las excepciones son instancias de una subclase de la clase `Throwable`, que podrán ser **lanzadas** (*thrown*)

por el sistema de ejecución y por la aplicación (véase la Sección 6.3.2). En C++, se pueden lanzar excepciones de cualquier tipo de objeto sin declararlas previamente.

6.2.2 El dominio de un manejador de excepciones

En el programa, puede que haya diversos manejadores para una misma excepción. Cada manejador lleva asociado un dominio que especifica la región del cómputo es la cual se activará dicho manejador si aparece la excepción. La resolución con la que pueda especificarse el dominio determinará el grado de precisión de la localización del origen de la excepción. En lenguajes estructurados por bloques, como Ada, el dominio suele ser el bloque. Por ejemplo, considere un sensor de temperatura cuyo valor está en el rango de 0 °C a 100 °C. El siguiente bloque Ada establece que la temperatura es un valor entero entre 0 y 100. Si el valor asignado cae fuera del rango, el sistema de apoyo de ejecución de Ada genera una excepción `Constraint_Error`. La invocación del manejador asociado permite efectuar la necesaria acción correctiva.

```
declare
  subtype Temperatura is Integer range 0 .. 100;
begin
  -- lee el sensor de temperatura y calcula su valor
exception
  -- manejador para Constraint_Error
end;
```

En breve se completarán los detalles en Ada.

Donde los bloques son la base de otras unidades, como pueden ser procedimientos o funciones, el dominio de un manejador de excepciones suele ser esa unidad.

En algunos lenguajes, como Java, C++ o Modula-3, no todos los los bloques pueden tener manejadores de excepciones, por lo que habrá que declarar explícitamente el dominio de cada manejador de excepciones, a partir de lo cual se considerará que el bloque está **vigilado** (guarded); en Java, esto se indica usando un «bloque try»:

```
try {
  // sentencias que pueden generar una excepción
}
catch (ExceptionType e) {
  // manejador para e
}
```

Puesto que el dominio de un manejador de excepciones especifica dónde puede haberse generado el error, podría ocurrir que el bloque presentara una granularidad inadecuada. Por ejemplo, considere la siguiente secuencia de cómputos, en cualquiera de los cuales podría generarse un `Constraint_Error`.

```
declare
  subtype Temperatura is Integer range 0 .. 100;
  subtype Presion is Integer range 0 .. 50;
  subtype Caudal is Integer range 0 .. 200;
begin
  -- lee el sensor de temperatura y calcula su valor
  -- lee el sensor de presión y calcula su valor
  -- lee el sensor de caudal y calcula su valor
  -- ajusta la temperatura, la presión y el caudal
  -- según los requisitos
exception
  -- manejador para Constraint_Error
end;
```

El problema para el manejador es decidir qué cálculo provocó que se generara la excepción. Aún hay más problemas si consideramos desbordamientos o subdesbordamientos aritméticos.

Una solución del problema, con dominios de manejo de excepciones basados en bloques, sería disminuir el tamaño del bloque y/o anidarlo. Para el ejemplo del sensor:

```
declare
  subtype Temperatura is Integer range 0 .. 100;
  subtype Presion is Integer range 0 .. 50;
  subtype Caudal is Integer range 0 .. 200;
begin
  begin
  -- lee el sensor de temperatura y calcula su valor
exception
  -- manejador para Constraint_Error para la temperatura
end;
begin
  -- lee el sensor de presión y calcula su valor
exception
  -- manejador para Constraint_Error para la presión
end;
begin
  -- lee el sensor de caudal y calcula su valor
exception
  -- manejador para Constraint_Error para el caudal
end;
  -- ajusta la temperatura, la presión y el caudal
  -- según los requisitos
exception
```

```
  -- manejador para otras excepciones posibles
end;
```

Otra alternativa es crear procedimientos con manejadores para cada uno de los bloques anidados. Sin embargo, y de cualquier modo, es una solución demasiado prolija y tediosa. Otra solución es permitir el manejo de excepciones a nivel de sentencia. Con esta aproximación, el ejemplo anterior podría escribirse de esta forma:

```
-- Ada NO VALIDO
declare
  subtype Temperatura is Integer range 0 .. 100;
  subtype Presion is Integer range 0 .. 50;
  subtype Caudal is Integer range 0 .. 200;
begin
  Lee_Sensor_Temperatura;
    exception -- manejador para Constraint_Error;

  Lee_Sensor_Presion;
    exception -- manejador para Constraint_Error;

  Lee_Sensor_Caudal;
    exception -- manejador para Constraint_Error;

  -- ajusta la temperatura, la presión y el caudal
  -- según los requisitos
end;
```

El lenguaje de programación CHILL (CCITT, 1980) dispone de dicha posibilidad. Aunque esto permite ubicar con más precisión la causa de la excepción, la mezcla del código de manejo de excepciones con el flujo de operación normal puede dar lugar a programas poco claros, violando así el requisito R2.

La mejor aproximación a este problema es permitir el paso de parámetros junto a las excepciones. Java lo hace automáticamente, dado que una excepción es un objeto, y en consecuencia puede contener tanta información como desee el programador. Por contra, Ada proporciona un procedimiento predefinido `Exception_Information` que devuelve detalles definidos en la implementación sobre la aparición de la excepción.

6.2.3 Propagación de excepciones

Íntimamente asociado al concepto de dominio de una excepción, se encuentra la noción de propagación de una excepción. Hasta el momento se daba por sentado que si un bloque o un procedimiento generaban una excepción, había un manejador asociado a ese bloque o procedimiento.

Sin embargo, no siempre es éste el caso. Hay dos formas posibles de proceder cuando no existe un manejador de excepciones inmediato.

La primera de ellas es entender la ausencia de manejador como un error del programador, que deberá notificarse en tiempo de compilación. Sin embargo, puede ocurrir que cierta excepción generada en un procedimiento sólo pueda ser manejada apropiadamente en el contexto del procedimiento que lo invoca. En estos casos, no es posible incluir el manejador junto al procedimiento. Por ejemplo, una excepción generada en un procedimiento al fallar cierta aserción sobre los parámetros, y que sólo puede manejarse en el contexto de la llamada. Desgraciadamente, el compilador no siempre puede comprobar si el contexto de llamada incluye los manejadores de excepciones adecuados, a no ser que efectúe un complejo análisis del control del flujo. Esto es especialmente difícil cuando el procedimiento llama a otros procedimientos que también pueden generar excepciones. Por esto, los lenguajes que precisan generación de errores en tiempo de compilación requieren que los procedimientos especifiquen qué excepciones pueden generar (es decir, cuáles no manejarán localmente). Así, el compilador podrá comprobar si el contexto de llamada dispone del manejador apropiado, y generará, si es preciso, el mensaje de error correspondiente. Ésta es la aproximación seguida por el lenguaje CHILL. Java y C++ permiten definir qué excepciones generará una función. Sin embargo, a diferencia de CHILL, no requieren que haya un manejador disponible en el contexto de llamada.

La segunda vía, que puede ser seguida cuando no puede localizarse un manejador de excepciones, es buscar manejadores en el pasado de la cadena de invocaciones en tiempo de ejecución; esto se conoce como **propagación** de excepciones. Ada y Java permiten la propagación de excepciones, así como C++ y Modula-2/3.

Cuando el lenguaje precisa declarar el alcance de las excepciones, puede aparecer un problema potencial con la propagación. Bajo ciertas condiciones puede que una excepción se propague fuera de su alcance, haciendo imposible su captura por un manejador. Para hacer frente a esta situación, la mayoría de los lenguajes proporcionan un manejador «atrapa todo» (catch all).

Una excepción sin manejar provoca que un programa secuencial sea abortado. Normalmente, si el programa consta de más de un proceso y alguno de ellos no maneja cierta excepción que hubiera generado, se abortará ese proceso. Sin embargo, no queda claro si se debiera propagar la excepción al proceso padre. Las excepciones en los programas multiproceso se considerarán detalladamente en el Capítulo 10.

Otra cuestión sobre la propagación de excepciones es si los manejadores se asocian a las excepciones dinámica o estáticamente. La asociación estática, como en CHILL, se efectúa en la compilación, por lo que no permite la propagación, al no conocerse la cadena de procedimientos invocadores. La asociación dinámica se realiza en ejecución, y consecuentemente permite la propagación. Aunque la asociación dinámica es más flexible, conlleva más sobrecarga en la ejecución donde haya que buscar el manejador correspondiente; en la asociación estática es posible generar la dirección del manejador en la compilación.

6.2.4 Modelo de reanudación y modelo de terminación

Una cuestión crucial en cualquier mecanismo de manejo de excepciones es si el que efectúa la llamada a la excepción debiera continuar su ejecución tras el manejo de la excepción. Si el invocador pudiera continuar, y suponiendo que el manejador resolviera el problema que causó la generación de la excepción, sería posible que pudiera reanudar su trabajo como si nada hubiera pasado. Esto se conoce como modelo de **reanudación** o de **notificación**. El modelo en el que no se devuelve el control al invocador se denomina de **terminación** o de **escape**. Obviamente, cabe pensar en un modelo donde el manejador pudiera decidir si reanudar la operación que provocó la excepción, o terminar la operación. Este modelo se denomina **híbrido**.

Modelo de reanudación

Para ilustrar el modelo de reanudación, considere tres procedimientos (*P*, *Q* y *R*). El procedimiento *P* invoca *Q*, que a su vez invoca *R*. El procedimiento *R* genera una excepción (*r*) que es manejada por *Q*, asumiendo que no hay un manejador local en *R*. El manejador para *r* es *Hr*. Mientras maneja *r*, *Hr* genera la excepción *q*, que es manejada por *Hq* en el procedimiento *P* (el invocador de *Q*). Una vez manejada *q*, *Hr* continúa su ejecución, y después continúa *R*. El diagrama de la Figura 6.1 representa esta secuencia de eventos con los arcos numerados de 1 a 6.

El modelo de reanudación se entiende mucho mejor si contemplamos el manejador como un procedimiento que se invoca implícitamente al generar la excepción.

El problema de esta aproximación es la dificultad a la hora de reparar errores generados por el entorno de ejecución. Por ejemplo, un desbordamiento aritmético en medio de una secuencia compleja de expresiones podría ocasionar que varios de los registros contuvieran evaluaciones parciales. Al llamar al manejador, es probable que se sobreescriban dichos registros.

Tanto el lenguaje Pearl como el lenguaje Mesa proporcionan un mecanismo que permite al manejador volver al contexto donde se generó la excepción. Ambos lenguajes soportan también el modelo de terminación.

Puesto que es difícil implementar un modelo de reanudación estricto, se puede tomar el compromiso de reejecutar el bloque asociado con el manejador de la excepción. El lenguaje Eiffel (Meyer, 1992) lo permite mediante **retry** (reintenta), como parte de su modelo de manejo de excepciones. El manejador puede establecer un indicador local que indique que se ha producido un error, y el bloque podrá comprobar dicho indicador. Observe que para que funcione este esquema, las variables locales del bloque no deben ser reinicializadas en el reintento.

Cuando realmente se nota la ventaja del modelo de reanudación es cuando la excepción ha sido generada asíncronamente, y por tanto tiene poco que ver con la ejecución actual del proceso. El manejo de eventos asíncrono se discute con detalle en la Sección 10.5.

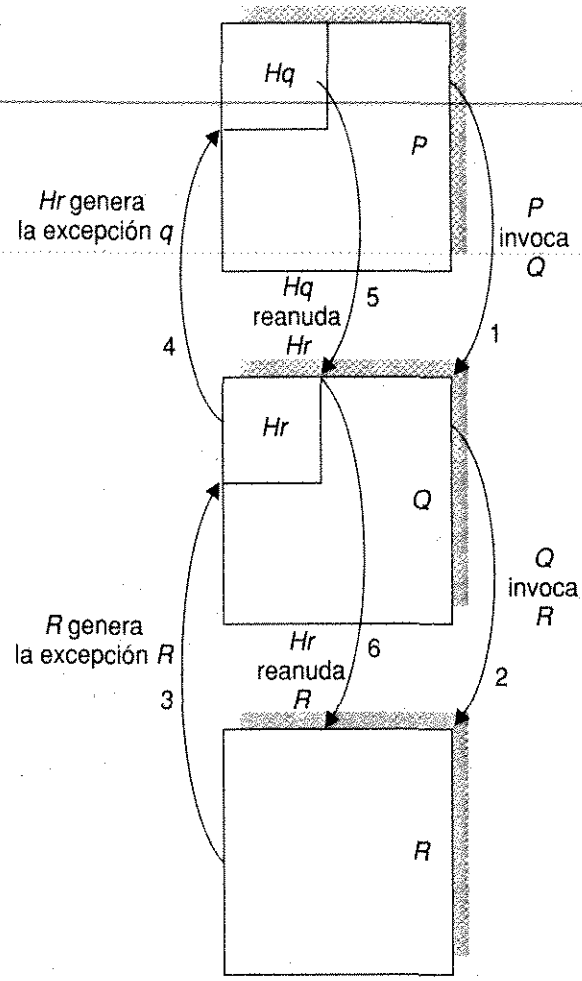


Figura 6.1. El modelo de reanudación.

Modelo de terminación

En el modelo de terminación, tras llamar al manejador para que atienda cierta excepción, el control no retorna al punto donde apareció la excepción, sino que se finaliza el bloque o procedimiento que contiene el manejador y se pasa el control al bloque o al procedimiento de llamada. De este modo, un procedimiento invocado puede terminar en varias condiciones de terminación, siendo una de ellas la **condición normal**, y las restantes **condiciones de excepción**.

Cuando el manejador está en un bloque, tras el manejo de la excepción se devuelve el control a la primera sentencia que sigue al bloque, como se indica en el siguiente ejemplo.

```
declare
  subtype Temperatura is Integer range 0 .. 100;
begin
  ...
  begin
    -- lee el sensor de temperatura y calcula su valor,
    -- puede ocasionar la generación de una excepción
```

```
exception
   -- manejador de Constraint_Error para temperatura,
   -- este bloque termina una vez realizado el manejo.
end;

   -- este bloque se ejecuta cuando el bloque acaba normalmente
   -- o cuando se ha generado y manejado una excepción.
exception
   -- manejador de otras excepciones posibles
end;
```

Con los procedimientos, al contrario que con los bloques, el control del flujo puede cambiar drásticamente, tal y como muestra la Figura 6.2. De nuevo, el procedimiento *P* ha invocado al procedimiento *Q*, que a su vez ha invocado al procedimiento *R*. Se ha generado una excepción en *R* y se ha manejado en *Q*.

Ada, Java, C++, Modula-2/3 y CHILL tienen todos el modelo de terminación de manejo de excepciones.

Modelo híbrido

En el modelo híbrido, es el manejador el que decide si el error es recuperable. Si es así, el manejador puede devolver un valor, y la semántica sería la misma que en el modelo de reanudación.

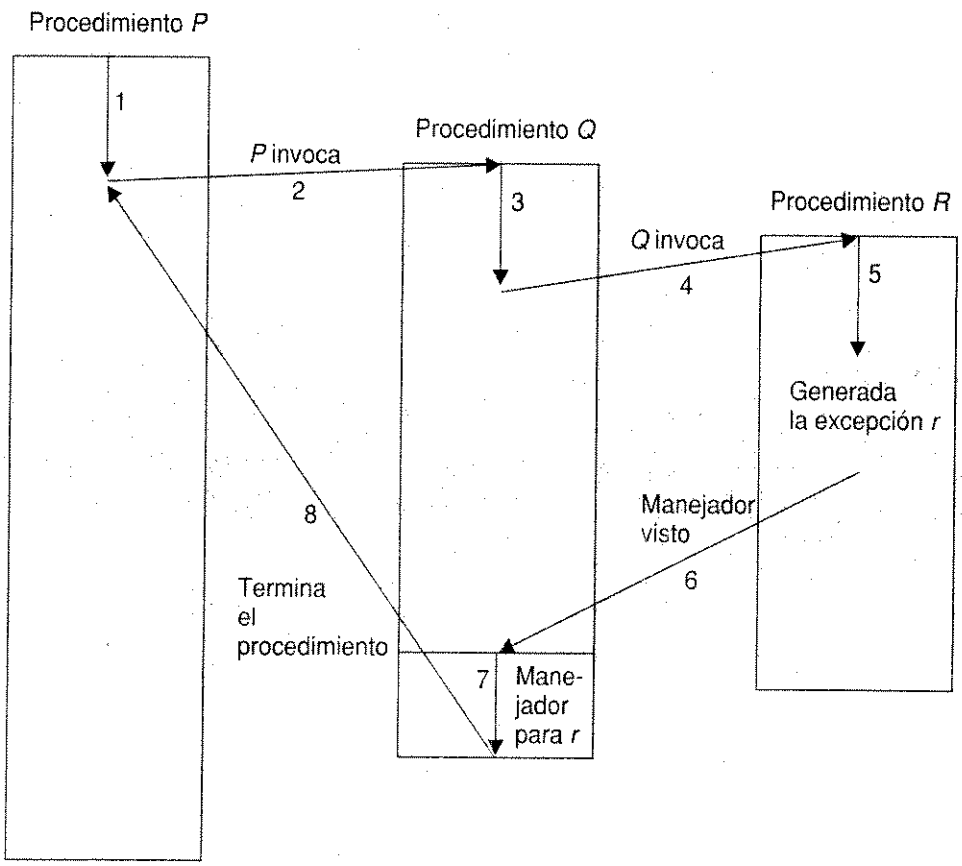


Figura 6.2. El modelo de terminación.

Si el error no es recuperable, se finaliza al invocador. El mecanismo de señales de Mesa y Real-Time Basic (Bull y Lewis, 1983) proporcionan este mecanismo.

Manejo de excepciones y sistemas operativos

Es muy probable que un programa en un lenguaje como Ada o Java acabe siendo ejecutado sobre un sistema operativo como POSIX o NT. En estos sistemas, bajo ciertas condiciones de error síncronas (como una violación de memoria o una instrucción ilegal), es habitual que el proceso en ejecución sea terminado. Sin embargo, muchos sistemas permitirán al programador un intento de recuperación del error. El modelo de recuperación soportado por POSIX, por ejemplo, permite que el programador maneje estas excepciones síncronas (mediante *señales* en POSIX), asociando a cada excepción un manejador. Este manejador es activado por el sistema cuando se detecta la condición de error. Una vez se termina el manejador, el proceso se reanuda en el punto donde fue «interrumpido», dado que POSIX soporta el modelo de reanudación.

Si un lenguaje permite el modelo de terminación, es responsabilidad del sistema de soporte de ejecución que el lenguaje capture el error y asuma la necesaria manipulación del estado del programa de modo que el programador pueda usar el modelo de terminación.

Las señales POSIX se considerarán con detalle en el Capítulo 10, dado que son efectivamente un mecanismo de concurrencia asíncrono.

6.3 Manejo de excepciones en Ada, Java y C

En esta sección se estudiará el manejo de excepciones en la programación secuencial con Ada, Java y C, lo que permitirá observar puntos de vista diferentes. El manejo de excepciones en los sistemas concurrentes se describirá en el Capítulo 10.

6.3.1 Ada

El lenguaje Ada soporta la declaración explícita de excepciones, el modelo de terminación de manejo de excepciones con propagación de excepciones no manejadas, y una forma limitada de parametrización de excepciones.

Declaración de excepciones

En Ada, las excepciones se declaran de dos formas. En primer lugar como constantes: el tipo de constante definido por la palabra clave **exception**. El siguiente ejemplo declara una excepción denominada `Valvula_Atascada`.

```
Valvula_Atascada : exception;
```

La alternativa es usar el paquete predefinido `Ada.Exceptions` (véase el Programa 6.1), que define un tipo privado llamado `Exception_Id`. Cada excepción declarada mediante **exception** lleva asociado un `Exception_Id`, que se puede obtener a través del atributo predefinido `Identity`. La identidad de la excepción `Valvula_Atascada` anterior puede verse así:

```
with Ada.Exceptions;
with Valvulas;
package Mis_Excepciones is
  Id : Ada.Exceptions.Exception_Id :=
    Valvulas.Valvula_Atascada'Identity;
end Mis_Excepciones;
```

suponiendo que `Valvula_Atascada` se declara en el paquete `Valvulas`. Observe que, al aplicar la función `Exception_Name` a `Id`, devolverá el string `Mis_Excepciones.Id`, y no `Valvula_Atascada`.

Se puede declarar una excepción en el mismo lugar que cualquier otra declaración, y esta excepción como toda declaración, tiene su alcance correspondiente.

El lenguaje dispone de varias excepciones estándar cuyo alcance es el programa completo. Estas excepciones pueden ser generadas por el sistema de soporte de ejecución en respuesta a ciertas condiciones de error, y son:

- `Constraint_Error`
  Generada, por ejemplo, cuando se intenta asignar a un objeto un valor fuera del rango declarado, cuando se intenta acceder a un elemento de un array más allá de sus límites, o cuando lo intentamos usando un puntero nulo. También se genera cuando se ejecuta cierta operación numérica predefinida que no puede proporcionar un resultado correcto dentro de la precisión declarada para los tipos reales; se incluye el conocido error de división por cero.

- `Storage_Error`
  Generada cuando el asignador del espacio de almacenamiento dinámico es incapaz de cumplir una demanda de almacenamiento porque han sido rebasadas las limitaciones físicas de la máquina.

Generación de una excepción

Así como el entorno en el que se ejecuta un programa puede generar excepciones, también puede generarlas el programa mediante la sentencia **raise** (genera, eleva). El siguiente ejemplo genera la excepción `Io_Error` (que deberá haber sido declarada previamente en el alcance) si una petición de E/S produce errores de dispositivo.

```
begin
  ...
  -- sentencias que solicitan que un dispositivo realice alguna E/S
```

Programa 6.1. Paquete Ada.Exceptions.

```
package Ada.Exceptions is
   type Exception_Id is private;
      -- cada excepción lleva asociado un identificador
   Null_Id : constant Exception_Id;
   function Exception_Name(Id : Exception_Id) return String;
      -- devuelve el nombre del objeto que
      -- tiene el identificador Exception_Id Id

   type Exception_Occurrence is limited private;
      -- cada ocurrencia de excepción tiene un identificador asociado
   type Exception_Occurrence_Access is
        access all Exception_Occurrence;
   Null_Occurrence : constant Exception_Occurrence;

   procedure Raise_Exception(E : in Exception_Id;
      Mensaje : in String := "");
      -- genera la excepción E y asocia Mensaje
      -- con la ocurrencia de la excepción

   function  Exception_Message(X : Exception_Occurrence)
      return String;
      -- permite que el manejador acceda al string pasado por
      -- Raise_Exception; si la excepción fue generada por la sentencia
      -- raise, el string contiene información sobre la excepción
      -- definida por la implementación

   procedure Reraise_Occurrence(X : in Exception_Occurrence);
      -- regenera la excepción identificada por el parámetro
      -- de la ocurrencia de la excepción

   function  Exception_Identity(X : Exception_Occurrence)
      return Exception_Id;
      -- devuelve el identificador de la excepción pasado como
      -- parámetro de la ocurrencia de la excepción

   function  Exception_Name(X : Exception_Occurrence)
      return String;
      -- igual que en Exception_Name(Exception_Identity(X)).

   function  Exception_Information(X : Exception_Occurrence)
      return String;
      -- igual que en Exception_Message(X) pero contiene más detalles
      -- si el mensaje proviene de la implementación

   procedure Save_Occurrence(Objetivo : out Exception_Occurrence;
      Fuente : in Exception_Occurrence);
      -- permite asignar a objetos del tipo Exception_Occurrence

   function Save_Occurrence(Fuente : Exception_Occurrence)
      return Exception_Occurrence_Access;
      -- permite asignar a objetos del tipo Exception_Occurrence
private
   ... -- no especificado por el lenguaje
end Ada.Exceptions;
```

```
   if Io_Device_In_Error then
      raise Io_Error;
   end if;
   ...
end;
```

Observe que no es imprescindible la parte `else` en la sentencia `if`, puesto que *no* se devuelve el control a la sentencia que sigue a `raise`.

Si se hubiera declarado `Io_Error` como `Exception_Id`, hubiera sido necesario generar la excepción mediante el procedimiento `Ada.Exceptions.Raise_Exception`. Esto hubiera permitido pasar un string como parámetro de la excepción.

Cada generación de una excepción se denomina una **ocurrencia** de una excepción, y se representa mediante un valor del tipo `Ada.Exceptions.Exception_Ocurrence`. Cuando se maneja una excepción, se puede obtener el valor de `Exception_Ocurrence` y usarse para conseguir más información sobre la causa de la excepción.

Manejo de excepciones

Como se vió en el Capítulo 3, cada bloque Ada (y cada subprograma, sentencia accept o tarea) pueden contener un conjunto opcional de manejadores de excepciones, que se declararán al final del bloque (o subprograma, sentencia accept o tarea). Cada manejador es una secuencia de sentencias precedida por la palabra clave **when**, un parámetro opcional al que se le asignará la identidad de la ocurrencia de la excepción, los nombres de las excepciones que serán servidas por el manejador, y el símbolo =>. A modo de ejemplo, el siguiente bloque declara tres excepciones y proporciona dos manejadores.

```
declare
   Sensor_Alto, Sensor_Bajo, Sensor_Muerto : exception;
   -- otras declaraciones
begin
   -- sentencias que pudieran ocasionar la generación
   -- de las excepciones anteriores
exception
   when E: Sensor_Alto | Sensor_Bajo =>
      -- toma alguna acción correctiva
      -- si tanto Sensor_Alto como Sensor_Bajo son generadas.
      -- E contiene la ocurrencia de la excepción
   when Sensor_Muerto =>
      -- hace sonar la alarma si se genera la excepción
      -- Sensor_Muerto
end;
```

Ada proporciona un nombre de manejador **when others** para no tener que citar todos los otros nombres de excepciones posibles. Sólo se permite en el último lugar de la lista de manejadores, y representa las excepciones no enumeradas previamente en el conjunto actual de manejadores. El bloque del siguiente ejemplo imprime información sobre la excepción, y hace sonar una alarma si se genera cualquiera excepción salvo `Sensor_Bajo` o `Sensor_Alto` (incluyendo `Sensor_Muerto`).

```
declare
  Sensor_Alto, Sensor_Bajo, Sensor_Muerto : exception;
  -- otras declaraciones
  use Text_Io;
begin
  -- sentencias que pudieran ocasionar la generación
  -- de las sentencias anteriores
exception
  when Sensor_Alto | Sensor_Bajo =>
     -- toma alguna acción correctiva
  when E: others =>
     Put(Exception_Name(E));
     Put_Line(" atrapada. Se dispone de la siguiente información ");
     Put_Line(Exception_Information(E));
     -- haz sonar la alarma
end;
```

Una excepción generada en un manejador de excepción no puede ser manejada por ése u otros manejadores del mismo bloque (o procedimiento). En lugar de esto, se termina el bloque y se busca un manejador en el bloque exterior o en el punto de llamada de un subprograma.

Propagación de excepciones

Si no hubiera ningún manejador en el bloque (o subprograma o sentencia accept), se genera de nuevo la excepción; es decir, Ada **propaga** excepciones. En el caso de un bloque se generará en el bloque superior, o subprograma. En el caso de un subprograma, se genera la excepción en su punto de invocación.

Un malentendido común sobre Ada es el de que se pueden proporcionar manejadores de excepciones en la sección de inicialización de los paquetes, y así poder manejar las excepciones que se generan en la ejecución de sus subprogramas anidados. Una excepción generada y *no* manejada por un subprograma se propaga hacia el invocador del subprograma. Es por esto por lo que tal excepción sólo será manejada por el código de inicialización si fue éste el que llamó al subprograma. El siguiente ejemplo muestra este punto.

```
package Control_Temperatura is
  subtype Temperatura is Integer range 0 .. 100;
```

```
   Sensor_Muerto, Actuador_Muerto : exception;

   procedure Pon_Temperatura(New_Temp : in Temperatura);
   function Lee_Temperatura return Temperatura;
end Control_Temperatura ;

package body Control_Temperatura is
   procedure Pon_Temperatura(Nueva_Temp : in Temperatura) is
   begin
      -- informa al actuador de la nueva temperatura
      if Sin_Respuesta then
         raise Actuador_Muerto;
      end if;
   end Pon_Temperatura;

   function Lee_Temperatura return Temperatura is
   begin
      -- lee sensor
      if Sin_Respuesta then
         raise Sensor_Muerto;
      end if;
      -- calcula la temperatura
      return Leyendo;
   exception
      when Constraint_Error =>
         -- la temperatura ha ido fuera de su rango esperado;
         -- toma alguna acción apropiada
   end Lee_Temperatura;
begin
   -- inicialización del paquete
   Pon_Temperatura(Lectura_Inicial);
exception
   when Actuador_Muerto =>
      -- toma alguna acción correctora
end Control_Temperatura;
```

En este ejemplo, el procedimiento `Pon_Temperatura`, que fue llamado desde el exterior del paquete, se invoca también durante la inicialización del paquete. Este procedimiento podría generar la excepción `Actuador_Muerto`. El manejador de `Actuador_Muerto` dado en la sección de inicialización del paquete *sólo atrapará la excepción cuando el procedimiento haya sido llamado desde el código de inicialización.* No atrapará excepciones producidas cuando se llame el procedimiento desde el exterior del paquete.

Si el código que inicializó un cuerpo de un paquete en sí mismo generó una excepción que no fue manejada localmente, ésta se propagará hasta el lugar en que se hizo visible el paquete.

Últimas voluntades

Una excepción también puede ser propagada por un programa que relance la excepción en el manejador local. La sentencia **raise** (o el procedimiento `Ada.Exceptions.Reraise_Ocurrence`) consigue el efecto de regenerar la última excepción (o la ocurrencia de excepción específica). Esto es útil en la programación de las **últimas voluntades**. A menudo, se da el caso de que siendo el significado de una excepción desconocido para el manejador local, debe ser manejado para poner en orden cualquier asignación parcial de recursos que pudiera haber ocurrido antes de la generación de la excepción. Por ejemplo, considere un procedimiento que reserva varios dispositivos. Cualquier excepción generada durante la rutina de reserva que se hubiese propagado hacia el invocador pudiera haber dejado sin asignar algunos dispositivos. El asignador, entonces, deseará «desasignar» los dispositivos asociados en el caso de que no haya sido posible realizar la reserva completa. El siguiente ejemplo ilustra esta aproximación.

```
subtype Dispositivos is Integer range 1 .. Max;

procedure Reserva (Cantidad : Dispositivos) is
begin
   -- solicita que sea reservado cada dispositivo por turno
   -- anotando qué peticiones son concedidas
exception
   when others =>
      -- desasigna los dispositivos reservados
      raise; -- regenera la excepción
end Reserva;
```

Usado de esta forma, puede decirse que el procedimiento implementa la propiedad de atomicidad de fallo de una acción atómica; o se reservan todos los recursos, o ninguno (véase el Capítulo 10).

Como un ejemplo a mayores, considere un procedimiento que establece las posiciones de elementos como *slats* y *flaps*, que varían la geometría del ala de un avión con vuelo asistido (fly-by-wire) durante su fase de aterrizaje. Estos elementos alteran el grado de empuje ascensional del avión; una configuración asimétrica de las alas en el despegue (o aterrizaje) podría ocasionar inestabilidades en la aeronave. Suponiendo que las posiciones iniciales son simétricas, el siguiente procedimiento asegura que permanezcan simétricas incluso si se genera una excepción como resultado de un fallo del sistema físico o a causa de un error de programa.[1]

[1] Como es lógico, éste es un ejemplo rudimentario para mostrar la forma de proceder, que no es necesariamente la que se tomaría en la práctica.

```
procedure Configuracion_Alas ( -- parametros relevantes) is
begin
   -- configura adecuadamente slats y flaps;
   -- podrían generarse excepciones
exception
   when others =>
      -- asegura que las posiciones son simétricas y
      -- regenera la excepción para indicar
      -- un aterrizaje sin slats o sin flaps
      raise;
end Configuracion_Alas;
```

Ada permite otro mecanismo más de últimas voluntades. En esencia, se declarará en el procedimiento una variable controlada adicional. Todas las variables controladas tienen procedimientos de finalización que son invocados automáticamente cuando se sale de su alcance. Con este procedimiento se puede asegurar cierta condición en presencia de excepciones: en el caso anterior, por ejemplo, la posición simétrica de los *slats* y los *flaps*. Véase la Sección 4.4.1.

Excepciones generadas durante la elaboración de declaraciones

Se puede generar una excepción durante la elaboración de la parte declarativa de un subprograma, bloque, tarea o paquete (por ejemplo al inicializar una variable con un valor fuera de su rango especificado). En general, cuando esto ocurre se abandona la parte declarativa y se genera una excepción en el bloque, subprograma, tarea o paquete que provocó la elaboración en primer lugar.

La definición completa de las reglas de manejo de excepciones es un poco más complicada que lo que se ha indicado aquí. Se remite al lector al manual *Ada 95 Reference Manual*, Capítulo 11, para los detalles completos.

Supresión de excepciones

Hay un aforismo que se ha hecho muy popular entre los programadores durante la última década, en el mundo anglosajón; normalmente toma la forma siguiente: «¡El almuerzo gratis no existe!».[2] Uno de los requisitos para los mecanismos de manejo de excepciones era que no debían introducir sobrecargas en la ejecución a menos que se presentaran excepciones (R3). A la vista de los mecanismos que proporciona Ada, parece que se cumple con este requisito. Sin embargo, siempre habrá alguna sobrecarga asociada a la detección de cualquier condición de error posible.

Ada, por ejemplo, proporciona una excepción estándar denominada `Constraint_Error`, que se genera cuando se emplea un puntero nulo, cuando se sobrepasan los límites de un array, o cuando se asigna a un objeto un valor por encima del rango permitido. Para atrapar estas

[2] N. de los T.: En el original «*there is no such thing as a free lunch*!».

condiciones de error, el compilador deberá generar el código correspondiente. Por ejemplo, cuando se pretende acceder a un objeto mediante un puntero, el compilador, en ausencia de cualquier análisis de control de flujo (o de apoyo por parte del hardware), insertará código para comprobar si el puntero es nulo antes de acceder al objeto. Aunque permanece oculto al programador, este código se ejecutará incluso cuando no se lance excepción alguna. Si un programador emplea muchos punteros, provocará una sobrecarga significativa, tanto en cuanto al tiempo de ejecución como al tamaño del código generado. Además, la presencia de este código pudiera requerir su comprobación durante cualquier proceso de validación, y esto también podría ser difícil de hacer.

El lenguaje Ada reconoce el sobrecoste impuesto por el entorno de ejecución sobre ciertas aplicaciones debido a las excepciones contempladas. En consecuencia, ofrece la posibilidad de suprimir estas comprobaciones. Esto se consigue con la directiva (pragma) `Suppress`, que elimina todo un abanico de comprobaciones en tiempo de ejecución. La directiva afecta sólo a la unidad de compilación en que aparece. Claro está que si se suprime una comprobación de error en ejecución y posteriormente aparece dicho error, el lenguaje considera que el programa es «erróneo», y no está definido cuál debe ser el comportamiento posterior del programa.

Un ejemplo completo

El siguiente paquete muestra el empleo de excepciones en un tipo de dato abstracto que implementa una `Pila` simple. Se eligió este ejemplo para poder realizar la especificación y el cuerpo completos sin dejar hueco a la imaginación del lector.

El paquete es genérico y, consecuentemente, puede ser instanciado para tipos diferentes.

```
generic
  Talla : Natural := 100;
  type Item is private;
package Pila is

  Pila_Llena, Pila_Vacia : exception;

  procedure Mete(X:in Item);
  procedure Saca(X:out Item);

end Pila;


package body Pila is

  type Indice_Pila is new Integer range 0..Talla-1;
  type Array_Pila is array(Indice_Pila) of Item;
  type Pila is
```

```
    record
      A : Array_Pila;
      Pa : Indice_Pila := 0; -- puntero al array
    end record;
  Pl : Pila;

  procedure Mete(X:in Item) is
  begin
    if Pl.Pa = Indice_Pila'Primero then
      raise Pila_Llena;
    end if;
    Pl.Pa :=Pl.Pa + 1;
    Pl.A(Pl.Pa) := X;
  end Mete;

  procedure Saca(X:out Item) is
  begin
    if Pl.Pa = 0 then
      raise Pila_Vacia;
    end if;
    X := Pl.A(Pl.Pa);
    Pl.Pa := Pl.Pa - 1;
  end Saca;

end Pila;
```

Podría emplearse así:

```
with Pila;
with Text_Io;
procedure Usa_Pila is
  package Pila_Enteros is new Pila(Item => Integer);
  X : Integer;
  use Pila_Enteros;
begin
  ...
  Mete(X);
  ...
  Saca(X);
  ...
exception
  when Pila_Llena =>
    Text_Io.Put_Line("!Desbordamiento de Pila!");
```

```
   when Pila_Vacia =>
      Text_Io.Put_Line("!Pila vacía!");
end Usa_Pila;
```

Dificultades del modelo de excepciones de Ada

Aunque el lenguaje Ada provee un completo conjunto de posibilidades para el manejo de excepciones, existen ciertas dificultades a la hora de emplearlas.

(1) **Excepciones y paquetes**. En la especificación de un paquete se declaran las excepciones que pueden ser generadas, junto a los subprogramas que pueden ser invocados. Sin embargo, no está claro qué excepciones pueden ser generadas en cada subprograma. Si los usuarios del paquete no conocen su implementación, deberán intentar asociar los nombres de las excepciones con los nombres de los subprogramas. En el ejemplo anterior de la pila, el usuario podría concluir que la excepción `Pila_Llena` se genera desde el procedimiento `Saca`, ¡y no desde `Mete`! En paquetes grandes, puede no estar claro qué excepciones puede generar cada subprograma. Así, el programador deberá recurrir a citar todas las posibles excepciones cada vez que se invoca un subprograma, o emplear **when others**. Los escritores de paquetes debieran indicar mediante comentarios qué excepciones pueden ser generadas desde cada subprograma.

(2) **Paso de parámetros**. Ada no permite pasar como parámetros a los manejadores un juego completo de parámetros, sino sólo una cadena de caracteres, lo cual es un inconveniente si es preciso pasar algún objeto de cierto tipo.

(3) **Alcance y propagación**. Es posible propagar excepciones más allá del alcance de su declaración. Tales excepciones sólo podrán ser atrapadas por **when others**. Sin embargo, pueden retornar a su alcance de nuevo si se propagan más arriba aún en la cadena dinámica de invocaciones. Este efecto es desconcertante, aunque probablemente inevitable cuando se emplea un lenguaje estructurado en bloques con propagación de excepciones.

6.3.2 Java

Java es similar a Ada en cuanto a que permite el modelo de terminación de manejo de excepciones. Sin embargo, las excepciones de Java, al contrario que en Ada, se integran en el modelo orientado al objeto.

Declaración de excepciones

En Java, todas las excepciones son subclases de la clase predefinida `java.lang.Throwable`. El lenguaje también define otras clases; por ejemplo: `Error`, `Exception`, y `RuntimeException`. En la Figura 6.3 se muestra la relación entre ellas. A lo largo de este libro, el término excepción Java se emplea para hacer referencia a cualquier clase derivada de `Throwable`, y no sólo a aquéllas derivadas de `Exception`.

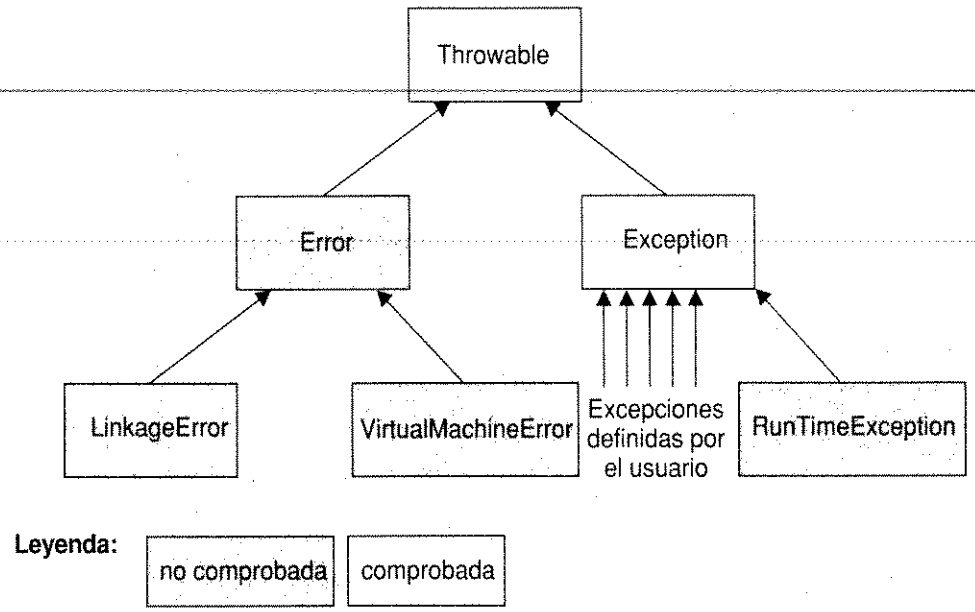


Figura 6.3. La jerarquía de clases predefinida de `Throwable`.

Los objetos derivados de `Error` describen los errores internos y el agotamiento de recursos del sistema de soporte de ejecución de Java. Aunque estos errores tienen obviamente un gran impacto sobre el programa, bien poco puede hacer éste cuando se lanzan (generan) estos errores, puesto que no es posible contemplar premisa alguna sobre la integridad del sistema.

Los objetos derivados de la jerarquía de `Exception` representan los errores que pueden manejar los programas y, potencialmente, lanzar. Las excepciones `RuntimeExceptions` son generadas por el sistema de soporte de ejecución como resultado de un error en el programa. Entre éstas se incluyen errores como aquéllos que provienen de una conversión errónea (`ClassCastException`), errores al sobrepasar los límites de un array (`IndexOutOfBoundException`), un acceso mediante un puntero nulo (`NullPointerException`), una división entera por cero (`ArithmeticException`), etc.

Los objetos arrojables (throwable) derivados de `Error` o `RuntimeException` se denominan excepciones **no comprobadas**. Esto quiere decir que el compilador Java no espera que sean identificadas en la clausula `throws` de la declaración de un método (véase más adelante).

Considere, por ejemplo, el controlador de temperatura dado en la Sección 6.3.1, que podría escribirse como sigue. Primero, hay que proveer una clase para representar un error de restricciones para los subtipos enteros. Aunque dado que es una condición de error general, podría venir dada por una biblioteca. Las variables públicas de la clase contendrán información sobre la causa del error.

```
public class ErrorRestriccionEntero extends Exception
{

  private int rangoInferior, rangoSuperior, valor;

  public ErrorRestriccionEntero(int I, int S, int V)
```

```
  {
     super(); // llama al constructor de la clase padre
     rangoInferior = I;
     rangoSuperior = S;
     valor = V;
  }
  public String getMessage()
  {
     return ("Error Restricción Entero: Rango Inferior " +
            java.lang.Integer.toString(rangoInferior) + " Rango Superior " +
            java.lang.Integer.toString(rangoSuperior) + " Hallado " +
            java.lang.Integer.toString(valor));
  }
};
```

Ahora puede describirse el tipo temperatura.

```
import bibliotecaExcepciones.ErrorRestriccionEntero;

public class Temperatura
{
  private int T;

  public Temperatura(int inicial) throws ErrorRestriccionEntero
   // constructor
  {
     ...;
  }
  public void ponValor(int V) throws ErrorRestriccionEntero
  {
     ...;
  };
  public int leeValor()
  {
     return T;
  };
  // tanto el constructor como leeValor pueden lanzar
     // ErrorRestriccionEntero
};
```

En el código anterior, las funciones miembro que pueden lanzar la excepción `ErrorRestriccionEntero` deben etiquetarse formalmente. Esto contrasta con la aproximación de Ada, que descansa sobre comentarios.

Ahora podemos definir la clase `ControladorTemperatura`. Ésta declara un tipo para representar la excepción que indica el fallo en el actuador. En este caso no se pasan datos en el objeto.

```
class ActuadorMuerto extends Exception
{
  public String getMessage()
  {
     return ("Actuador Muerto");
  }
};
class ControladorTemperatura
{
  public ControladorTemperatura(int T)
         throws ErrorRestriccionEntero
  {
     temperaturaActual = new Temperatura(T);
  };

  Temperatura temperaturaActual;

  public void ponTemperatura(int T)
         throws ActuadorMuerto, ErrorRestriccionEntero
  {
     temperaturaActual.ponValor(T);
  };
     int leeTemperatura()
  {
        return temperaturaActual.leeValor();
  }
};
```

En general, cada función deberá especificar una lista de excepciones lanzables comprobadas **throws** A, B, C, en cuyo caso la función podría lanzar cualquier excepción de esta lista y cualquiera de las excepciones no comprobadas. A, B, y C podrían ser subclases de `Exception`. Si una función intentara lanzar una excepción que no constara en su lista de excepciones lanzables, aparecería un error de compilación.

Lanzar una excepción

Al hecho de generar una excepción en Java se le denomina **lanzar** o arrojar la excepción. Por ejemplo, observe la implementación completa de la clase `Temperatura` bosquejada en la sección anterior.

```
import bibliotecaExcepciones.ErrorRestriccionEntero;
class Temperatura
{
  int T;

  void comprueba(int valor) throws ErrorRestriccionEntero
  {
     if(valor > 100 || valor < 0) {
      throw new ErrorRestriccionEntero(0, 100, valor);
     };
  }

  public Temperatura(int inicial) throws ErrorRestriccionEntero
     // constructor
  {
     comprueba(inicial);
     T = inicial;
  }

  public void ponValor(int V) throws ErrorRestriccionEntero
  {
     comprueba(V);
     T =V;
  };

  public int leeValor()
  {
     return T;
  };
};
```

Aquí **throw new** `ErrorRestriccionEntero(0, 100, valor)` crea y lanza un objeto del tipo (`ErrorRestriccionEntero`) con los valores adecuados de sus variables de instancia.

Manejo de excepciones

En Java, una excepción sólo puede ser manejada desde el interior de un **bloque try** (intenta). Cada manejador se especifica mediante una sentencia `catch` (atrapa). Vea el siguiente fragmento de código:

```
// dado ControladorTemperatura

try {
```

```
  ControladorTemperatura TC = new ControladorTemperatura(20);

  TC.ponTemperatura(100);
  // sentencias que manipulan la temperatura
}
catch (ErrorRestriccionEntero error) {
  // excepción atrapada, imprime el mensaje de error
  // sobre la salida estándar
  System.out.println(error.getMessage());
}
catch (ActuadorMuerto error) {
  System.out.println(error.getMessage());
}
```

La sentencia **`catch`** es similar a una declaración de función, cuyo parámetro identifica el tipo de excepción a atrapar. En el interior del manejador, el nombre del objeto se comporta como una variable local.

Un manejador con parámetro de tipo `T` atrapará un objeto lanzado de tipo `E` si:

(1) `T` y `E` son el mismo tipo.

(2) `T` es una clase padre (super clase) de `E` en el punto donde se lanza.

Es este último punto el que hace que el manejo de excepciones de Java sea muy potente. En el ejemplo anterior, dos excepciones se derivan de la clase `Exception`: `ErrorRestriccionEntero` y `ActuadorMuerto`. En el siguiente bloque `try` se atrapan ambas excepciones:

```
try {
  // sentencias que pudieran generar la excepción
  // ErrorRestriccionEntero o ActuadorMuerto
}
catch(Exception E) {
  // éste manejador atrapará todas las excepciones de tipo
  // Exception y cualquier tipo derivado; pero desde el
  // interior del manejador sólo son accesibles los métodos de E
}
```

Por supuesto, una llamada a `E.getMessage` llamará a la rutina adecuada para el tipo del objeto lanzado. Ciertamente, `catch(Exception E)` equivale al **`when others`** de Ada.

Propagación de excepciones

Java soporta propagación de excepciones. Como ocurría con Ada, si no se encuentra ningún manejador de excepción en el contexto de llamada de una función, el contexto de llamada se da por terminado y se busca un manejador en su contexto de llamada superior.

Últimas voluntades

En la Sección 6.3.1 se discutía cómo podía emplearse **when others** para programar las últimas voluntades en Ada. Obviamente, un `catch(Exception E)` también puede usarse para obtener este efecto. Sin embargo, Java también soporta una clausula **finally** como parte de una sentencia try. El código asociado a esta clausula tiene garantizada su ejecución ocurra lo que ocurra en la sentencia try, sean las excepciones lanzadas, atrapadas, o propagadas; o incluso si no se lanza excepción alguna.

```
try
{
   ...
}
catch(..)
{
...
}
finally
{
  // código que será ejecutado bajo cualquier circunstancia
}
```

En Ada puede lograrse el mismo efecto usando variables controladas y finalización (véase la Sección 4.4.1).

Un ejemplo completo

El siguiente ejemplo en Java es el equivalente del de la pila desarrollado anteriormente para Ada. Hay dos cuestiones interesantes que comentar sobre este ejemplo.

Primeramente, la pila es genérica, en el sentido de que puede pasarse cualquier objeto como argumento de mete. Es decir, no es preciso instanciar la pila (como en el caso de Ada). Además, cada pila puede manejar más de un tipo de objeto (a diferencia del caso de Ada). Obviamente, esto es así porque la pila Java es en realidad una pila de referencias a objetos. Sería posible programar esto en Ada explícitamente.

La segunda cuestión es que en el ejemplo de Java sólo se manejan objetos, y no tipos primitivos como los enteros. Ada, por contra, maneja todos los tipos.

```
public class ExcepcionPilaVacia extends Exception
{
    public ExcepcionPilaVacia() {
    }
}
```

```
public class ExcepcionPilaLlena extends Exception
{
    public ExcepcionPilaLlena() {
    }
}
public class Pila {

  public Stack(int capacidad)
  {
    arrayPila = new Object[capacidad];
    indicePila = 0;
    capacidadPila = capacidad;
  }

  public void mete(Object item) throws ExcepcionPilaLlena
  {
    if(indicePila == capacidadPila) throw new ExcepcionPilaLlena();
    arrayPila[indicePila++] = item;
  }

  public synchronized Object saca() throws ExcepcionPilaVacia
  {
    if(indicePila == 0) throw new ExcepcionPilaVacia();
    return arrayPila[--indicePila];
  }
  protected Object arrayPila[];
  protected int indicePila;
  protected int capacidadPila;
}
```

Las clases anteriores pueden usarse como sigue:

```
import Pila;
import ExcepcionPilaLlena;
import ExcepcionPilaVacia;

public class UsaPila
{

  public static void main(...)
  {
    Pila P = new Pila(100);

    try {
```

```
            ...
            P.mete(algunObjeto);
            ...
            algunObjeto = P.saca();
            ...
        }
        catch (ExcepcionPilaLlena F) ...
        catch (ExcepcionPilaVacia E) ... ;
    }
}
```

Hay que decir que Java proporciona una clase estándar `Stack` en su paquete `util`.

6.3.3 C

En el lenguaje C no hay definido mecanismo alguno de manejo de excepciones. Tal tipo de omisión limita claramente la utilidad del lenguaje para la programación estructurada de sistemas fiables. Aun así, es posible proporcionar algún mecanismo de manejo de excepciones mediante el entorno de macros del lenguaje. Para mostrar esta aproximación, se considerará la implementación de un entorno de simple macros para el manejo de excepciones al estilo de Ada. Esta aproximación se basa en la dada por Lee (1983).

Para implementar en C un modelo de terminación, es preciso guardar el estatus de los registros del programa y demás información a la entrada del dominio de una excepción, para restaurarlos si acaece una excepción. Tradicionalmente se ha asociado C con UNIX, de modo que para este fin se pueden utilizar las primitivas de POSIX `setjmp` y `longjmp`. La rutina `setjmp` almacena el estatus del programa y devuelve un 0; la rutina `longjmp` restaura el estatus del programa y provoca que el programa abandone su ejecución actual reiniciándose desde la posición donde se invocó `setjmp`. En esta ocasión, `setjmp` devuelve el valor pasado por `longjmp`. Se precisa el siguiente código:

```
/* comienza el dominio de la excepción */

typedef char *excepcion;
  /* un tipo de puntero a una cadena de caracteres */
excepcion error ="error";
  /* la representación de una excepción denominada "error" */

if((excepcion_actual = (excepcion) setjmp(area_almacen)) == 0) {
  /* guarda los registros y demás en area_almacen */
  /* se devuelve 0*/

 /* la region protegida */
```

```
    /* cuando se identifica una excepcion "error" */
    longjmp(area_almacen, (int) error);
    /* no vuelve */

  }
  else {
    if(excepcion_actual == error) {
      /* manejador para "error" */
    }
    else {
      /* regenerar la excepción en el dominio del entorno */
    }
  }
```

El código anterior es claramente difícil de comprender. Sin embargo, puede definirse un conjunto de macros para ayudar a estructurar el programa.

```
#define NUEVA_EXCEPCION(nombre) ...
  /* código para declarar una excepción */
#define COMIENZO ...
  /* código para entrar en el dominio de una excepción */
#define EXCEPCION ...
  /* código para el comienzo de los manejadores de excepciones */
#define FIN ...
  /* código para dejar el dominio de una excepción */
#define LANZA(nombre) ...
  /* código para generar una excepción */
#define CUANDO(nombre) ...
  /* código para el manejador */
#define OTROS ...
  /* código para el manejador que captura cualquier excepción */
```

Considere ahora el siguiente ejemplo:

```
  NUEVA_EXCEPCION(sensor_alto);
  NUEVA_EXCEPCION(sensor_bajo);
  NUEVA_EXCEPCION(sensor_muerto);
  /* otras declaraciones */

COMIENZO
  /* sentencias que podrían provocar la generacion de las */
  /* excepciones anteriores; por ejemplo */
  LANZA(sensor_alto);
```

```
EXCEPCION
  CUANDO(sensor_alto)
     /* tomar alguna acción correctora */
  CUANDO(sensor_bajo)
     /* tomar alguna acción correctora */
  CUANDO(OTROS)
     /* hacer sonar una alarma */
FIN;
```

Lo anterior proporciona un modelo de terminación simple similar al de Ada.

6.4 Manejo de excepciones en otros lenguajes

Dentro de este libro, la atención se enfoca primeramente sobre los lenguajes Ada, Java, C y occam2. Por supuesto que existen muchos otros lenguajes que se emplean en las aplicaciones de tiempo real, pero sería imposible abarcarlos todos dentro de un solo volumen. Este libro sólo pretende discutir las características particulares de otros lenguajes cuando sean distintas o importantes. Consecuentemente, en esta sección sólo se revisa brevemente el manejo de excepciones en CHILL, CLU, Mesa y C++.

6.4.1 CHILL

CHILL es similar a Ada en cuanto a que las excepciones forman parte de la definición del lenguaje, y en cuanto a que ambos soportan el modelo de terminación de manejo de excepciones. Sin embargo, las otras posibilidades que ofrece son significativamente diferentes de las de Ada. CHILL posee una sintaxis para sus nombres de tipo y sus declaraciones de objetos ligeramente diferente de las de los otros lenguajes que se estudian con detalle en este libro; por eso, se utilizará una sintaxis al estilo de Ada en los ejemplos que se proporcionan a continuación.

Según se indicó en la Sección 6.2.1, el dominio de un manejador de excepciones en CHILL es la sentencia, el bloque o el proceso. La principal diferencia entre CHILL y Ada es, consecuentemente, que en CHILL puede añadirse un manejador de excepciones al final de cada sentencia. En lugar de la palabra clave `exception`, CHILL usa `on` y `end` para delimitar un manejador. Entre medias, la sintaxis es bastante similar a la de una sentencia `case`. El siguiente ejemplo muestra un manejador de excepciones en CHILL.

```
  -- sintaxis al estilo Ada para mostrar los conceptos
  -- no es sintaxis CHILL válida
declare
  subtype temperatura is integer range 0 .. 100;
  A : temperatura;
  B,C : integer;
```

```
begin
  ...
  A:=B +C on
            (overflow): .....;
            (rangefail): .....;
            else ....;
          end;
  ...
end on
    -- manejadores de excepciones para el bloque
    (overflow): .....;
    (rangefail): .....;
    else ....;
  end;
```

CHILL define varias excepciones estándar similares a las de Ada. En el ejemplo anterior, puede generarse `overflow` (un desbordamiento aritmético) cuando sumamos `B` y C; además, pudiera ocurrir `rangefail` (excepción de fallo en el rango) si el resultado que se asigna a `A` se encuentra fuera del rango de 0 a 100. La opción `else` es equivalente a la opción de Ada `when others`.

La asociación entre una excepción y su manejador es estática, y en consecuencia deberá ser conocida en tiempo de compilación. Por ello, no es preciso predeclararlas, pero pueden ser devueltas desde un procedimiento mediante una declaración apropiada en el encabezamiento del procedimiento. De nuevo, usando una sintaxis tipo Ada:

```
procedure mete(x:in item) exceptions (pila_llena);
procedure saca(x:out item) exceptions (pila_vacia);
```

La regla para determinar estáticamente un manejador para una excepción E que aparece durante la ejecución de la sentencia S, es:

- debe estar añadida a S, o
- debe estar añadida a un bloque que directamente encierra S, o
- debe estar añadida a un bloque que directamente encierra S, o
- el procedimiento que lo encierra directamente debe haber definido E en su especificación, o
- se encuentra añadido al proceso que directamente encierra S.

Cuando no se puede encontrar la excepción empleando las reglas anteriores, el programa se encuentra en estado de error. No hay propagación de excepciones.

6.4.2 CLU

CLU es un lenguaje experimental que, aun no siendo un lenguaje pensado para «tiempo real», posee ciertas características interesantes, una de las cuales es su mecanismo de manejo de excepciones (Liskov y Snyder, 1979). Es similar a C++ y CHILL en cuanto a que cada procedimiento declara las excepciones que puede generar. También permite el paso de parámetros hacia el manejador.

Por ejemplo, considere la función `suma_stream`, que lee una secuencia de enteros con signo en decimal de un stream de caracteres, y devuelve la suma de estos enteros. Son posibles las siguientes excepciones: `overflow` (desbordamiento; la suma del número cae fuera del rango de enteros implementado); `unrepresentable_integer` (entero no representable; un número del stream cae fuera del rango de enteros implementado); y `bad_format` (formato inválido; el stream contiene un campo no entero). Con las dos últimas excepciones se pasa la cadena errónea al manejador.

```
suma_stream= proc(s : stream) return(int)
             signals
                (overflow,
                unrepresentable_integer(string),
                bad_format(string)
                )
```

Como con CHILL, no hay propagación de excepciones;

```
x = suma_stream
    except
       when overflow:
          S1
       when unrepresentable_integer(f : string):
          S2
       when bad_format(f:string):
          S3
    end
```

donde `S1`, `S2` y `S3` son secuencias arbitrarias de sentencias.

6.4.3 Mesa

En Mesa, las excepciones se denominan **señales**. Son dinámicas en el sentido de que pueden ser propagadas, y sus manejadores se adhieren al modelo híbrido. Al igual que ocurría con las excepciones de Ada, pueden declararse señales, pero, a diferencia de Ada, su declaración es similar al tipo de un procedimiento más que a la declaración de una constante. En consecuencia, pueden tomar parámetros y devolver valores. El cuerpo de un procedimiento señal es, como no,

el manejador de excepciones. A diferencia de los cuerpos de los procedimientos que se ligan *estáticamente* al procedimiento en el momento de la compilación, los manejadores de excepciones se ligan *dinámicamente* a la señal en el momento de la ejecución.

Los manejadores pueden asociarse a bloques, como ocurría con Ada, C++ y CHILL. Como en Ada, los procedimientos y funciones de Mesa no pueden especificar qué señales van a retornar; sin embargo, se puede asociar un manejador con una llamada a un procedimiento, proporcionando un efecto similar.

Hay dos tipos de declaración de señal: mediante la palabra clave *signal* y mediante *error*. Las señales declaradas como error no pueden ser reanudadas por sus correspondientes manejadores.

6.4.4 C++

C++ es similar a Java, excepto en que no requiere una declaración explícita de excepciones. Más bien, puede lanzarse como excepción cualquier instancia de una clase. No existen excepciones predefinidas.

En general, cada función puede especificar

- Una lista de objetos lanzables, `throw (A, B, C)` (en cuyo caso la función podrá lanzar cualquier objeto de la lista).
- Una lista de objetos lanzables vacía, `throw ()` (en cuyo caso la función no lanzará objeto alguno).
- Sin lista de objetos lanzables (en cuyo caso la función podrá lanzar cualquier objeto).

Si una función intenta lanzar un objeto no permitido por su lista de objetos lanzables, se llamará automáticamente a la función `unexpected`. Por defecto, `unexpected` llamará a la función `terminate`, cuyo comportamiento por defecto es abortar el programa. Es posible sobreseer, en la llamada, las operaciones por defecto de ambas funciones.

```
typedef void(*PFV)();

PFV set_unexpected(PFV nuevo_manejador);
/* establece la acción por defecto a nuevo_manejador y */
/* devuelve la acción previa */

PFV set_terminate(PFV nuevo_manejador);•178 EXCEPCIONES Y MANEJO DE EXCEPCIONES
/* establece la acción por defecto a nuevo_manejador y */
/* devuelve la acción previa */
```

Como Java, C++ tiene su sentencia **`catch`**, cuyo parámetro identifica el tipo del objeto a atrapar.

Un manejador con tipo de parámetro `T` atrapará un objeto lanzado de tipo `E` si:

(1) T y E son del mismo tipo.

(2) T es un tipo puntero y E es un tipo puntero que puede convertirse a T mediante una conversión de tipo estándar de C++ en el punto de lanzamiento.

(3) T es un tipo puntero y E es un objeto del tipo al que apunta T. En este caso, se crea un puntero al objeto lanzado; tanto el puntero como el objeto persistirán hasta que se salga del manejador de la excepción.

(4) T es una clase base de E en el punto de lanzamiento.

Se pueden construir jerarquías de excepciones como en Java.

6.5 Bloques de recuperación y excepciones

En el Capítulo 5, se presentó la noción de bloque de recuperación como un mecanismo para la programación tolerante a fallos. Su principal ventaja sobre los mecanismos de recuperación de errores hacia delante es que puede emplearse para recuperarse de errores imprevistos, particularmente de errores en el diseño de componentes software. Hasta este punto del capítulo, sólo se han considerado errores anticipables, aunque los manejadores de excepciones «atrapa todo» pueden emplearse para atrapar excepciones desconocidas. En esta sección, se describe la implementación de los bloques de recuperación mediante excepciones y manejadores de excepciones.

Como recordatorio, se muestra a continuación la estructura de un bloque de recuperación:

```
ensure <test de aceptación>
by
  <módulo primario>
else by
  <módulo alternativo>
else by
  <módulo alternativo>
  .
  .
  .
else by
  <módulo alternativo>
else error
```

El mecanismo de detección de error viene dado por el test de aceptación. Esta prueba es simplemente la negación de una prueba que generaría una excepción empleando una recuperación de error hacia delante. El único problema es la implementación de la salvaguardia del estado y su restauración correspondiente.

En el ejemplo siguiente, se muestra cómo un paquete Ada implementa una caché de recuperación. El procedimiento `Guarda` almacena el estado de las variables locales y globales del programa en una caché de recuperación; esto no incluye los valores del contador de programa, el puntero de pila y otros. Una llamada a `Restaura` restablecerá las variables del programa a los estados guardados.

```
package Cache_Recuperacion is
  procedure Guarda;
  procedure Restaura;
end Cache_Recuperacion;
```

Obviamente, hay algo de magia en este paquete que requerirá la ayuda del sistema de ejecución y, posiblemente, incluso el apoyo del hardware para la caché de recuperación. Asimismo, puede que no sea ésta la forma más eficiente de efectuar la restauración de un estado. Sería preferible proporcionar primitivas más básicas, y permitir que el programa emplee su conocimiento de la aplicación para optimizar la cantidad de información salvaguardada (Rogers y Wellings, 2000).

La finalidad del próximo ejemplo es mostrar que es posible usar bloques de recuperación con las técnicas dadas de la implementación de la caché de recuperación en un entorno de manejo de excepciones.

El esquema de bloque de recuperación puede además implementarse usando un lenguaje con excepciones, contando además con un poquito de ayuda por parte del sistema de soporte de ejecución. Por ejemplo, en Ada la estructura para un bloque de recuperación con triple redundancia podría ser:

```
procedure Bloque_Recuperacion is
  Fallo_Primario, Fallo_Secundario,
        Fallo_Terciario: exception;
  Fallo_Bloque_Recuperacion : exception;
  type Modulo is (Primario, Secundario, Terciario);
  function Test_Aceptacion return Boolean is
  begin
    -- código del test de aceptación
  end Test_Aceptacion;

  procedure Primario is
  begin
    -- código del algoritmo primario
    if not Test_Aceptacion then
      raise Fallo_Primario;
    end if;
  exception
```

```
      when Fallo_Primario =>
         -- recuperación hacia delante para devolver
         -- el entorno al estado requerido
         raise;
      when others =>
         -- error inesperado
         -- recuperación hacia delante para devolver
         -- el entorno al estado requerido
      raise Fallo_Primario;
   end Primario;

   procedure Secundario is
   begin
      -- código de un algoritmo secundario
      if not Test_Aceptacion then
         raise Fallo_Secundario;
      end if;
   exception
      when Fallo_Secundario =>
         -- recuperación hacia delante para devolver
         -- el entorno al estado requerido
         raise;
      when others =>
         -- error inesperado
         -- recuperación hacia delante para devolver
         -- el entorno al estado requerido
         raise Fallo_Secundario;
   end Secundario;

   procedure Terciario is
   begin
      -- código del algoritmo terciario
      if not Test_Aceptacion then
         raise Fallo_Terciario;
      end if;
   exception
   when Fallo_Terciario =>
      -- recuperación hacia delante para devolver
      -- el entorno al estado requerido
      raise;
   when others =>
```

```
      -- error inesperado
      -- recuperación hacia delante para devolver
      -- el entorno al estado requerido
      raise Fallo_Terciario;
    end Terciario;

begin
  Cache_Recuperacion.Guarda;
  for Intento in Modulo loop
    begin
      case Intento is
        when Primario => Primario; exit;
        when Secundario => Secundario; exit;
        when Terciario => Terciario;
      end case;
    exception
      when Fallo_Primario =>
        Cache_Recuperacion.Restaura;
      when Fallo_Secundario =>
        Cache_Recuperacion.Restaura;
      when Fallo_Terciario =>
        Cache_Recuperacion.Restaura;
        raise Fallo_Bloque_Recuperacion;
      when others =>
        Cache_Recuperacion.Restaura;
        raise Fallo_Bloque_Recuperacion;
    end;
  end loop;
end Bloque_Recuperacion;
```

Resumen

En este capítulo se han estudiado los diversos modelos de manejo de excepciones para procesos secuenciales. Aunque existen muchos modelos diferentes, todos abordan las siguientes cuestiones.

- La representación de excepciones: en los lenguajes, las excepciones pueden representarse explícita o implícitamente.
- El dominio de un manejador de excepciones: con cada manejador viene asociado un dominio que especifica la región de cómputo durante la cual se activará el manejador si aparece

una excepción. El dominio se encuentra normalmente asociado a un bloque, un subprograma o una sentencia.

- La propagación de excepciones: íntimamente relacionada con la idea de dominio de una excepción. Es posible que, cuando se lance una excepción, no exista ningún manejador de excepciones en el dominio que la encierra. En este caso, podrá propagarse la excepción hacia el siguiente nivel de dominio que lo envuelve, o podrá considerarse como un error de programación (que suele ser indicado en la compilación).

- El modelo de reanudación o de terminación: determina la acción a tomar tras el manejo de una excepción. Con el modelo de reanudación, el invocador de la excepción se reanuda en la sentencia posterior a la que provocó la excepción. Con el modelo de terminación, el bloque o procedimiento que contiene el manejador es terminado, y se pasa el control al bloque o procedimiento que lo llamó. El modelo híbrido permite que el manejador elija si reanudar o terminar.

- El paso de parámetros al manejador: puede estar permitido o no.

En la Tabla 6.1, se resumen las diversas posibilidades de manejo de excepción para los distintos lenguajes. No es algo unánimemente aceptado el hecho de que se proporcionen mecanismos de manejo de excepciones en un lenguaje. Los lenguajes C y occam2, por ejemplo, no los tienen. Para los escépticos, una excepción es un `goto` en el que el destino es impredecible y el origen es desconocido. Así pues, pueden ser consideradas la antítesis de la programación estructurada. No es ésta, sin embargo, la visión adoptada en este libro.

Tabla 6.1. Mecanismos de manejo de excepciones de varios lenguajes.

Lenguaje	Dominio	Propagación	Modelo	Parámetros
Ada	Bloque	Sí	Terminación	Limitado
Java	Bloque	Sí	Terminación	Sí
C++	Bloque	Sí	Terminación	Sí
CHILL	Sentencia	No	Terminación	No
CLU	Sentencia	No	Terminación	Sí
Mesa	Bloque	Sí	Híbrido	Sí

Lecturas complementarias

Cristian, F. (1982), «Exception Handling and Software Fault Tolerance», *IEEE Transactions on Computing*, c-31(6), 531-540.

Cui, Q., y Gannon, J. (1992), «Data-oriented Exception Handling», *IEEE Transactions on Software Engineering*, 18(5), 393-401.

Lee, P. A. (1983), «Exception Handling in C Programs», *Software – Practice and Experience*, 13(5), 389-406.

Papurt, D. M. (1998), «The Use of Exceptions», *JOOP*, 11(2), 13-17.

Ejercicios

6.1 Compare y contraste las aproximaciones de manejo de excepciones y los bloques de recuperación en relación con la tolerancia a fallos en el software.

6.2 Proporcione ejemplos de:

(1) Una excepción síncrona detectada por la aplicación.

(2) Una excepción asíncrona detectada por la aplicación.

(3) Una excepción síncrona detectada por el entorno de ejecución.

(4) Una excepción asíncrona detectada por el entorno de ejecución.

6.3 El paquete `Es_Caracter` cuya especificación se da a continuación, proporciona una función para leer caracteres del terminal. También provee un procedimiento para desechar todos los caracteres restantes de la línea actual. El paquete puede generar la excepción `Error_Es`.

```
package Es_Caracter is
   function Dame return Caracter;
      -- lee un carácter del terminal

   procedure Barre;
      -- desecha todos los caracteres de la línea actual
   Error_Es : exception;
end Es_Caracter;
```

Otro paquete `Busca` contiene la funcion `Lee`, que rastrea la línea de entrada actual buscando los símbolos de puntuación coma (,), punto (.), y punto y coma (;). La función devolverá el siguiente símbolo de puntuación encontrado, o generará la excepción `Puntuacion_Ilegal`, si se encuentra un caracter no alfanumérico. Si durante el proceso de rastreo de la línea de entrada se encuentra un `Error_Es`, se propagará la excepción hacia el invocador de `Lee`. A continuación se proporciona la especificación de `Busca`.

```
with Es_Caracter; use Es_Caracter;
   package Busca is
   type Puntuacion is (Coma, Punto, Puntoycoma);
   function Lee return Puntuacion;
```

```
        -- lee el siguiente , . o ; del terminal

        Puntuacion_Ilegal : exception;
    end Busca;
```

Bosqueje el cuerpo del paquete `Busca` usando el paquete `Es_Caracter` para leer los caracteres del terminal. Al recibir un símbolo de puntuación legal, un símbolo de puntuación ilegal, o una excepción `Error_Es`, se desechará el resto de la línea. Deberá presuponer que una línea de entrada siempre contará con un símbolo de puntuación legal o ilegal, y que `Error_Es` ocurrirá aleatoriamente. Usando el paquete `Busca`, bosqueje el código de un procedimiento `Dame_Puntuacion` que devuelva siempre el siguiente símbolo de puntuación a pesar de las excepciones que pudiera generar `Busca`. Ha de suponer un stream de entrada infinito.

6.4 En una aplicación de control, se calienta gas en una cámara cerrada. La cámara se encuentra rodeada por un refrigerante que reduce la temperatura del gas por conducción. También hay una válvula que, al ser abierta, libera el gas en la atmósfera. La operación del proceso está controlada por un paquete Ada cuya especificación se proporciona más adelante. Por razones de seguridad, el paquete reconoce diversas condiciones de error; éstas se notifican al usuario mediante la generación de excepciones. La excepción `Calentador_Atascado_Encendido` se genera desde el procedimiento `Apaga_Calentador` cuando es incapaz de apagar el calefactor. La excepción `Temperatura_Aun_Subiendo` se genera desde el procedimiento `Aumenta_Refrigerante` si es incapaz de bajar la temperatura del gas incrementando el caudal de refrigerante. Finalmente, la excepción `Valvula_Atascada` se genera desde el procedimiento `Abre_Valvula` si es incapaz de liberar el gas hacia la atmósfera.

```
package Control_Temperatura is
   Calentador_Atascado_Encendido, Temperatura_Aun_Subiendo
        Valvula_Atascada : exception;

   procedure Enciende_Calentador;
      -- enciende el calentador

   procedure Apaga_Calentador;
      -- apaga el calentador
      -- genera Calentador_Atascado_Encendido

   procedure Aumenta_Refrigerante;
      -- Hace que se incremente el caudal de refrigerante
      -- que rodea la cámara hasta que la temperatura
      -- alcance un nivel seguro
      -- genera Temperatura_Aun_Subiendo

   procedure Abre_Valvula;
```

```
        -- abre una valvula para soltar algo de gas y así
        -- evitar una explosión
        -- genera Valvula_Atascada

    procedure Panico;
        -- hace sonar una alarma y llama a los servicios
        -- de bomberos, hospital y policía
end Control_Temperatura;
```

Escriba un procedimiento Ada que al ser llamado intente apagar el calentador de la cámara de gas. Si el calentador está atascado y encendido aumentará el caudal de refrigerante que rodea la cámara. Si la temperatura continúa aumentando, entonces deberá abrir la válvula de escape para soltar el gas. Si esto falla, deberá hacer sonar una alarma e informar a los servicios de emergencia.

6.5 Escriba un paquete Ada de propósito general para implementar bloques de recuperación anidados. (Sugerencia: deberá encapsular los módulos primario y secundario, el test de aceptación y demás, y pasarlo como un parámetro al módulo genérico.)

6.6 ¿Hasta qué punto los mecanismos de manejo de excepciones de Ada podrían ser implementados aparte del lenguaje en un paquete estándar?

6.7 ¿Cómo defendería la opinión de que «una excepción Ada es un `goto` en el que el destino está indeterminado y el origen es desconocido»? ¿Se sostendría el mismo argumento para (a) el modelo de reanudación de manejo de excepciones y (b) el modelo de terminación de CHILL?

6.8 Compare los dominios de excepciones de los siguientes fragmentos de código aparentemente iguales (la variable `Inicial` es de tipo entero).

```
procedure Haz_Algo is
   subtype Int_Corto is Integer range -16..15;
   A : Int_Corto := Inicial;
begin
   ...
end Haz_Algo;
```

```
procedure Haz_Algo is
   subtype Int_Corto is Integer range -16..15;
   A : Int_Corto;
begin
   A := Inicial;
   ...
end Haz_Algo;
```

```
procedure Haz_Algo is
   subtype Int_Corto is Integer range -16..15;
   A : Int_Corto;
begin
   begin
      A := Inicial;
      ...
   end;
end Haz_Algo;
```

6.9 Muestre cómo pueden implementarse los macros C indicados en la Sección 6.3.3.

6.10 Considere el siguiente programa:

```
I : Integer := 1;
J : Integer := 0;
K : Integer := 3;

procedure Primario is
begin
   J := 20;
   I := K*J/2
end Primario;

procedure Secundario is
begin
   I :=J;
   K :=4;
end Secundario;

procedure Terciario is
begin
   J := 20;
   I := K*J/2
end Terciario;
```

Este código se ejecutaría en un entorno de un bloque de recuperacióny en un entorno de manejo de excepciones. El siguiente es el entorno de bloque de recuperación:

```
ensure I = 20
by
   Primario;
else by
   Secundario;
```

```
else by
   Terciario;
else
   I := 20;
end;
```

Lo siguiente es el entorno de manejo de excepciones:

```
Fallo : exception;
type Modulo is (P,S,T);

for Intento in Modulo loop
   begin
      case Intento is
         when P =>
            Primario;
            if I /=20then
               raise Fallo;
            end if;
            exit;
         when S =>
            Secundario;
            if I /=20then
               raise Fallo;
            end if;
            exit;
         when T =>
            Terciario;
            if I /=20then
               raise Fallo;
            end if;
            exit;
         end case;
   exception
      when Fallo =>
         if Intento = T then
            I := 20;
         end if;
   end;
end loop;
```

Teniendo en cuenta el *modelo de terminación* de manejo de excepciones, compare y contraste la ejecución de *ambos* fragmentos, el entorno de bloque de recuperación, y el entorno de manejo de excepciones.

6.11 Muestre cómo pueden implementarse los bloques de recuperación en C++ y Java. Sugerencia: véase Rubira-Calsavara y Stroud (1994).

6.12 Rehaga la solución al Ejercicio 6.3 empleando Java.

6.13 Rehaga la solución al Ejercicio 6.4 empleando Java.

6.14 Explique bajo qué circunstancias **`when others`** `(e: Exception Id)` en Ada no es equivalente a **`catch`**`(Exception e)` en Java.

6.15 ¿Cuáles son las ventajas y desventajas de disponer de las excepciones integradas en el modelo POO?

6.16 Se ha definido un subconjunto de Ada 95 para su empleo en sistemas embebidos de seguridad crítica. El lenguaje tiene una semántica formal y un conjunto de herramientas que permiten realizar técnicas de análisis estático. No dispone de mecanismos de manejo de excepciones. Los diseñadores argumentan que en los programas han demostrado ser correctos, y por tanto no pueden aparecer excepciones. ¿Qué argumentos pueden darse a favor de la introducción de mecanismos de manejo de excepciones en el lenguaje?

Capítulo 7

Programación concurrente

Virtualmente, todos los sistemas de tiempo real son inherentemente concurrentes. Los lenguajes destinados a ser usados en este dominio tienen mayor potencia expresiva si proporcionan al programador primitivas que se ajusten al paralelismo de las aplicaciones.

> Denominamos programación concurrente a la notación y técnicas de programación que expresan el paralelismo potencial y que resuelven los problemas resultantes de la sincronización y la comunicación. La implementación del paralelismo es un tema de los sistemas informáticos (hardware y software) esencialmente independiente de la programación concurrente. La importancia de la programación concurrente está en que proporciona un entorno abstracto donde estudiar el paralelismo sin tener que enfrascarse en los detalles de implementación. (Ben-Ari, 1982)

Este capítulo y los dos siguientes se centran en los asuntos relacionados con la programación concurrente en general. El tema del tiempo y de cómo los programas pueden representarlo y manipularlo se deja para el Capítulo 12.

7.1 La noción de proceso

Cualquier lenguaje, natural o informático, tiene un carácter dual: a la vez que es capaz de expresarse, también limita el entorno en el que se aplica dicha capacidad expresiva. Si un lenguaje no permite cierta noción o concepto particular, entonces aquéllos que utilizan el lenguaje no podrán emplear este concepto, y puede que, incluso, desconcozcan por completo su existencia.

Pascal, C, FORTRAN y COBOL comparten la propiedad común de ser lenguajes de programación secuenciales. Los programas escritos en estos lenguajes tienen un único hilo de control. Comienzan la ejecución en cierto estado y avanzan ejecutando una sentencia cada vez, hasta que el programa finaliza. La traza a través del programa puede diferir debido a variaciones en los datos de entrada, aunque para una ejecución concreta del programa existe una única traza. Esto no

es lo adecuado para la programación de sistemas de tiempo real. Según los trabajos precursores de Dijkstra (1968a), un programa concurrente puede verse como un conjunto de procesos secuenciales autónomos, que son ejecutados (lógicamente) en paralelo. Todos los lenguajes de programación concurrente incorporan, explícita o implícitamente, la noción de proceso; cada proceso tiene un hilo de control.[1]

La implementación real (esto es, la ejecución) de un conjunto de procesos tiene lugar normalmente de tres formas. Los procesos pueden:

(1) Multiplexar sus ejecuciones sobre un único procesador.

(2) Multiplexar su ejecuciones en un sistema multiprocesador con acceso a memoria compartida.

(3) Multiplexar sus ejecuciones en diversos procesadores que no comparten memoria (a estos sistemas se les denomina, normalmente, sistemas distribuidos).

También es posible encontrar híbridos de estos tres métodos.

Sólo en los casos (2) y (3) es posible una verdadera ejecución paralela de más de un proceso. El término **concurrente** indica paralelismo potencial. Los lenguajes de programación concurrente permiten al programador expresar actividades lógicamente paralelas sin tener en cuenta su implementación. En el Capítulo 14 se consideran los asuntos concernientes a la auténtica ejecución paralela.

En la Figura 7.1 se muestra, de una manera sencilla, la vida de un proceso. Una vez creado un proceso, pasa al estado de inicialización, y continúa con su ejecución hasta que finaliza. Obsérvese que puede que algunos procesos no terminen, y que otros fallen durante la inicialización y pasen a terminación sin haber sido ejecutados. Después de la finalización, los procesos pasan a no existencia, donde ya no puede accederse a ellos (al estar fuera del alcance). Sin duda, el estado más importante para un proceso es el de ejecución; sin embargo, dado que hay un límite de procesadores, no todos podrán estar en ejecución a la vez. Es decir, el término **ejecutable** expresa que el proceso podría ser ejecutado si existiera un procesador disponible.

Considerando así los procesos, queda claro que la ejecución de un programa concurrente no es tan directa como la ejecución de un programa secuencial. Los procesos deben ser creados y finalizados, así como distribuidos hacia/desde los procesadores disponibles. Estas actividades son efectuadas por el sistema de soporte de ejecución (RTSS; Run-Time Support System) o núcleo de ejecución. El RTSS posee muchas de las características del planificador de un sistema operativo, y está ubicado, lógicamente, entre el hardware y el software de aplicación. En realidad, puede tomar una de las siguientes formas:

(1) Una estructura software programada como parte de la aplicación (esto es, como un componente del programa concurrente). Ésta es la aproximación adoptada por el lenguaje Modula-2.

[1] Recientes trabajos sobre sistemas operativos han aportado una noción explícita de hilo y de procesos multihilo. Posteriormente se volverá a discutir este asunto en esta sección.

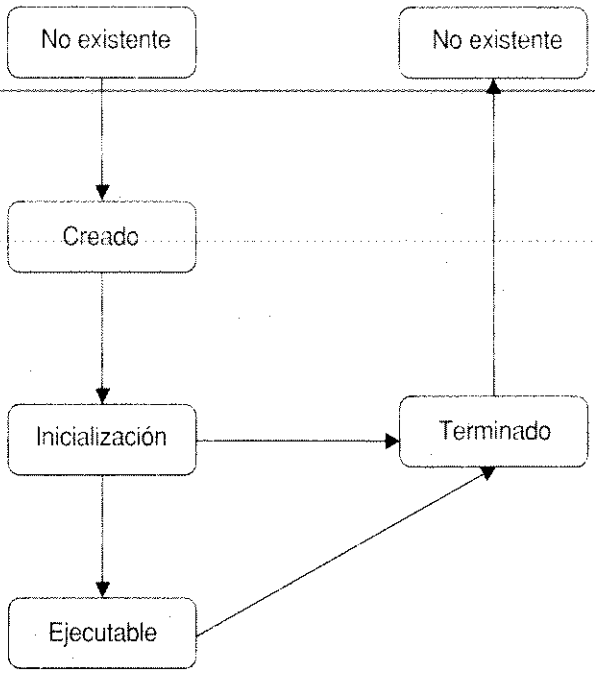


Figura 7.1. Diagrama de estados sencillo para un proceso.

(2) Un sistema software estándar generado junto al código objeto del programa por el compilador. Ésta es la estructura habitual en los programas de Ada y Java.

(3) Una estructura hardware microcodificada en el procesador, por motivos de eficiencia. Un programa occam2 que se ejecuta en un transputer tiene este sistema de ejecución.

El algoritmo que utiliza el RTSS para planificar (esto es, para decidir qué proceso se ejecuta a continuación en el caso de haber más de uno ejecutable) afectará al comportamiento temporal del programa, aunque, para programas bien construidos, el comportamiento lógico no dependerá del RTSS. Desde el punto de vista del programa, se asume que el RTSS planifica los procesos de forma no determinista. En el caso de los sistemas de tiempo real, las características de la planificación son relevantes, y serán tenidas en cuenta en el Capítulo 13.

Aunque este libro se centra en los lenguajes de tiempo real concurrentes, es claro que una alternativa sería utilizar un lenguaje secuencial y un sistema operativo de tiempo real. Todos los sistemas operativos proporcionan mecanismos para crear procesos concurrentes. Normalmente, cada proceso se ejecuta en su propia máquina virtual, para evitar interferencias con otros procesos no relacionados. Cada proceso consta, realmente, de un único programa. Sin embargo, en los últimos años se tiende a permitir la creación de procesos dentro de los programas. Los sistemas operativos modernos permiten crear procesos dentro del mismo programa accediendo de modo compartido, y sin restricciones, a la memoria común (estos procesos suelen llamarse **hilos** o hebras). Por tanto, en los sistemas operativos que se ajustan a POSIX, es preciso distinguir entre la concurrencia de programas (procesos), y la concurrencia dentro de un programa (hilos). También es frecuente distinguir entre aquellos hilos visibles desde el sistema operativo y aquéllos que provienen únicamente del soporte de ciertas rutinas de biblioteca. Por ejemplo, Windows 2000 soporta hilos y **fibras**, siendo estas últimas invisibles para el núcleo.

Ha habido un amplio debate entre programadores, diseñadores de lenguajes y diseñadores de sistemas operativos, sobre si lo apropiado es que sea el lenguaje quien dé soporte para la concurrencia, o si éste debiera ser proporcionado únicamente por el sistema operativo. Los argumentos a favor de incluir la concurrencia en los lenguajes de programación son los siguientes:

(1) Lleva a programas más legibles y fáciles de mantener.

(2) Existen muchos tipos distintos de sistemas operativos; al definir la concurrencia en el lenguaje se consiguen programas más portables.

(3) Puede que el computador embebido ni siquiera disponga de un sistema operativo residente.

Claramente, éstos fueron los argumentos que más pesaron sobre los diseñadores de Ada y Java. Los argumentos en contra de la concurrencia en el lenguaje son los siguientes:

(1) Cada lenguaje tiene un modelo de concurrencia distinto; resulta más sencillo componer programas de distintos lenguajes si todos utilizan el mismo modelo de concurrencia del sistema operativo.

(2) Puede no ser fácil implementar eficientemente cierto modelo de concurrencia de un lenguaje sobre algún modelo de sistema operativo.

(3) Comienzan a aparecer estándares de sistema operativo, y por tanto los programas se vuelven más portables.

La necesidad de soportar diversos lenguajes fue una de las principales razones por las que la industria aeronáutica civil, al desarrollar su programa de Aviónica Modular Integrada, optó por una interfaz de programación de aplicaciones-núcleo estándar (llamada APEX) que soporta concurrencia, en vez de adoptar el modelo de concurrencia de Ada (ARINC AEE Committee, 1999). El debate, sin duda, continuará durante algún tiempo.

7.1.1 Construcciones de programación concurrente

A pesar de que las construcciones de programación concurrente varían de un lenguaje (y sistema operativo) a otro, deben proporcionar tres servicios fundamentales, que se citan a continuación:

(1) La expresión de ejecución concurrente mediante la noción de proceso.

(2) La sincronización de procesos.

(3) La comunicación entre procesos.

Al considerar la interacción entre procesos, es útil distinguir entre tres tipos de comportamiento:

- Independiente
- Cooperativo
- Competitivo

Los procesos independientes no se comunican o sincronizan entre sí. Por contra, los procesos cooperativos se comunican con regularidad y sincronizan sus actividades para realizar alguna operación común. Por ejemplo, un componente de un sistema embebido puede constar de diversos procesos involucrados en el mantenimiento dentro de ciertos límites de la temperatura y humedad de un gas en un contenedor. Esto puede requerir frecuentes interacciones.

Un sistema informático consta de un número finito de recursos que pueden ser compartidos entre los procesos; por ejemplo, periféricos, memoria y potencia de procesador. Para que los procesos obtengan su proporción justa de recursos, deben competir entre sí. La asignación de recursos necesita inevitablemente comunicación y sincronización entre los procesos del sistema. Pero, aunque estos procesos se comuniquen y sincronicen para obtener recursos, son esencialmente independientes.

Los tres próximos capítulos se centran en el examen de las facilidades que permiten la creación e interacción de procesos.

Ejecución concurrente

A pesar de que la noción de proceso es común a todos los lenguajes de programación concurrente, hay variaciones considerables en los modelos de concurrencia que se adoptan. Estas variaciones conciernen a los siguientes elementos:

- Estructura
- Nivel
- Granularidad
- Inicialización
- Finalización
- Representación

La *estructura* de un proceso puede ser clasificada de la siguiente forma:

- Estática: el número de procesos es fijo y conocido en tiempo de compilación.
- Dinámica: los procesos son creados en cualquier momento. El número de procesos existentes sólo se determina en tiempo de ejecución.

Tabla 7.1. Características de estructura y nivel de varios lenguajes de programación concurrente.

Lenguaje	Estructura	Nivel
Pascal concurrente	estática	plano
occam2	estática	anidado
Modula-1	dinámica	plano
Modula-2	dinámica	plano
Ada	dinámica	anidado
Java	dinámica	anidado
Mesa	dinámica	anidado
C/POSIX	dinámica	plano

Otra distinción entre lenguajes proviene del *nivel* de paralelismo soportado. De nuevo, se pueden identificar dos casos distintos:

(1) Anidado: los procesos se definen en cualquier nivel del texto del programa; en particular, se permite definir procesos dentro de otros procesos.

(2) Plano: los procesos se definen únicamente en el nivel más externo del texto del programa.

La Tabla 7.1 indica las características de estructura y nivel para varios lenguajes de programción concurrentes. En esta tabla se considera que el lenguaje C incorpora las primitivas «fork» y «wait» de POSIX.

Dentro de los lenguajes que permiten contrucciones anidadas, existe también una interesante distinción entre lo que se puede llamar paralelismo de **grano grueso** y de **grano fino**. Un programa concurrente de grano grueso contiene relativamente pocos procesos, cada uno con una historia de vida significativa. Por su parte, los programas con paralelismo de grano fino tienen un número mayor de procesos sencillos, algunos de los cuales existen para una única acción. La mayoría de los lenguajes de programación concurrentes, representados por Ada, muestran paralelismo de grano grueso. Occam2 es un buen ejemplo de lenguaje concurrente con paralelismo de grano fino.

Cuando se crea un proceso, puede ser necesario proporcionar información relacionada con su ejecución (de la misma forma que un subprograma requiere cierta información cuando es invocado). Hay dos formas de realizar esta inicialización: la primera es pasar al proceso la información en forma de parámetros; la segunda es la comunicación explícita con el proceso, después de que haya comenzado su ejecución.

La **finalización** de procesos se puede realizar de distintas formas. A continuación se resumen las circunstancias en las que se permite que un proceso finalice:

(1) Finalización de la ejecución del cuerpo del proceso.

(2) Suicidio por ejecución de una sentencia de «autofinalización».

(3) Aborto por medio de una acción explícita de otro proceso.

(4) Ocurrencia de una condición de error sin tratar.

(5) Nunca: procesos que se ejecutan en bucles que no terminan.

(6) Cuando ya no son necesarios.

Con la anidación en niveles, es posible crear jerarquías de procesos y establecer relaciones entre ellos. Para cualquier proceso, es útil distinguir entre el proceso (o bloque) que es responsable de su creación, y el proceso (o bloque) que es afectado por su finalización. La primera relación es conocida como **padre/hijo**, y posee la característica de que el padre puede ser detenido mientras el hijo se crea e inicializa. La segunda se denominada **guardián/dependiente**, y en ella un proceso puede depender del propio proceso guardián o de un bloque interno de éste. El guardián no puede terminar un bloque hasta que todos los procesos dependientes hayan terminado (esto es, un proceso no puede existir fuera de su alcance). Consecuentemente, un guardián no puede terminar hasta que lo hayan hecho todos sus procesos dependientes. La consecuencia de esta regla es que un programa no podrá terminar hasta que todos los procesos creados en él hayan terminado también.

En algunas situaciones, el padre de un proceso será también su guardián. Éste será el caso cuando se utilicen lenguajes que sólo permiten estructuras estáticas de procesos (por ejemplo occam2). Con estructuras dinámicas de procesos (que también son anidadas), el padre y el guardián pueden no ser el mismo. Esto se ilustrará posteriomente cuando se vea Ada.

La Figura 7.2 incluye los nuevos estados que aparecen a raíz de la anterior discusión.

Una de las formas en que un proceso puede finalizar –punto (3) de la lista anterior– es por la aplicación de una sentencia que lo aborta (abort). La existencia de abort en los lenguajes de programación concurrentes es una cuestión de cierta controversia, y se tratará en el Capítulo 11 en el contexto del control de recursos. Para una jerarquía de procesos suele ser necesario que la interrupción de un guardián implique la interrupción de todos los dependientes (y de los dependientes de éstos, y así recursivamente).

La última circunstancia de finalización de la lista anterior se verá con más detalle al describir los métodos de comunicación de procesos. En esencia, permite que un proceso servidor termine si el resto de procesos que podrían comunicar con él ya han terminado.

7.2.1 Procesos y objetos

El paradigma de programación orientada al objeto anima a que los constructores de sistemas (y de programas) consideren el artefacto en construcción como un conjunto de objetos cooperantes o, para utilizar un término más neutral, de entidades. Bajo este paradigma, es provechoso considerar dos tipos de objetos: activos y reactivos. Los objetos **activos** acometen acciones espontáneamente (con un procesador de por medio): hacen posible que la computación prosiga. Los

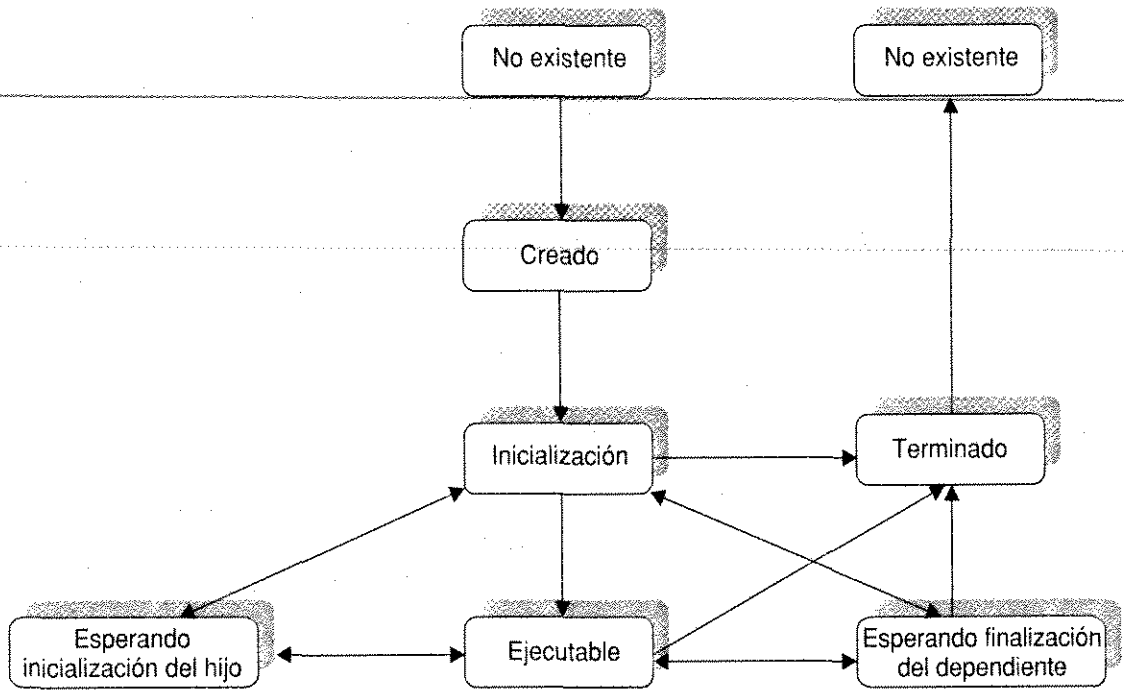


Figura 7.2. *Diagrama de estados de un proceso.*

objetos **reactivos**, por contra, sólo entran en acción cuando son «invocados» por un objeto activo. Otros paradigmas de programación, como el de flujos de datos o el de redes de tiempo real, identifican agentes activos y datos pasivos.

Sólo las entidades activas dan lugar a acciones espontáneas. Los recursos son reactivos, aunque pueden controlar el acceso a su estado interno (y a cualquier recurso real que controlen). Algunos recursos sólo pueden ser usados por un único agente en cada momento; en otros casos, las operaciones realizables en un determinado momento dependen de los estados actuales de los recursos. Un ejemplo común del último caso es el del búfer de datos, cuyos elementos no pueden ser extraídos si está vacío. El término **pasivo** se utilizará para designar entidades reactivas que permiten un acceso completo.

La implementación de entidades de recurso requiere algún tipo de agente de control. Si el agente de control es pasivo (como un semáforo), entonces se dice que el recurso está **protegido** (o **sincronizado**). Por otro lado, si se precisa de un agente activo para programar el nivel adecuado de control, podemos decir, en cierto sentido, que el recurso es activo. El término **servidor** se utilizará para identificar a este último tipo de entidades, y el término **recurso protegido** para el tipo pasivo. Éstos, junto con **activo** y **pasivo**, son las cuatro entidades abstractas de programas utilizadas en este libro.

En un lenguaje de programación concurrente, las entidades activas se representan mediante procesos. Las entidades pasivas pueden representarse directamente como variables de datos, o pueden ser encapsuladas por alguna construcción módulo/paquete/clase que proporcione una interfaz procedural. Los recursos protegidos también pueden estar encapsulados en una construcción tipo módulo, y necesitar disponer de un servicio de sincronización de bajo nivel. Los servidores requieren un proceso, ya que es necesario programar el agente de control.

Una cuestión clave para los diseñadores de lenguajes es soportar o no primitivas para los recursos protegidos y para los servidores. Los recursos suelen implementarse eficientemente (al menos en sistemas uniprocesadores), ya que normalmente emplean un agente de control de bajo nivel (por ejemplo un semáforo). Pero, para algunas clases de programas, esto puede ser poco flexible, y llevar a estructuras de programa deficientes (esto se discute con más detalle en el Capítulo 8). Los servidores son esencialmente flexibles, ya que el agente de control se programa mediante un proceso. El inconveniente de esta aproximación es que puede desembocar en un aumento de procesos y, en consecuencia, en un elevado número de cambios de contexto durante la ejecución. Esto es particularmente problemático si el lenguaje no permite recursos protegidos y hay que utilizar un servidor para cada entidad. Como se ilustrará en este capítulo, y en los Capítulos 8 y 9, Ada, Java y POSIX soportan todo el rango completo de entidades. Occam2, por contra, sólo soporta servidores.

7.3 Representación de procesos

Hay tres mecanismos básicos para representar la ejecución concurrente: fork y join, cobegin y la declaración explícita de procesos. A veces se incluyen también las corrutinas como mecanismo para expresar la ejecución concurrente.

7.3.1 Corrutinas

Las corrutinas son como subrrutinas, salvo que permiten el paso explícito de control entre ellas de una forma simétrica en vez de estrictamente jerárquica. El control se transfiere de una corrutina a otra mediante una sentencia **reanuda** (resume) que incluye el nombre de la corrutina con la que se continúa. Cuando una corrutina realiza una «reanudación», deja de ejecutarse, pero guarda información del estado local, de forma que si, posteriormente, otra corrutina la hace reanudar, podrá retomar su ejecución. La Figura 7.3 ilustra la ejecución de tres corrutinas; las flechas numeradas representan el flujo de control que comienza en la corrutina *A* y finaliza en la corrutina *B* (habiendo visitado dos veces la corrutina *C*).

Cada corrutina puede verse como parte de un proceso separado; sin embargo, no es necesario un sistema de soporte de ejecución, ya que las propias corrutinas se colocan en orden de ejecución. Claramente, las corrutinas no son adecuadas para un auténtico procesamiento paralelo, ya que su semántica sólo permite la ejecución de una rutina cada vez. Modula-2 es un ejemplo de lenguaje que soporta corrutinas.

7.3.2 Fork y join

Esta sencilla aproximación no proporciona una entidad visible de proceso, sino que aporta dos instrucciones. La instrucción fork (bifurca) indica que cierta rutina deberá comenzar a ejecutarse concurrentemente con quien ha invocado fork. La instrucción join (reune) permite al que invoca detenerse, y por ende sincronizarse, hasta la terminación de la rutina invocada. Por ejemplo:

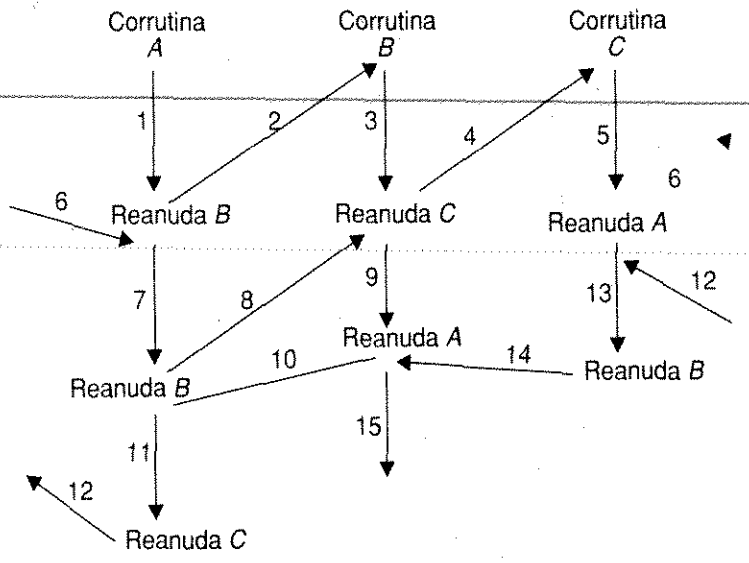


Figura 7.3. Flujo de control de corrutinas.

```
function F return ... ;
.
.
.
end F;

procedure P;
   ...
   C:= fork F;
      .
      .
      .
   J:= join C;
   ...
end P;
```

Entre la ejecución de fork y join, el procedimiento P y la función F se ejecutarán concurrentemente. Al llegar a join, el procedimiento esperará hasta que finalice la función (si es que no lo ha hecho ya). La Figura 7.4 ilustra la ejecución de fork y join.

El lenguaje Mesa proporciona la notación fork y join. También POSIX proporciona una versión de fork y join; aunque aquí `fork` sirve para crear una copia del invocador, y el join efectivo se obtiene mediante la llamada del sistema `wait`.

Fork y join permiten crear procesos dinámicamente, y proporcionan un mecanismo para pasar información al proceso hijo a través de parámetros. Normalmente, al terminar el hijo, devuelve un solo valor. Aunque flexibles, fork y join no proporcionan una aproximación estructurada a la creación de procesos, y su utilización es propensa a errores. Por ejemplo, un proceso guardián debe «reunir» explícitamente todos los procesos dependientes, en vez de esperar simplemente a que terminen.

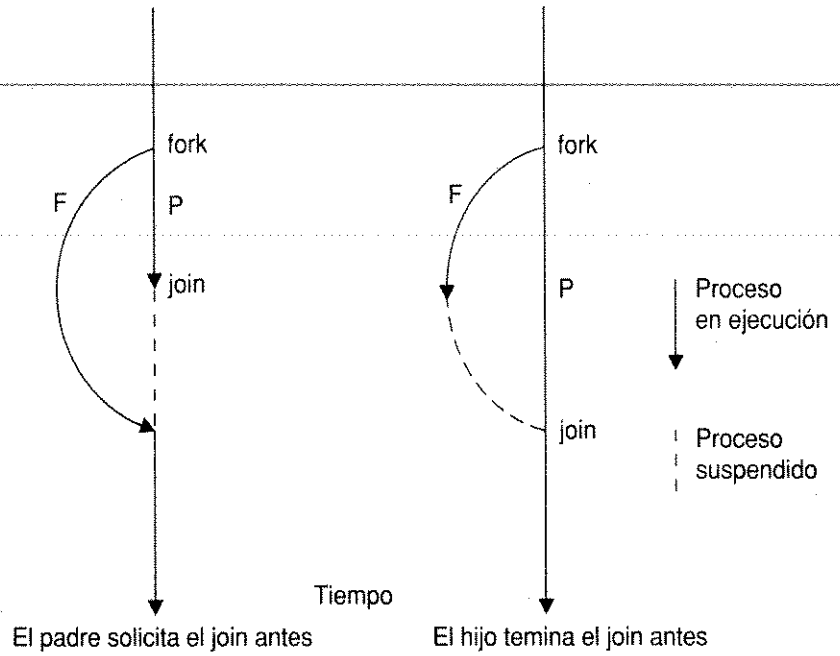


Figura 7.4. Fork y join.

7.3.3 Cobegin

Cobegin (o parbegin, o par) es una forma estructurada de denotar la ejecución concurrente de un conjunto de instrucciones:

```
cobegin
  S1;
  S2;
  S3;
  .
  .
  .
  Sn
coend
```

Este código permite la ejecución concurrente de las instrucciones `S1`, `S2`, etc. La instrucción cobegin («cocomienzo») termina cuando han terminado todas las instrucciones concurrentes. Cada instrucción `Si` puede ser cualquiera de las construcciones permitidas en el lenguaje, incluyendo las asignaciones sencillas o las llamadas a procedimientos. De invocar algún procedimiento, podrán pasarse datos a los procesos invocados mediante los parámetros de la llamada. La instrucción cobegin podría incluir, a su vez, una secuencia de instrucciones donde apareciera cobegin, y así construir una jerarquía de procesos. La Figura 7.5 ilustra la ejecución de la instrucción cobegin.

Cobegin puede encontrarse en occam2.

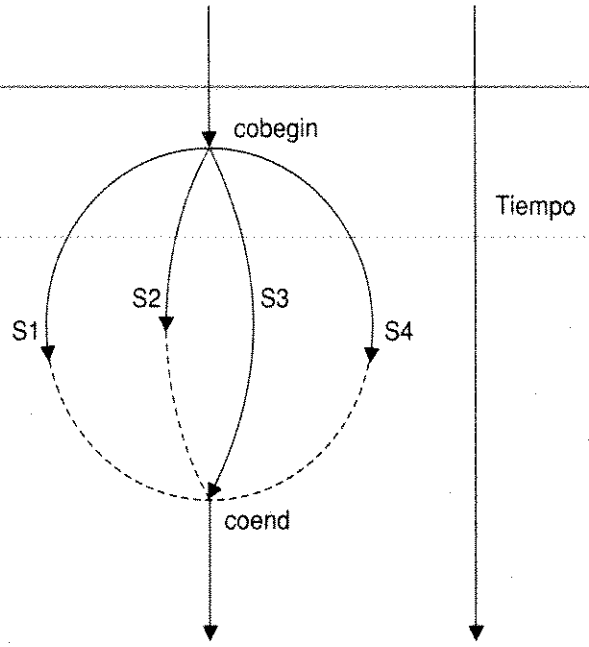


Figura 7.5. Cobegin.

7.3.4 Declaración explícita de procesos

A pesar de que cobegin y fork permiten expresar la ejecución concurrente de rutinas secuenciales, la estructura de un programa concurrente es mucho más nítida si las propias rutinas son quienes establecen su ejecución concurrente. La declaración explícita de procesos proporciona esta posibilidad. El siguiente ejemplo (en Modula-1) presenta una estructura sencilla para un controlador de brazo de robot. Para controlar cada dimensión del movimiento se emplea un proceso distinto. Cada proceso es iterativo, y lee una nueva indicación para su dimensión, llamando después al procedimiento de bajo nivel `mover_brazo`, que efectúa el movimiento del brazo.

```
MODULE main;
  TYPE dimension = (xplano, yplano, zplano);

  PROCESS control(dim : dimension);
    VAR  posicion : integer;     (* posición absoluta *)
         indicacion : integer;   (* movimiento relativo *)
    BEGIN
    posicion := 0;               (* posición restante *)
    LOOP
      mover_brazo(dim, posicion);
      nueva_indicacion(dim, indicacion);
      posicion := posicion + indicacion
    END
  END control;
BEGIN
```

```
  control(xplano);
  control(yplano);
  control(zplano)
END main.
```

Aquí, se declara el proceso `control` con un parámetro enviado en tiempo de creación. Según lo anterior, se crean tres instancias de este proceso, a cada una de las cuales se le pasa un parámetro diferente.

Otros lenguajes que soportan la declaración explícita de procesos, como por ejemplo Ada, permiten también la creación implícita de tareas. Todos los procesos declarados dentro de un bloque comienzan a ejecutarse concurrentemente al final de la parte declarativa de dicho bloque.

En la anterior discusión se han resumido los modelos básicos de ejecución concurrente, y se ha hecho referencia a lenguajes que soportan ciertas características particulares. Con el fin de proporcionar ejemplos concretos sobre lenguajes de programación reales, se verá la ejecución concurrente en occam2, Ada y Java. Complementariamente, se comentará la ejecución concurrente en POSIX.

7.3.5 Ejecución concurrente en occam2

Occam2 utiliza la estructura cobegin `PAR`. Considérese, por ejemplo, la ejecución concurrente de dos asignaciones simples (`A:=1` y `B:=1`):

```
PAR
  A :=1
  B :=1
```

Compárese esta estructura `PAR`, que puede ser anidada, con la forma secuencial:

```
SEQ
  A :=1
  B :=1
```

Obsérvese que, dado un conjunto de acciones, puede definirse explícitamente su ejecución en secuencia o en paralelo; no existe valor por defecto. Ciertamente, puede decirse que `PAR` es la forma más general y que sería la estructura natural, a no ser que el código en cuestión requiera `SEQ` (secuencia).

Si el proceso designado por un `PAR` es una instancia de un `PROC` (procedimiento) parametrizado, podrán pasarse los datos al proceso en la creación. El siguiente código, por ejemplo, crea tres procesos a partir de un único `PROC`, pasando un array a cada uno de ellos:

```
PAR
  ProcesoEjemplo(A[1])
  ProcesoEjemplo(A[2])
```

```
  ProcesoEjemplo(A[3])
```

Es posible crear un conjunto mayor de procesos, partiendo del mismo PROC, mediante un «replicador» que aumente la potencia del PAR:

```
PAR i=1 FOR N
  ProcesoEjemplo(A[i])
```

En el Capítulo 3 se utilizó un replicador con un SEQ para presentar un bucle «for» estándar. La única distinción entre un SEQ replicado y un PAR replicado es que el número de réplicas (N en el ejemplo anterior) debe ser constante en el PAR; esto es, es conocido en tiempo de compilación. Se sigue, entonces, que occam2 tiene una estructura estática de procesos.

El siguiente fragmento de programa en occam2 es el ejemplo del sistema de brazo de robot mostrado anteriormente en Modula-1. Observe que occam2 no contempla tipos enumerados, y por lo tanto las dimensiones se representan mediante los enteros 1, 2 y 3.

```
PROC control(VAL INT dim)
  INT  posicion,                -- posición absoluta
       indicacion:              -- movimiento relativo
  SEQ
    posicion := 0               -- posición restante
    WHILE TRUE
      SEQ
        MoverBrazo(dim,posicion)
        NuevaIndicacion(dim,indicacion)
        posicion := posicion + indicacion
:
PAR
  control(1)
  control(2)
  control(3)
```

La terminación de procesos es bastante directa en occam2. No existe la facilidad para abortar ni existe ninguna excepción. Un proceso debe terminar normalmente o no terminar (como en el ejemplo anterior).

En general, occam2 presenta una vista de concurrencia de grano fino. La mayoría de los lenguajes de programación concurrentes insertan la noción de proceso en un marco esencialmente secuencial. No es éste el caso de occam2; el concepto de proceso es básico en este lenguaje. Todas las actividades son consideradas procesos, incluyendo las operaciones de asignación y las llamadas a procedimientos. Ni siquiera la noción de instrucción existe en occam2: un programa es un único proceso construido como una jerarquía de procesos. En el nivel más bajo, se considera que todas las acciones primitivas son procesos, y los constructores (IF, WHILE, CASE y el resto) son propiamente procesos constructores.

7.3.6 Ejecución concurrente en Ada

En Ada, la unidad convencional de paralelismo, el proceso secuencial, se denomina **tarea**. Pueden declararse tareas en cualquier nivel de programa; se crean implícitamente en la entrada al alcance de su declaración. El siguiente ejemplo muestra un procedimiento que contiene dos tareas (A y B).

```
procedure Ejemplo1 is
  task A;
  task B;

  task body A is
  -- declaraciones locales para la tarea A
  begin
  -- secuencia de instrucciones para la tarea A
  end A;

  task body B is
  -- declaraciones locales para la tarea B
  begin
  -- secuencia de instrucciones para la tarea B
  end B;

begin
  -- las tareas A y B comienzan su ejecución antes
  -- de la primera instrucción de la secuencia de
  -- instrucciones pertenecientes al procedimiento
  .
  .
  .
end Ejemplo1; -- el procedimiento no termina
              -- hasta que las tareas A y B hayan terminado
```

Las tareas, como los paquetes, están formadas por una especificación y un cuerpo, y se les pueden pasar datos de inicialización en la creación.

Las tareas anteriores, A y B (que son creadas al invocar el procedimiento), se dice que son de tipo anónimo, ya que no tienen declarado un tipo (por analogía con los tipos anónimos de arrays). Se podrían haber dado fácilmente tipos para A y B:

```
task type Tipo_A;
task type Tipo_B;
A : Tipo_A;
B : Tipo_B;
```

```
task body Tipo_A is
  -- igual que antes para el cuerpo de la tarea A
task body Tipo_B is
  -- igual que antes para el cuerpo de la tarea B
```

Con los tipos de tareas, se puede declarar fácilmente un conjunto de instancias del mismo proceso utilizando un array:

```
task type T;
A,B : T;
type Largo is array (1..100) of T;
type Mezcla is
record
  Indice : Integer;
  Accion : T;
end record;
L : Largo; M : Mezcla;
task body T is ...
```

Un ejemplo más concreto de utilización de tareas en un programa Ada es el sistema del brazo de robot presentado anteriormente en Modula-1 y en occam2.

```
procedure Main is
  type Dimension is (Xplano, Yplano, Zplano);
  task type Control(Dim : Dimension);

  C1 : Control(Xplano);
  C2 : Control(Yplano);
  C3 : Control(Zplano);

  task body Control is
    Posicion : Integer;        -- posición absoluta
    Indicacion : Integer;      -- movimiento relativo
  begin
    Posicion := 0;             -- posición restante
    loop
      Mover_Brazo(Dim, Posicion);
      Nueva_Indicacion(Dim, Indicacion);
      Posicion := Posicion + Indicacion;
    end loop;
  end Control;
begin
  null;
end Main;
```

Merece la pena destacar que Ada sólo permite el paso de tipos discretos y tipos acceso como parámetros de inicialización de una tarea.

Se puede crear dinámicamente cierto número de tareas dando valores dinámicamente a los límites de un array (de tareas). La creación dinámica de tareas también se puede obtener utilizando explícitamente el operador «new» (nuevo) sobre un tipo acceso (de un tipo tarea):

```
procedure Ejemplo2 is
  task type T;
  type A is access T;
  P :A;
  Q : A:= new T;
begin
  ...
  P:= new T;
  Q:= new T;
  ...
end Ejemplo2;
```

Q es de tipo A, y se le da el «valor» de una nueva realización de T. Esto crea una tarea que comienza inmediatamente su inicialización y ejecución; la tarea se denomina `Q.all` (**all** es la convención que se utiliza en Ada para indicar la propia tarea, y no el apuntador de acceso). Durante la ejecución del procedimiento, a P se le asigna una tarea (`P.all`) seguida de una asignación adicional de Q. Existen entonces tres tareas activas dentro del procedimiento: `P.all`, `Q.all`, y la tarea que fue creada en primer lugar. Esta primera tarea es ahora anónima, ya que Q ha dejado de apuntar a ella. Además de estas tres tareas, existe una tarea o programa principal, que ejecuta el propio código del procedimiento; por tanto, en total hay cuatro hilos de control distintos.

Las tareas que se crean por una operación de realización («new») tienen la importante propiedad de que en ellas el bloque que actúa a modo de guardián (o **maestro**, como se conoce en Ada) no es el bloque en el que es creada, sino el que contiene la declaración del tipo acceso. Para ilustrar este punto, considérese lo siguiente:

```
declare
  task type T;
  type A is access T;
begin
  .
  .
  .
  declare          -- bloque interno
    X :T;
    Y : A:= new T;
```

```
   begin
      -- secuencia de instrucciones
   end;   -- debe esperar a que termine X, pero no Y.all
   .
   .      -- Y.all podría todavía estar activa, aunque el nombre Y
          -- está fuera del alcance
   .
end;      -- debe esperar a que termine Y.all
```

A pesar de que tanto X como `Y.all` son creadas en el bloque interno, sólo X lo tiene como maestro. La tarea `Y.all` se considera dependiente del bloque externo, y por tanto no afecta a la finalización del bloque interno.

Si una tarea falla mientras está siendo inicializada (en Ada este proceso se denomina **activación**), entonces el padre de la tarea recibe la excepción `Tasking_Error` «generada». Esto ocurre, por ejemplo, si se da a una variable un valor de inicialización inadecuado. Una vez que la tarea comienza su verdadera ejecución, puede «atrapar» cualquier excepción que se genere.

Identificación de tareas

Uno de los usos principales de las variables de acceso es el de proporcionar otra forma de dar nombre a las tareas. En Ada, todos los tipos tarea se consideran *privados limitados*. Por tanto, no es posible pasar una tarea por asignación a otra estructura de datos o unidad de programa. Por ejemplo, si `Brazo_Robot` y `Nuevo_Brazo` son dos variables del mismo tipo de acceso (el tipo acceso obtenido del tipo tarea), será ilegal lo siguiente:

```
Brazo_Robot.all := Nuevo_Brazo.all;  -- no legal en Ada
```

Sin embargo,

```
Brazo_Robot := Nuevo_Brazo;
```

es perfectamente legal, y significa que a `Brazo_Robot` se le asigna la misma tarea que a `Nuevo_Brazo`.

Aquí hay que tener cuidado, ya que la duplicación de nombres puede causar confusión y hacer que los programas sean difíciles de entender. Es más, puede eliminarse todo apuntador a una cierta tarea, con lo que ésta se convierte en **anónima**. Por ejemplo, si `Brazo_Robot` apuntaba a una tarea cuando fue sobrescrito con `Nuevo_Brazo`, entonces la tarea previa se habrá convertido en anónima, caso de no tener otros apuntadores.

En algunas circunstancias es útil disponer de identificadores únicos para las tareas (mejor que nombres). Veamos el porqué: normalmente a una tarea servidor no le importa el tipo de las tareas cliente, y cuando se comente la comunicación y la sincronización en el siguiente capítulo, se verá que el servidor no tiene conocimiento directo de quiénes son los clientes; sin embargo, hay ocasiones en las que el servidor necesita conocer que la tarea cliente con la que se está comunicando es la misma con la que se comunicó previamente. Aunque el núcleo de Ada no proporciona esta

utilidad, el Anexo de Programación de Sistemas incluye un mecanismo por el que una tarea puede obtener su identificador único, de modo que podrá pasarlo a otras tareas:

```
package Ada.Task_Identification is

  type Task_Id is private;
  Null_Task_Id : constant Task_Id;

  function "=" (Left, Right : Task_Id) return Boolean;

  function Current_Task return Task_Id;
     -- devuelve id único de la tarea que invoca

  -- otras funciones no relevantes para esta discusión
private
  ...
end Ada.Task_Identification;
```

Además de este paquete, el Anexo aporta dos atributos:

(1) Para cualquier prefijo `T` de un tipo tarea, `T'Identity` devuelve un valor de tipo `Task_Id`, que representa un identificador único de la tarea designada por `T`.

(2) Para cualquier prefijo `E` que represente una declaración de entrada, `E'Caller` devuelve un valor de tipo `Task_Id`, que representa el identificador único de la tarea a cuya llamada de entrada se está dando servicio. El atributo sólo está permitido dentro de un cuerpo de entrada o de una instrucción accept (véanse los Capítulos 8 y 9).

Hay que andar con ojo al utilizar identificadores de tareas, ya que no hay garantía alguna de que en algún momento posterior cierta tarea siga activa o esté en el alcance.

Finalización de tareas

Una vez consideradas la creación y la representación de tareas, falta la finalización. Ada proporciona una serie de opciones. Una tarea finalizará si:

(1) Se completa su ejecución, bien normalmente, bien como resultado de una excepción sin manejar.

(2) Ejecuta una alternativa de terminación de una instrucción select (esto se explica en la Sección 9.4.2), lo que implica que ya no se necesita.

(3) Es abortada.

Si una excepción sin controlar ha causado la muerte de una tarea, entonces el efecto del error está confinado en la tarea, y otra tarea podrá preguntar (utilizando un atributo) si la tarea ha terminado:

```
if T'Terminated then -- para alguna tarea T
  -- acción de recuperación del error
end if;
```

Sin embargo, la tarea que hace la pregunta no puede diferenciar entre una finalización normal o anómala.

Cualquier tarea puede abortar cualquier otra cuyo nombre esté en su alcance. Cuando se aborta una tarea, se abortan también sus dependientes. La función «aborta» permite la eliminación de tareas irregulares. Sin embargo, si una tarea *engañosa* es anónima, no podrá ser identificada ni tampoco abortada (a no ser que se aborte su maestra). Es deseable, por tanto, que sólo las tareas finalizadas se hagan anónimas.

7.3.7 Ejecución concurrente en Java

Java dispone de una clase predefinida, `java.lang.Thread`, que proporciona el mecanismo de creación de hilos (procesos). Sin embargo, para evitar que todos los hilos sean clases hijas de `Thread`, Java también incluye una interfaz estándar denominada `Runnable`:

```
public interface Runnable {
  public abstract void run();
}
```

De ahí que cualquier clase que desee expresar ejecución concurrente deba implementar esta interfaz y proporcionar el método `run`. La clase `Thread` del Programa 7.1 hace justamente esto.

`Thread` es una subclase de `Object`. Proporciona diversos métodos constructores y un método `run`. Se pueden crear hilos de dos formas distintas utilizando los constructores (los grupos de hilos se considerarán posteriormente, en esta sección).

La primera consiste en declarar una clase como subclase de `Thread` y redefinir el método `run`. Así, posteriormente se podrá asignar y arrancar una instancia de esta subclase. Por ejemplo, en el caso del brazo de robot se puede asumir que se dispone de las siguientes clases y objetos:

```
public class InterfazUsuario
{
  public int nuevaIndicacion (int Dim) { ... }
  ...
}

public class Brazo
{
  public void mover(int dim, int pos) { ... }
}
InterfazUsuario IU = new InterfazUsuario();
Brazo Robot = new Brazo();
```

Programa 7.1. La clase Thread de Java.

```
public class Thread extends Object implements Runnable
{
   // Constructores

   public Thread();
   public Thread(String nombre);
   public Thread(Runnable objetivo);
   public Thread(Runnable objetivo, String nombre);

   public Thread(ThreadGroup grupo, String nombre);
   //            lanza SecurityException, IllegalThreadStateException
   public Thread(ThreadGroup grupo, Runnable objetivo);
   //            lanza SecurityException, IllegalThreadStateException
   public Thread(ThreadGroup grupo, Runnable objetivo, String nombre);
   //            lanza SecurityException, IllegalThreadStateException

   public void run();
   public native synchronized void start();
   //      lanza IllegalThreadStateException

   public static Thread currentThread();

   public final void join() throws InterruptedException;

   public final native boolean isAlive();

   public void destroy();
   //      lanza SecurityException;

   public final void stop();
   //      lanza SecurityException
   // EN DESUSO

   public final synchronized void stop(Throwable o);
   //      lanza SecurityException, NullPointerException
   // EN DESUSO

   public final void setDaemon();
   //      lanza SecurityException, IllegalThreadStateException

   public final boolean isDaemon();

   // a lo largo de este libro se introducirán otros métodos.
   // para una descripción completa de la clase véase el Apéndice A

   // Obsérvese que las excepciones de tiempo de ejecución no aparecen
   // como parte de la especificación de cada método, sino como comentarios.
}
```

Con las clases anteriores bajo alcance, el siguiente código declarará las clases que podrán ser utilizadas para representar los tres controladores:

```
public class Control extends Thread
{
  private int dim;

  public Control(int Dimension) // constructor
  {
     super();
     dim = Dimension;
  }

  public void run()
  {

     int posicion = 0;
     int indicacion;

     while(true)
     {
        Robot.mover(dim, posicion);
        indicacion = IU.nuevaIndicacion(dim);
        posicion = posicion + indicacion;
     }
  }
}
```

Ahora se pueden crear los tres controladores:

```
final int xPlano = 0; // final indica una constante
final int yPlano = 1;
final int zPlano = 2;

Control C1 = new Control(xPlano);
Control C2 = new Control(yPlano);
Control C3 = new Control(zPlano);
```

Hasta aquí se han creado los hilos, las variables declaradas han sido inicializadas, y se ha invocado a los métodos constructores de las clases `Control` y `Thread`. (En Java esto se conoce como estado *new*.) Sin embargo, el hilo no comienza su ejecución hasta que se invoca el método `start`:[2]

[2] El método `start` tiene los modificadores native y synchronize. El modificador native indica que el cuerpo del método viene en un *lenguaje diferente*. El modificador synchronize se utiliza para obtener exclusión mutua sobre el objeto. Se considerará en detalle en la Sección 8.8.

```
C1.start();
C2.start();
C3.start();
```

Fíjese en que si se invoca explícitamente al método `run`, entonces el código se ejecuta secuencialmente.

La segunda forma de crear un hilo es declarar una clase que implementa la interfaz `Runnable`, y después asignar una instancia de la clase que se pasa como argumento durante la creación de un objeto hilo. Recuerde que los hilos Java no se crean automáticamente cuando se crean sus objetos asociados, sino que deben ser creados explícitamente, e incluso iniciados, mediante el método `start`.

```
public class Control implements Runnable
{

  private int dim;

  public Control(int Dimension) // constructor
  {
     dim = Dimension;
  }

  public void run()
  {
     int posicion = 0;
     int indicacion;

     while(true)
     {
        Robot.mover(dim, posicion);
        indicacion = IU.nuevaIndicacion(dim);
        posicion = posicion + indicacion;
     }
  }
}
```

Ahora se pueden crear los tres controladores:

```
final int xPlano = 0;
final int yPlano = 1;
final int zPlano = 2;

Control C1 = new Control(xPlano); // todavía no se ha creado el hilo
```

```
Control C2 = new Control(yPlano);
Control C3 = new Control(zPlano);
```

y después se puede asociarlos con hilos e iniciarlos:

```
// se pasa a los constructores una instancia Runnable y se crean los hilos
Thread X = new Thread(C1);
Thread Y = new Thread(C2);
Thread Z = new Thread(C2);
X.start(); // hilo iniciado
Y.start();
Z.start();
```

Como Ada, Java permite la creación dinámica de hilos, pero (a diferencia de Ada) permite también el paso de datos arbitrarios como parámetros (mediante los constructores). Por supuesto, todos los objetos se pasan por referencia, de modo análogo a las variables acceso de Ada.

Aunque Java permite crear jerarquías de hilos y de grupos de hilos, no existe el concepto de maestro o guardián. Esto es así porque Java depende de la recolección de basura para eliminar los objetos fuera de alcance. El programa principal es una excepción, dado que sólo finaliza cuando lo han hecho ya todos sus hilos de usuario (user).

Un hilo puede esperar la terminación de otro hilo (el objetivo) invocando el método `join` del objeto hilo objetivo. Más aún, el método `isAlive` permite al hilo determinar si el hilo objetivo ha terminado.

Identificación de hilos

Existen dos formas de identificar los hilos. Si el código del hilo es una subclase de la clase `Thread`, se puede definir un identificador a un hilo de la siguiente forma:

```
Thread hiloID;
```

Gracias a la semántica de referencias de Java, es posible asignar calquier subclase de `Thread` a este objeto. Observe que donde el código del hilo se pasa por un constructor con interfaz `Runnable`, el identificador del hilo será el objeto hilo, y no el objeto que proporciona la interfaz. Para definir un identificador que identificase al objeto que proporciona el código, se necesita implementar una interfaz `Runnable`.

```
Runnable codigoHiloID;
```

Sin embargo, una vez que se ha obtenido una referencia al objeto `Runnable`, poco se puede hacer con ella explícitamente. Todas las operaciones relacionadas con el hilo vienen dadas por la clase `Thread`.

La identidad del hilo en ejecución puede hallarse con el método `currentThread`. Este método tiene un modificador `static`, lo que significa que existe un único método para todas las

instancias de objetos `Thread`, y por tanto el método siempre puede ser invocado utilizando la clase `Thread`.

Finalización de hilos

Un hilo de Java puede finalizar de distintas formas:

(1) Al completarse la ejecución de su método `run` normalmente, o como resultado de una excepción sin manejar.

(2) Por llamada de su método `stop`, en cuyo caso se para el método `run` y la clase hilo «limpia» antes de la finalización del hilo (libera los bloqueos que mantiene y ejecuta cualquier cláusula final); a partir de ahí, el objeto hilo es puesto a disposición del recolector de basura. Si se pasa como parámetro de `stop` un objeto lanzable (`Throwable`), se pasará al hilo objetivo. Esto permite una salida más elegante del método de ejecución y una «limpieza» posterior más efectiva.[3] El método `stop` es inherentemente inseguro, ya que libera bloqueos sobre objetos y puede dejar a estos objetos en estados inconsistentes. Por esta razón, últimamente se considera un método obsoleto (lo que en Java se conoce como «en desuso» o *deprecated*), y por lo tanto no debería utilizarse.

(3) Por la invocación de su método `destroy`: finaliza el hilo sin dar oportunidad al hilo objeto de «limpiar». Parece ser que nunca se ha implementado este método en la máquina virtual de Java.

Aunque (3) parece indicar que cualquier hilo puede destruir a otro, el programa puede proporcionar cierta protección contra esto mediante los mecanismos siguientes:

- Redefinición del método en la creación del nuevo hilo, y realización de alguna acción alternativa dentro del método.
- Utilización de grupos de hilos para restringir el acceso.
- Encapsulamiento del hilo dentro de otra clase, permitiendo sólo ciertas operaciones sobre el hilo.

Con el método `stop` sólo se pueden utilizar las dos últimas aproximaciones, al estar etiquetado como `final` y, por tanto, no poder ser redefinido.

Los hilos Java pueden ser de dos tipos: **de usuario** (user) y **daemon**. Los hilos daemon sirven para proporcionar servicios generales, que habitualmente nunca terminan. Así pues, sólo terminan cuando todos los hilos de usuario ya han terminado, de modo que se pueda finalizar el programa principal. Los hilos daemon proporcionan la misma funcionalidad que la opción «or terminate» de Ada en la instrucción «select» (véase la Sección 9.4.2). Antes de comenzar el hilo, habrá que invocar el método `setDaemon`.

[3] De hecho, cuando `stop` no tiene parámetros, se lanza la excepción `ThreadDeath`, que es una subclase de `Error`, y que por tanto no podrá ser interceptada por el programa (véase la Sección 6.3.2).

Grupos de hilos

Los grupos de hilos permiten agrupar conjuntos de hilos para así poder manipularlos como un grupo en lugar de individualmente. También suponen una forma de restringir quién hace qué a cuál de los hilos. En Java, todo hilo es miembro de algún grupo de hilos; por ello, existe un grupo por defecto asociado al programa principal al que pertenecen todos los demás hilos, a no ser que se especifique otra cosa. Los grupos de hilos se representan por la clase que aparece en el Programa 7.2.

Programa 7.2. Un extracto de la clase Java para grupos de hilos.

```
public class ThreadGroup {

    // Construye un nuevo grupo de hilos. El padre de este nuevo
    // grupo es el grupo del hilo actualmente en ejecución
    public ThreadGroup(String nombre);

    // Crea un nuevo grupo de hilos. El padre de este nuevo grupo es
    // el grupo de hilos especificado.
    public ThreadGroup(ThreadGroup padre, String nombre);
    //        lanza SecurityException

    // Devuelve el padre del grupo de hilos en ejecución
    public final ThreadGroup getParent();

    // Comprueba si este grupo es un grupo de hilos daemon
    public final boolean isDaemon();

    // Comprueba si el grupo ha sido destruido
    public synchronized boolean isDestroyed();

    // Cambia el estatus daemon de este grupo de hilos
    public final void setDaemon(boolean daemon);
    //                lanza SecurityException

    // Comprueba si este grupo es el que se da como argumento
    // o uno de sus grupos de hilos ancestros
    public final boolean parentOf(ThreadGroup g);

    // Determina si el hilo actualmente en ejecución tiene permiso para
    // modificar este grupo de hilos
    public final void checkAccess();

    // Devuelve una estimación del número de hilos activos en este
    // grupo de hilos
    public int activeCount();
```

(Continuación)

```
    // Devuelve una estimación del número de grupos activos en este
    // grupo de hilos
    public int activeGroupCount();

    // Finaliza todos los procesos en este grupo de hilos
    public final void stop();
    //                lanza SecurityException

    // Destruye este grupo de hilos y todos sus subgrupos
    public final void destroy();
    //                lanza IllegalThreadStateException

  // A lo largo de este libro se presentarán otros métodos. Para
  // una descripción completa de esta clase, véase el Apéndice A.
}
```

Cuando un hilo crea un nuevo grupo de hilos, lo hace desde dentro de un grupo. Por tanto, el nuevo grupo de hilos es hijo del grupo de hilos actual, a no ser que se pase como parámetro del constructor un grupo de hilos diferente. Utilizando estos dos métodos constructores, se pueden crear jerarquías de grupos de hilos.

Al crear hilos se pueden ubicar explícitamente en un grupo concreto usando el constructor apropiado de la clase `Thread`. Las peticiones de `stop` o `destroy` de un grupo de hilos se aplicarán (si lo permite el gestor de seguridad) a todos los hilos del grupo.

Excepciones relacionadas con los hilos

En el Capítulo 6 se presentan las excepciones de ejecución (`RuntimeExceptions`) de Java. Las relevantes para los hilos son:

`IllegalThreadStateException`; se lanza cuando:

- Se invoca el método `start` y el hilo ya ha sido creado.
- Se ha invocado el método `setDaemon` y el hilo ya ha sido creado.
- Se intenta destruir un grupo de hilos no vacío.
- Se intenta ubicar cierto hilo en un grupo de hilos (por medio del constructor `Thread`), y el grupo de hilos ya ha sido destruido.

`SecurityException`; se lanza desde el gestor de seguridad[4] cuando:

[4] El gestor de seguridad de Java no se ve en este libro, ya que es más propio del uso de Java en un sistema distribuido abierto.

- Ha sido invocado el constructor `Thread` por un objeto que no tiene permiso para ello.
- Ha sido invocado el método `stop` o `destroy` por un objeto que pertenece a un hilo que no tiene permiso para ello.
- Ha sido invocado el constructor `ThreadGroup` y se ha pedido que el grupo padre del grupo recién creado sea uno sobre el que el grupo que hace la llamada no tiene permiso.
- Ha sido invocado el método `stop` o `destroy` de un grupo de hilos desde un grupo que no tiene los permisos adecuados.

`NullPointerException`; se lanza cuando:

- Se pasa un puntero nulo al método `stop`.
- Se pasa un puntero nulo al constructor `ThreadGroup` para el grupo padre.

`InterruptedException`; se lanza si un hilo que ha invocado un método `join` es «despertado» por el hilo que está siendo interrumpido en vez de por el hilo objetivo que está terminando (véase la Sección 10.9).

Hilos en tiempo real

El lenguaje de programación Java no dispone de muchos mecanismos de programación de sistemas de tiempo real; de ahí la importancia del desarrollo de Java para tiempo real. Java para tiempo real presenta varias clases de hilos nuevas, que se tendrán en cuenta en la Sección 12.7.3.

7.3.8 Comparación de Ada, Java y occam2

En las secciones anteriores se ha visto la estructura básica de los modelos de procesos de occam2, Ada y Java, y aunque sin comunicación entre procesos no se pueden representar programas interesantes, ya es posible mostrar algunas de las diferencias entre los modelos de procesos de occam2, Ada y Java. Los mecanismos de sincronización se mostrarán más adelante. En occam2, cuando se efectúan dos llamadas seguidas a procedimientos, se ejecutarán secuencial o concurrentemente:

```
SEQ     -- forma secuencial
  proc1
  proc2

PAR     -- forma concurrente
  proc1
  proc2
```

Para lograr concurrencia en Ada es preciso introducir dos tareas:

```
begin      -- forma secuencial
```

```
  Proc1;
  Proc2;
end;

declare   -- forma concurrente
  task Uno;
  task Dos;
  task body Uno is
  begin
    Proc1;
  end Uno;
  task body Dos is
  begin
    Proc2;
  end Dos;
begin
  null;
end;
```

Aunque, en realidad, sería posible utilizar sólo una tarea y hacer que el propio bloque hiciera la otra llamada.

```
declare
  task Uno;
  task body Uno is
  begin
    Proc1;
  end Uno;
begin
  Proc2;
end;
```

Pero esta forma ha perdido la simetría lógica del algoritmo, y no se recomienda.

En Java, para cada procedimiento es preciso crear un objeto que implemente la interfaz `Runnable`.

```
public class Proc1 implements Runnable
{
  public void run()
  {
    // código para proc1
  }
}
```

```
public class Proc2 implements Runnable
{
  public void run()
  {
    // código para proc2
  }
}

Proc1 P1 = new Proc1();
Proc2 P2 = new Proc2();

// forma secuencial
{
  P1.run;
  P2.run;
}

//forma concurrente
Thread T1 = new Thread(P1);
Thread T2 = new Thread(P2)
T1.start();
T2.start();
```

7.3.9 Ejecución concurrente en POSIX

POSIX para tiempo real proporciona tres mecanismos para crear actividades concurrentes. El primero es el mecanismo tradicional de UNIX, `fork`, junto a la llamada asociada `wait`, que permite crear una copia completa de un proceso. Los detalles de fork se encuentran en cualquier libro de texto sobre sistemas operativos –por ejemplo, Silberschatz y Galvin (1994)–, y por ello no serán comentados aquí. En la Sección 9.5 aparece un interesante ejemplo de creación de procesos utilizando `fork`. El segundo mecanismo es la llamada al sistema `spawn`, que no es más que la combinación de `fork` y `join`.

Programa 7.3. Una interfaz POSIX C para hilos.

```
typedef ... pthread_t; /* no se definen los detalles */
typedef ... pthread_attr_t;
typedef ... size_t;

int pthread_attr_init(pthread_attr_t *attr);
   /* inicializa un atributo de hilo apuntado por attr a
      sus valores por defecto */
```

(Continuación)

```
int pthread_attr_destroy(pthread_attr_t *attr);
   /* destruye un atributo de hilo apuntado por attr*/

int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t tamañopila);
   /* establece el tamaño de pila sobre un atributo de hilo */

int pthread_attr_getstacksize(const pthread_attr_t *attr,
                              size_t *tamañopila);
   /* obtiene el tamaño de pila del atributo de hilo */

int pthread_attr_setstackaddr(pthread_attr_t *attr,
                              void *direccionpila);
   /* establece la dirección de pila sobre un atributo de hilo */

int pthread_attr_getstackaddr(const pthread_attr_t *attr,
                              void **direccionpila);
   /* obtiene la dirección de pila de un atributo de hilo */

int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int estadodesunido);
   /* establece el estado de desunido del atributo */

int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *desunido);
   /* obtiene el estado de desunido del atributo */

int pthread_create(pthread_t *hilo, const pthread_attr_t *attr,
                   void *(*rutina_arranque)(void *), void *arg);
   /* crea un nuevo hilo con el atributo dado e invoca a la
      rutina de arranque dada con el argumento dado */

int pthread_join(pthread_t hilo, void **valor_ptr);
   /* suspende el hilo que hace la llamada hasta que el hilo referenciado
      haya terminado, cualquier valor devuelto es apuntado por valor_ptr */

int pthread_exit(void *valor_ptr);
   /* finaliza el hilo que hace la llamada y hace disponible valor_ptr
      para cualquier hilo que con el que se una (join) */

int pthread_detach(pthread_t hilo);
   /* el espacio de almacenamiento asociado con el hilo dado puede ser
      reclamado cuando el hilo termine */

pthread_t pthread_self(void);
   /* devuelve el id del hilo que hace la llamada */

int pthread_equal(pthread_t t1, pthread_t t2);
   /* compara dos ids de hilos */

/* Todas las funciones enteras anteriores devuelven 0 con éxito,
   y en caso contrario devuelven un número de error */
```

POSIX para tiempo real permite que cada proceso pueda contener diversos «hilos» de ejecución. Todos estos hilos tienen acceso a las mismas direcciones de memoria, y se ejecutan en un único espacio de direcciones. Por tanto, pueden ser comparados con las tareas de Ada, los hilos de Java, y los procesos de occam2. El Programa 7.3 muestra la interfaz C primaria para la creación de hilos en POSIX.

Todos los hilos POSIX tienen atributos (por ejemplo, su tamaño de pila). Para manipular estos atributos es preciso definir un objeto atributo (de tipo `pthread_attr_t`), y después llamar a las funciones que establecen y obtienen los atributos. Una vez establecido el atributo correcto, puede crearse un hilo y pasarle el atributo apropiado. Todo hilo que se crea tiene asociado un identificador (de tipo `pthread_t`), que es único con respecto a otros hilos del mismo proceso. Un hilo puede obtener su propio identificador (mediante `pthread_self`).

Un hilo puede ser elegido para ejecución tan pronto como es creado por `pthread_create`; no hay un equivalente al estado de activación de Ada o al estado nuevo de Java. Esta función toma cuatro argumentos puntero: un identificador de hilo (devuelto por la función), un conjunto de atributos, una función que representa el código de ejecución del hilo, y el conjunto de parámetros que se pasan a la función cuando se invoca. El hilo puede finalizar normalmente al retornar de su `rutina_arranque`, llamando a `pthread_exit`, o al recibir alguna señal que se le envíe (véase la Sección 10.6). También puede ser abortado utilizando `pthread_cancel`. Un hilo puede esperar a que otro hilo finalice por medio de la función `pthread_join`.

Finalmente, la actividad de limpieza después de la ejecución de un hilo y la liberación de su almacenamiento se denominan **desunión** (detaching). Existen dos formas de conseguir esto: llamando a la función `pthread_join` y esperando a que el hilo finalice, o estableciendo el atributo de desunión del hilo (bien en el momento de creación, bien invocando la función `pthread_detach`). Al establecer el atributo de desunión del hilo, no se efectuará un join, y el espacio de almacenamiento se reclamará automáticamente cuando finalice el hilo.

En muchos sentidos, la interfaz POSIX para hilos es similar a la que utilizaría un compilador para interactuar con su sistema de soporte de ejecución. De hecho, el sistema de soporte de ejecución de Ada podría perfectamente ser implementado utilizando hilos POSIX. La ventaja de proporcionar abstracciones de lenguaje de alto nivel (como las de Ada, Java y occam2) es que, utilizando la interfaz, se eliminan posibilidades de error.

Para ilustrar el fácil uso del mecanismo de creación de hilos POSIX, se muestra a continuación el programa del brazo de robot.

```
#include <pthread.h>

pthread_attr_t atributos;
pthread_t xp, yp, zp;

typedef enum {xplano, yplano, zplano} dimension;

int nueva_indicacion(dimension D);
```

```
void mover_brazo(dimension D, int P);

void controlador(dimension *dim)
{
  int posicion, indicacion;

  posicion = 0;
  while (1) {
    mover_brazo(*dim, posicion);
    indicacion = nueva_indicacion(*dim);
    posicion = posicion + indicacion;
  }
  /* nota, no hay llamada a pthread_exit, el proceso no finaliza */
}

#include <stdlib.h>
int main() {
  dimension X, Y, Z;
  void *resultado;

  X = xplano,
  Y = yplano;
  Z = zplano;
  if(pthread_attr_init(&atributos) != 0)
    /* establecer atributos por defecto */
    exit(EXIT_FAILURE);
  if(pthread_create(&xp, &atributos,
                    (void *)controlador, &X) != 0)
    exit(EXIT_FAILURE);
  if(pthread_create(&yp, &atributos,
                    (void *)controlador, &Y) != 0)
    exit(EXIT_FAILURE);
  if(pthread_create(&zp, &atributos,
                    (void *)controlador, &Z) != 0)
    exit(EXIT_FAILURE);
  pthread_join(xp, (void **)&resultado);
  /* preciso para bloquear el programa principal */

  exit(EXIT_FAILURE);
  /* salida de error, el programa no debería finalizar */
}
```

Cada atributo de hilo se crea con ciertos valores por defecto. Las llamadas a `pthread_create` crean cada una de las instancias del hilo `controlador`, y pasan un parámetro que indica su dominio de operación. Si el sistema operativo devuelve cualquier error, el programa finaliza invocando a la rutina `exit`. Observe que un programa que finaliza con hilos pendientes da como resultado la finalización de estos hilos. De ahí que sea necesario que el programa principal realice una llamada de sistema `pthread_join` aunque los hilos no terminen.

En la Sección 6.1.1 se presenta un modo de manejar de los errores devueltos de POSIX. Se parte de que cada llamada tiene definida una macro que comprueba los valores devueltos, e invoca, si es preciso, a las rutinas de manejo de errores. Así, es posible escribir el código anterior de otra forma más legible:

```
int main() {
  dimension X, Y, Z;
  void *resultado;

  X = xplano,
  Y = yplano;
  Z = zplano;

  PTHREAD_ATTR_INIT(&atributos);

  PTHREAD_CREATE(&xp, &atributos, (void *)controlador, &X);
  PTHREAD_CREATE(&yp, &atributos, (void *)controlador, &Y);
  PTHREAD_CREATE(&zp, &atributos, (void *)controlador, &Z);

  PTHREAD_JOIN(xp, (void **)&resultado);
}
```

Finalmente debe notarse que la división (fork) de un proceso (programa) que contiene múltiples hilos no es directa, ya que algunos hilos del proceso pueden mantener recursos o estar ejecutando llamadas de sistema. El estándar POSIX especifica que el proceso hijo tendrá un único hilo –véase POSIX 1003.1c (IEEE, 1995)–.

7.4 Un sistema embebido sencillo

Para mostrar algunas de las ventajas y desventajas de la programación concurrente, se considerará un sencillo sistema embebido. La Figura 7.6 resume este sistema simple: un proceso *T* hace lecturas de un conjunto de termopares (de un conversor analógico-digital; ADC), y realiza cambios en un calefactor (por medio de un conmutador controlado digitalmente). El proceso *P* realiza una función similar, pero para la presión (utiliza un conversor digital-analógico; DAC). *T* y *P* comunican información a *S*, que presenta las medidas a un operador por pantalla. Obsérvese que

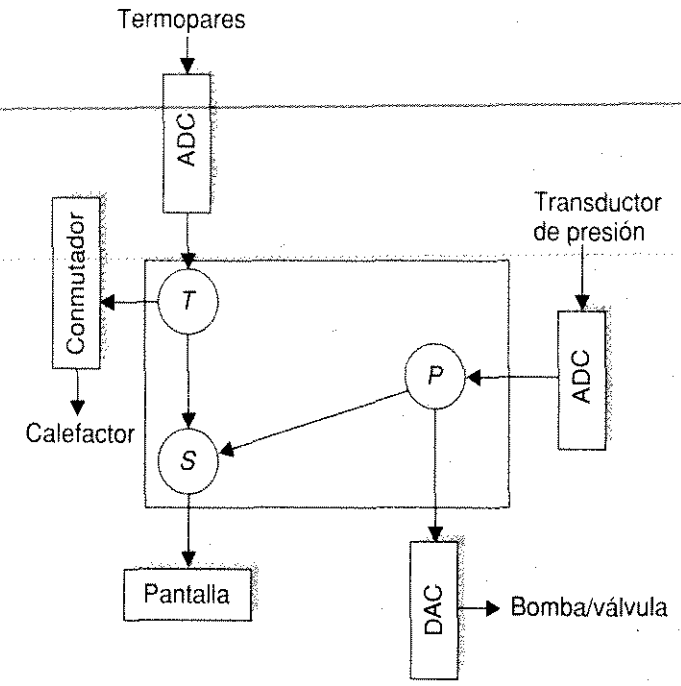


Figura 7.6. Un sistema embebido sencillo.

P y *T* son activos; *S* es un recurso (simplemente responde a las peticiones de *T* y *P*): puede ser implementado como un recurso protegido, o como un servidor, si la interacción con el usuario es más extensa.

El objetivo principal de este sistema embebido es mantener la temperatura y la presión de algún proceso químico dentro de ciertos límites definidos. Un sistema real de este tipo sería claramente más complejo, y permitiría, por ejemplo, que el operador cambiara los límites. Sin embargo, incluso para este sistema, la implementación podría seguir tres caminos:

(1) Se utiliza un único programa, ignorando la evidente concurrencia de *T*, *P* y *S*. No se necesita soporte del sistema operativo.

(2) *T*, *P* y *S* se escriben en un lenguaje de programación secuencial (bien como programas separados, bien como procedimientos distintos en el mismo programa), y se utilizan las primitivas del sistema operativo para la creación e interacción de los programas/procesos.

(3) Se utiliza un único programa concurrente que mantiene la estructura lógica de *T*, *P* y *S*. No se necesita un soporte directo del sistema operativo, aunque es necesario un sistema de soporte de ejecución.

Para ilustrar estas soluciones, considérese el código Ada que implementa el sistema embebido. Para simplificar la estructura del software de control, se supone que ya han sido implementados los siguientes paquetes pasivos:

```
package Tipos_Datos is
   -- definiciones de los tipos necesarios
   type Lectura_Temp is new Integer range 10..500;
```

```
  type Lectura_Presion is new Integer range 0..750;
  type Ajuste_Calefactor is (On, Off);
  type Ajuste_Presion is new Integer range 0..9;
end Tipos_Datos;

with Tipos_Datos; use Tipos_Datos;
package IO is
  -- procedimientos para intercambio de datos con el entorno
  procedure Leer(LecT : out Lectura_Temp); -- del ADC
  procedure Leer(LecP : out Lectura_Presion);
    -- obsérvese que esto es un ejemplo de sobrecarga; se definen
    -- dos lecturas, pero tienen tipos de parámetros distintos
    -- éste es también el caso con las escrituras siguientes
  procedure Escribir(AjuC : Ajuste_Calefactor);     -- al conmutador.
  procedure Escribir(AjuP : Ajuste_Presion);        -- al DAC
  procedure Escribir(LecT : Lectura_Temp);          -- a la pantalla
  procedure Escribir(LecP : Lectura_Presion);       -- a la pantalla
end IO;

with Tipos_Datos; use Tipos_Datos;
package Procedimientos_Control is
  -- procedimientos que convierten una lectura en
  -- un ajuste apropiado de salida
  procedure Convertir_Temp(LecT : Lectura_Temp;
                           AjuC : out Ajuste_Calefactor);
  procedure Convertir_Presion(LecP : Lectura_Presion;
                              AjuP : out Ajuste_Presion);
end Procedimientos_Control;
```

Solución secuencial

Un sencillo programa secuencial de control podría tener la siguiente estructura:

```
with Tipos_Datos; use Tipos_Datos;
with IO; use IO;
with Procedimientos_Control; use Procedimientos_Control;
procedure Controller is
  LecT : Lectura_Temp;
  LecP : Lectura_Presion;
  AjuC : Ajuste_Calefactor;
  AjuP : Ajuste_Presion;
begin
  loop
```

```
      Leer(LecT);                    -- del ADC
      Convertir_Temp(LecT,AjuC);     -- convertir lectura en ajuste
      Escribir(AjuC);                -- al conmutador
      Escribir(LecT);                -- a la pantalla
      Leer(LecP);                    -- como antes, para la presión
      Convertir_Presion(LecP,AjuP);
      Escribir(AjuP);
      Escribir(LecP);
  end loop; -- bucle indefinido, común en software embebido
end Controller;
```

Este código tiene el problema inmediato de que las lecturas de temperatura y presión se toman con la misma frecuencia, lo que podría no coincidir con los requisitos. El uso de contadores y sentencias **if** apropiadas mejorará la situación, aunque puede que todavía sea necesario dividir las secciones computacionalmente intensivas (los procedimientos de conversión `Convertir_Temp` y `Convertir_Presión`) en un conjunto de acciones distintas, y entremezclar estas acciones de manera que se cumpla el balance de trabajo requerido. Incluso con esto, esta estructura de programa presenta una seria limitación: mientras se espera a leer la temperatura no se presta atención a la presión (y viceversa). Más aún, si hay un fallo en el sistema, y como resultado de ello nunca se devuelve control desde la lectura de la temperatura, entonces, además de este problema, no se realizaría ninguna lectura adicional de la presión.

Se puede mejorar este programa secuencial incluyendo en el paquete `IO` dos funciones lógicas, `Temp_Lista` y `Pres_Lista`, que indican la disponibilidad de la lectura para un elemento. El programa de control pasa a ser

```
with Tipos_Datos; use Tipos_Datos;
with IO; use IO;
with Procedimientos_Control; use Procedimientos_Control;
procedure Controller is
  LecT : Lectura_Temp;
  LecP : Lectura_Presion;
  AjuC : Ajuste_Calefactor;
  AjuP : Ajuste_Presion;
  Temp_Lista, Pres_Lista : Boolean;
begin
  loop
    ...
    if Temp_Lista then
      Leer(LecT);
      Convertir_Temp(LecT,AjuC);
      Escribir(AjuC);   -- asumiendo que la escritura es fiable
      Escribir(LecT);
    end if;
```

```
      if Pres_Lista then
        Leer(LecP);
        Convertir_Presion(LecP,AjuP);
        Escribir(AjuP);
        Escribir(LecP);
      end if;
    end loop;
end Controller;
```

Esta solución es más fiable, aunque el programa emplea ahora una alta proporción de su tiempo en un bucle «ocupado», intentando ver si los aparatos de entrada están disponibles. Por lo general, estas esperas son inaceptablemente ineficientes, retardan al procesador, y dificultan la imposición de una disciplina de cola sobre las peticiones en espera. Además, los programas basados en espera «ocupada» son difíciles de diseñar y comprender, y es complicado probar su corrección.

La principal crítica al programa secuencial proviene de que no se reconoce el hecho de que los ciclos de presión y temperatura son subsistemas completamente independientes. En un entorno de programación concurrente, esto puede rectificarse codificando cada sistema como un proceso (en terminología Ada, una tarea).

Utilización de primitivas del sistema operativo

Considere un sistema operativo tipo POSIX que permita la creación y comienzo de nuevos procesos/hilos llamando al siguiente subprograma Ada:

```
package Interfaz_Sistema_Operativo is
  type Hilo_ID is private;
  type Hilo is access procedure; -- un tipo apuntador

  function Crear_Hilo(Code : Hilo) return Hilo_ID;
  -- otros subprogramas para la interacción de hilos
private
  type Hilo_ID is ...;
end Interfaz_Sistema_Operativo;
```

Ahora, el sistema embebido puede implementarse de la siguiente forma: primero, se ubican los dos procedimientos controladores en un paquete:

```
package Procesos is
  procedure Controlador_Presion;
  procedure Controlador_Temp;
end Procesos;

with Tipos_Datos; use Tipos_Datos;
with IO; use IO;
```

```
with Procedimientos_Control; use Procedimientos_Control;
package body Procesos is
   procedure Controlador_Temp is
      LecT : Lectura_Temp;
      AjuC : Ajuste_Calefactor;
   begin
      loop
         Leer(LecT);
         Convertir_Temp(LecT,AjuC);
         Escribir(AjuC);
         Escribir(LecT);
      end loop;
   end Controlador_Temp;

   procedure Controlador_Presion is
      LecP : Lectura_Presion;
      AjuP : Ajuste_Presion;
   begin
      loop
         Leer(LecP);
         Convertir_Presion(LecP,AjuP);
         Escribir(AjuP);
         Escribir(LecP);
      end loop;
   end Controlador_Presion;
end Procesos;
```

Ahora se da el procedimiento `Controlador`:

```
with Interfaz_Sistema_Operativo; use Interfaz_Sistema_Operativo;
with Procesos; use Procesos;

procedure Controlador is
   Tc, Pc: Hilo_Id;
begin
   -- crea los hilos
   -- 'Access devuelve un apuntador al procedimiento
   Tc := Crear_Hilo(Controlador_Temp'Access);
   Pc := Crear_Hilo(Controlador_Presion'Access);
end Controlador;
```

Los procedimientos `Controlador_Temp` y `Controlador_Presion` se ejecutan concurrentemente, y cada uno contiene un bucle indefinido dentro del cual se define el ciclo de con-

trol. Mientras un hilo está suspendido esperando por una lectura, el otro puede estar ejecutándose; si ambos están suspendidos, no hay ningún bucle «ocupado» en ejecución.

Aunque esta solución presenta ventajas respecto a la solución secuencial, la falta de soporte del lenguaje para expresar concurrencia implica que el programa puede resultar difícil de escribir y mantener. Para el sencillo ejemplo presentado anteriormente, la complejidad añadida es aceptable. Sin embargo, para sistemas grandes con muchos procesos concurrentes y posibles interacciones complejas entre ellos, el tener una interfaz procedural oscurece la estructura del programa. Por ejemplo, no está claro qué procedimientos son realmente procedimientos y cuáles se pretende que sean actividades concurrentes.

Utilización de un lenguaje de programación concurrente

En un lenguaje de programación concurrente, las actividades concurrentes se pueden identificar explícitamente en el código:

```
with Tipos_Datos; use Tipos_Datos;
with IO; use IO;
with Procedimientos_Control; use Procedimientos_Control;
procedure Controlador is
  task Controlador_Temp;
  task Controlador_Presion;

  task body Controlador_Temp is
    LecT : Lectura_Temp; AjuC : Ajuste_Calefactor;
  begin
    loop
      Leer(LecT);
      Convertir_Temp(LecT,AjuC);
      Escribir(AjuC);
      Escribir(LecT);
    end loop;
  end Controlador_Temp;

  task body Controlador_Presion is
    LecP : Lectura_Presion; AjuP : Ajuste_Presion;
  begin
    loop
      Leer(LecP);
      Convertir_Presion(LecP,AjuP);
      Escribir(AjuP);
      Escribir(LecP);
    end loop;
  end Controlador_Presion;
```

```
begin
  null;    -- Controlador_Temp y Controlador_Presion
           -- han comenzado sus ejecuciones
end Controlador;
```

La lógica de la aplicación queda ahora reflejada en el código; el paralelismo inherente del dominio se representa en el programa con la ejecución de tareas concurrentes.

Aunque es una mejora, esta solución de dos tareas mantiene un problema importante. Tanto `Controlador_Temp` como `Controlador_Presion` envían datos a la pantalla, pero la pantalla es un recurso que sólo puede ser accedido de manera razonable por un proceso cada vez. En la Figura 7.6, el control de la pantalla corresponde a una tercera entidad (S) que será representada en el programa `Controlador_Pantalla`. Esta entidad puede ser un servidor o un recurso protegido (dependiendo de la definición completa del comportamiento requerido para `Controlador_Pantalla`). Esto ha convertido el problema de acceso concurrente a un recurso pasivo en uno de comunicación entre tareas, o al menos de comunicación entre una tarea y alguna otra primitiva de concurrencia: es necesario que las tareas `Controlador_Temp` y `Controlador_Presion` pasen datos a `Controlador_Pantalla`. Más aún, `Controlador_Pantalla` debe garantizar que trata una única petición en cada momento. Estos requisitos y dificultades son de capital importancia en el diseño de lenguajes de programación concurrentes, y serán considerados en los siguientes capítulos.

Resumen

Los dominios de aplicación de la mayoría de los sistemas de tiempo real son inherentemente paralelos. Por tanto, la inclusión de la noción de proceso en los lenguajes de programación de tiempo real supone una enorme diferencia para la potencia descriptiva y la facilidad de uso del lenguaje. Estos factores, por su parte, contribuyen significativamente a la reducción de los costes de producción del software, a la vez que mejoran la fiabilidad del sistema final.

Sin concurrencia, el software tiene que ser construido con un único bucle de control. La estructura de este bucle no puede reflejar la diferenciación lógica entre los componentes del sistema. Particularmente difícil es aportar requisitos de fiabilidad y temporización orientada al proceso sin que la noción de proceso sea visible en el código.

La utilización de un lenguaje de programación concurrente conlleva, sin embargo, un coste. En concreto, se hace necesaria la utilización de un sistema de soporte de ejecución (o sistema operativo) que gestione la ejecución de los procesos del sistema.

El comportamiento de un proceso se describe mejor en términos de estados. En este capítulo se han comentado los siguientes estados:

- No existente.
- Creado.

- Inicializado.
- Ejecutable.
- Esperando terminación del dependiente.
- Esperando inicialización del hijo.
- Finalizado.

Dentro de los lenguajes de programación concurrentes, existe un conjunto de variaciones sobre el modelo de procesos adoptado. Estas variaciones pueden ser analizadas en relación con seis conceptos.

(1) Estructura: modelo de procesos estático o dinámico.

(2) Nivel: sólo procesos de alto nivel (plano) o multinivel (anidado).

(3) Inicialización: con o sin paso de parámetros.

(4) Granularidad: grano fino y grueso.

(5) Finalización.

- Natural.
- Suicidio.
- Aborto.
- Error sin tratar.
- Nunca.
- Cuando no se necesita más.

(6) Representación: corrutinas, fork/join, cobegin; declaraciones explícitas de procesos.

Occam2 utiliza una representación cobegin (`PAR`) con procesos estáticos anidados; se pueden pasar datos de inicialización a un proceso, aunque no se permiten las finalizaciones de aborto ni la de «ya no necesario». Ada y Java proporcionan un modelo dinámico que soporta tareas anidadas y un abanico de opciones de finalización. POSIX permite la creación dinámica de hilos con estructura plana; los hilos deben ser explícitamente finalizados o *matados*.

Lecturas complementarias

Andrews, G. A. (1991), *Concurrent Programming Principles and Practice*, Redwood City, CA: Benjamin/Cummings.

Ben-Ari, M. (1990), *Principles of Concurrent and Distributed Programming*, New York: Prentice Hall.

Burns, A., y Wellings, A. J. (1995), *Concurrency in Ada*, Cambridge: Cambridge University Press.

Burns, A. (1998), *Programming in occam2*, Reading: Addison-Wesley.

Butenhof, D. R. (1997), *Programming With Posix Threads*, Reading, MA: Addison-Wesley.

Galletly, J. (1990), *Occam2*, London: Pitman.

Hyde, P. (1999), *Java Threads Programming*, Indianapolis, IN: Sams Publishing.

Lea, D. (1999), *Concurrent Programming in Java: Design Principles and Patterns*, Reading, MA: Addison-Wesley.

Nichols, B., Buttlar, D., y Farrell, J. (1996), *POSIX Threads Programming*, Sebastopol, CA: O'-Reilly.

Oaks, A. y Wong, H. (1997), *Java Threads*, Sebastopol, CA: O'Reilly.

Silberschatz, A., y Galvin, P. A. (1998), *Operating System Concepts*, New York: John Wiley & Sons.

Ejercicios

7.1 Un sistema operativo concreto tiene una llamada de sistema (*EjecutarConcurrentemente*) que admite un array ilimitado. Cada elemento del array es un apuntador a un procedimiento sin parámetros. La llamada de sistema ejecuta concurrentemente todos los procedimientos, y devuelve control cuando todos han finalizado. Muestre cómo puede implementarse en Ada esta llamada de sistema utilizando las funcionalidades de tareas de Ada. Suponga que el sistema operativo y la aplicación se ejecutan en el mismo espacio de direcciones.

7.2 Escriba código Ada para la creación de un array de tareas en el que cada una de ellas tiene un parámetro que indica su posición en el array.

7.3 Muestre cómo puede implementarse *cobegin* en Ada.

7.4 ¿Se pueden implementar en Ada los métodos de creación de procesos *fork* y *join*, sin utilizar comunicación entre tareas?

7.5 ¿Cuántos procesos POSIX se crean con el procedimiento siguiente?

```
for(i=0; i<=10;i++) {
   fork();
}
```

7.6 Reescriba el sencillo sistema embebido que se muestra en la Sección 7.4, en Java, C y POSIX, y occam2.

7.7 Si un proceso multihilo ejecuta una llamada de sistema fork tipo POSIX, ¿cuántos hilos debería contener el proceso creado?

7.8 Muestre, utilizando procesos concurrentes, la estructura de un programa de control simple para el acceso de un aparcamiento de coches. Suponga que el aparcamiento de coches tiene una única barrera de entrada, una única de salida, y una señal de completo.

7.9 Explique, con la ayuda del siguiente programa, la interacción entre las reglas Ada para la finalización de tareas y su modelo de propagación de excepciones. Inclúyase una consideración del comportamiento del programa (salida) para valores iniciales de la variable C de 2,1 y 0.

```
with Ada.Text_Io; use Ada.Text_Io;
procedure Main is
   task A;

   task body A is
      C : Positive := Algun_Valor_Entero;
      procedure P(D : Integer) is
         task T;
         A : Integer;
         task body T is
         begin
            delay 10.0;
            Put("T Finalizado"); New_Line;
         end T;
      begin
         Put("P Comenzado"); New_Line;
         A := 42/D;
         Put("P Finalizado"); New_Line;
      end P;

   begin
      Put("A Comenzado"); New_Line;
      P(C-1);
      Put("A Finalizado"); New_Line;
   end A;

begin
   Put("Procedimiento principal comenzado"); New_Line;
exception
   when others =>
      Put("Procedimiento principal fallido"); New_Line;
end Main;
```

7.10 Para toda tarea en el siguiente programa Ada, indíquense su padre y su guardián (maestro), y, si es el caso, sus hijos y sus dependientes. Indíquese también cuáles son los dependientes de los procedimientos `Main` y `Jerarquia`.

```
procedure Main is
   procedure Jerarquia is
      task A;
      task type B;

      type Pb is access B;
      Punterob : Pb;

      task body A is
         task C;
         task D;
         task body C is
         begin
            -- incluye secuencia de instrucciones
            Punterob := new B;
         end C;
         task body D is
            Otro_Punterob : Pb;
         begin
            -- incluye secuencia de instrucciones
            Otro_Punterob := new B;
         end D;
      begin
         -- secuencia de instrucciones
      end A;

      task body B is
      begin
         -- secuencia de instrucciones
      end B;

   begin
      -- secuencia de instrucciones
   end Jerarquia;

begin
   -- secuencia de instrucciones
end Main;
```

7.11 ¿En qué medida puede utilizarse la Figura 7.2 para representar el diagrama de transición de estados de: (a) hilos POSIX, (b) procesos occam2 y (c) hilos Java?

7.12 Dado el siguiente código:

```
public class Calcular implements Runnable
```

```
{
    public void run()
    {
        /* cálculo largo */
    }
}

Calcular MiCalculo = new Calcular();
```

cuál es la diferencia entre:

```
MiCalculo.run();
```

y

```
new Hilo(MiCalculo).start();
```

7.13 Explique cómo se puede proteger a un hilo Java de ser destruido por un hilo cualquiera.

Capítulo 8

Comunicación y sincronización basada en variables compartidas

Las mayores dificultades asociadas con la programación concurrente surgen de la interacción de procesos. Casi nunca los procesos son independientes entre sí, como lo eran en el ejemplo sencillo del final del Capítulo 7. El comportamiento correcto de un programa concurrente depende estrechamente de la sincronización y la comunicación entre procesos. En su sentido más amplio, sincronizar es satisfacer las restricciones en el entrelazado de las acciones de diferentes procesos (por ejemplo, una acción particular de un proceso sólo ocurre después de una acción específica de otro proceso). El término se utiliza también en el sentido más restringido de llevar simultáneamente a dos procesos a estados predefinidos. Comunicar es pasar información de un proceso a otro. Los dos conceptos están ligados, puesto que algunas formas de comunicación requieren sincronización, y la sincronización puede ser considerada como comunicación sin contenido.

La comunicación entre procesos se basa normalmente o en el uso de **variables compartidas** o en el **paso de mensajes**. Las variables compartidas son objetos a los que puede acceder más de un proceso; la comunicación puede realizarse, por tanto, referenciando en cada proceso dichas variables cuando sea apropiado. El paso de mensajes implica el intercambio explícito de datos entre dos procesos mediante un mensaje que pasa de un proceso a otro siguiendo algún mecanismo. Hay que señalar que la elección entre variables compartidas y paso de mensajes deben realizarla los diseñadores de los lenguajes o de los sistemas operativos, y no implica que deba utilizarse un método particular de implementación. Las variables compartidas son más fáciles de soportar si hay memoria compartida entre procesos, pero pueden ser utilizadas incluso si el hardware incorpora un medio de comunicación. De forma similar, puede obtenerse una primitiva de paso de mensajes mediante memoria compartida o mediante una red física de paso de mensajes. Además, una aplicación puede ser programada en un estilo u otro y obtener la misma funcionalidad (Lauer y Needham, 1978). La sincronización y comunicación basadas en paso de mensajes se discuten en el Capítulo 9. Este capítulo se concentrará en las primitivas de sincronización y comunicación basadas en memoria compartida. En particular se verán los conceptos de espera ocupada, semáforos, regiones críticas condicionales, monitores, tipos protegidos y métodos sincronizados.

8.1 Exclusión mutua y condición de sincronización

Aunque las variables compartidas parecen una forma directa de pasar información entre procesos, su uso no restringido es poco fiable e inseguro, debido a los múltiples problemas de actualización. Considere dos procesos que actualizan una variable compartida, X, mediante la sentencia:

```
X:= X+1
```

En la mayoría del hardware esto no se ejecuta como una operación **indivisible** (atómica), sino que se implementará en tres instrucciones distintas:

(1) Carga el valor de X en algún registro (o en la parte superior de la pila).

(2) Incrementa el valor en el registro en 1.

(3) Almacena el valor del registro de nuevo en X.

Como ninguna de las tres operaciones es indivisible, dos procesos que actualicen la variable simultáneamente generarían un entrelazamiento que podría producir un resultado incorrecto. Por ejemplo, si X era originalmente 5, los dos procesos podrían cargar cada uno 5 en sus registros, incrementarlos, y almacenar entonces 6.

Una secuencia de sentencias que debe aparecer como ejecutada indivisiblemente se denomina **sección crítica**. La sincronización que se precisa para proteger una sección crítica se conoce como **exclusión mutua**. La atomicidad, aunque ausente de la operación de asignación, se supone que está presente en el nivel de la memoria. Por tanto, si un proceso está ejecutando X :=5 simultáneamente con otra ejecución de X :=6, el resultado será o 5 o 6 (no otro valor). Si esto no fuera cierto, sería difícil razonar sobre los programas concurrentes o implementar niveles de atomicidad más altos, como sincronización de exclusión mutua. Sin embargo, está claro que si dos procesos están actualizando un objeto estructurado, esta atomicidad solo se aplicará al nivel del elemento palabra.

El problema de la exclusión mutua fue descrito inicialmente por Dijkstra (1965). Está en el núcleo de la mayoría de las sincronizaciones de procesos concurrentes, y tiene un gran interés teórico y práctico. La exlusión mutua no es la única sincronización importante; evidentemente, si dos procesos no comparten variables no se necesita exclusión mutua. La sincronización condicionada, o de condición, es otro requisito significativo, y es necesaria cuando un proceso desea realizar una operación que sólo puede ser realizada adecuadamente, o de forma segura, si otro proceso ha realizado alguna acción o está en algún estado definido.

Un ejemplo de sincronización de condición se produce al usar búferes. Dos procesos intercambiando datos funcionarán mejor si no se comunican directamente, sino mediante un búfer. La ventaja está en el desacoplamiento de los procesos, que permitirá pequeñas fluctuaciones en la velocidad de trabajo de los dos procesos. Un proceso de entrada, por ejemplo, podrá recibir ráfagas de datos que deberá almacenar en el búfer destinado al proceso de usuario correspondien-

te. Los búferes son habituales en la programación concurrente para enlazar dos procesos, que forman un sistema **productor-consumidor**.

Si se implementa un búfer finito (limitado), se precisa de dos condiciones de sincronización. En primer lugar, el proceso productor no deberá intentar llenar el búfer si éste está lleno. En segundo lugar, no se permite al proceso consumidor extraer objetos del búfer si está vacío. Además, si es posible insertar y extraer simultáneamente, habrá que garantizar que se excluyan mutuamente ciertas operaciones, como por ejemplo que dos productores utilicen a la vez el apuntador «hueco libre siguiente» del búfer, corrompiéndolo.

La implementación de cualquier forma de sincronización, suele acarrear la detención de algún proceso hasta que pueda continuar. En la Sección 8.2, se programarán la exclusión mutua y la sincronización de condición (en un lenguaje tipo Pascal con declaración explícita de procesos) utilizando bucles de *espera ocupada* e **indicadores**. De su análisis resultará clara la necesidad de incluir nuevas primitivas que faciliten la codificación de algoritmos que requieren sincronización.

8.2 Espera ocupada

Una forma de implementar la sincronización es comprobar las variables compartidas que actúan como indicadores en un conjunto de procesos. Esta aproximación sirve razonablemente bien para implementar sincronización de condición, pero no hay un método simple para la exclusión mutua. Para indicar una condición, un proceso activa el valor de un indicador; para esperar por esta condición, otro proceso comprueba este indicador, y continúa sólo cuando se lee el valor apropiado:

```
process P1; (* proceso esperando *)
  ...
  while indicador = abajo do
    null
  end;
  ...
end P1;

process P2; (* proceso indicando *)
  ...
  indicador := arriba;
  ...
end P2;
```

Si la condición no es aún correcta (esto es, si el indicador está aún abajo), `P1` no tiene elección, y deberá continuar en el bucle para volver a comprobar el indicador. Esto se conoce como **espera ocupada**, y también como **giro** –y a los indicadores como **cerrojos de giro** (spin locks)–.

Los algoritmos de espera ocupada son en general ineficientes, y hacen que los procesos consuman ciclos de proceso en un trabajo inútil. Es más, en sistemas multiprocesador, pueden dar lugar a un exceso de tráfico en el bus de la memoria o en la red (si el sistema es distribuido). Además, no es posible imponer fácilmente disciplinas de colas si hay más de un proceso esperando por una condición (es decir, comprobando el valor de un indicador).

Si los algoritmos usados son más complejos, la exclusión mutua se vuelve también más dificultosa. Considere dos procesos (P1 y P2) con secciones críticas mutuas. Para proteger el acceso a ellas, se puede pensar en un protocolo que es ejecutado por cada proceso antes de entrar en la sección crítica, y que va seguido de otro protocolo de salida. El aspecto que tendría cada proceso viene a ser el siguiente:

```
process P;
  loop
    protocolo de entrada
      sección crítica
    protocolo de entrada
    sección no crítica
  end
end P;
```

Antes de dar una solución que sirva para la exclusión mutua, se discutirán tres aproximaciones inexactas. La primera emplea una solución con dos indicadores que es (casi) una extensión lógica del algoritmo de sincronización de condición de espera ocupada:

```
process P1;
  loop
    indicador1 := arriba;      (* anuncia el intento de entrar *)
    while indicador2 = arriba do
      null          (* espera ocupada si el otro proceso está dentro *)
    end;            (* de su sección crítica *)
    <sección crítica>
    indicador1 := abajo;       (* fin del protocolo *)
    <sección no crítica>
  end
end P1;

process P2;
  loop
    indicador2 := arriba;
    while indicador1 = arriba do
      null
    end;
    <sección crítica>
```

```
    indicador2 := abajo;
    <sección no crítica>
  end
end P2;
```

Ambos procesos anuncian su intención de entrar en sus secciones críticas y comprueban si el otro proceso está dentro de su sección crítica. Pero, esta «solución» adolece de un problema no despreciable. Considere un entrelazado con la siguiente traza:

```
P1 activa su indicador (indicador1 ahora arriba)
P2 activa su indicador (indicador2 ahora arriba)
P2 comprueba indicador1 (está arriba, por tanto P2 se mantiene en su bucle)
P2 entra en su espera ocupada
P1 comprueba indicador2 (está arriba, por tanto P1 se mantiene en su bucle)
P1 entra en su espera ocupada
```

El resultado es que ambos procesos permanecen en sus esperas ocupadas. Ninguno podrá salir, porque el otro no puede salir. Este fenomeno se conoce como **interbloqueo activo** (livelock), y es una severa condición de error.

El problema con el algoritmo anterior nace de que cada proceso anuncia su intención de entrar en su sección crítica antes de comprobar si es aceptable anunciarlo. En esta otra aproximación se invierte el orden de estas dos acciones:

```
process P1;
  loop
    while indicador2 = arriba do
      null          (* espera ocupada si el otro proceso está dentro *)
    end;            (* su sección critica *)
    indicador1 := arriba;   (* anuncia su intento de entrar *)
    <sección crítica>
    indicador1 := abajo;    (* protocolo de salida *)
    <sección no crítica>
  end
end P1;

process P2;
  loop
    while indicador1 = arriba do
      null
    end;
    indicador2 := arriba;
    <sección crítica>
    indicador2 := abajo;
```

```
    <sección no crítica>
  end
end P2;
```

Ahora se puede producir un entrelazado, que de hecho falla al proporcionar la exclusión mutua.

```
P1 y P2 están en su sección no crítica ( indicador1 = indicador2 = abajo)
P1 comprueba flag2 (está abajo)
P2 comprueba flag1 (está abajo)
P2 activa su indicador (indicador2 ahora arriba)
P2 entra en su sección crítica
P1 activa su indicador (indicador1 ahora arriba)
P1 entra en su sección crítica
(P1 y P2 están ambos en sus secciones críticas).
```

El problema de estas dos estructuras es que la activación del indicador propio y la comprobación de los otros procesos no es una acción indivisible. Así, podría pensarse que el camino correcto es usar sólo un indicador que indique qué próximo proceso debería entrar en su sección crítica. Como este indicador decide de quién es el turno de entrar, se llamará `turno`.

```
process P1;
  loop
    while turno = 2 do
      null
    end
    <sección crítica>
    turno:= 2
    <sección no crítica>
  end
end P1;

process P2;
  loop
    while turno = 1 do
      null
    end
    <sección crítica>
    turno:=1;
    <sección no crítica>
  end
end P2;
```

Con esta estructura, la variable `turno` deberá valer o `1` o `2`. Si vale `1`, entonces `P1` no puede ser retrasado indefinidamente, y `P2` no puede entrar en su sección crítica. Sin embargo, `turno` no

podrá tomar el valor 2 mientras P1 esté en su sección crítica, dado que la única forma de asignarle 2 es en el protocolo de salida de P1. De una forma simétrica, si turno tiene el valor 2, implica que se garantiza la exclusión mutua, y no es posible el interbloqueo activo si ambos procesos están ejecutando sus respectivos bucles.

Sin embargo, este ultimo punto es crucial: si P1 falla en su sección no crítica, turno tendrá eventualmente el valor 1, y permanecerá con ese valor (es decir, P2 tendrá prohibido entrar en su sección crítica aunque no se este ejecutando P1). Incluso aunque se ejecute normalmente, al usar una sola variable de turno, los procesos funcionarán con el mismo tiempo de ciclo. No es posible que, por ejemplo, P1 entre tres veces en su sección crítica por una de P2. Y esta restricción es inaceptable para procesos autónomos.

Por último, se presenta un algoritmo que evita el estrecho acoplamiento anterior, pero que da exclusión mutua y ausencia de interbloqueo activo. Fue presentado inicialmente por Peterson (1981). Ben-Ari (1982) discute otro algoritmo famoso, el de Dekker. Ambas aproximaciones tienen dos indicadores (indicador1 e indicador2), que son tratados por el proceso que las «posee», y una variable turno que sólo se utiliza en caso de contienda al entrar en las secciones críticas:

```
process P1;
  loop
     indicador1:= arriba;        (* anuncia su intento de entrar *)
     turno:= 2;                  (* da prioridad al otro proceso *)
     while indicador2 = arriba and turno = 2 do
        null
     end;
     <sección crítica>
     indicador1:= abajo;
     <sección no crítica>
  end
end P1;

process P2;
  loop
     indicador2:= arriba;        (* anuncia su intento de entrar *)
     turno:= 1;                  (* da prioridad al otro proceso *)
     while indicador1 = arriba and turno = 1 do
        null
     end;
     <sección crítica>
     indicador2:= abajo;
     <sección no crítica>
  end
end P2;
```

Si no hay más que un proceso queriendo entrar en su sección crítica, entonces el indicador del otro proceso estará abajo y entrará inmediatamente. Pero si se han establecido ambos indicadores, se tendrá en cuenta el valor de `turno`. Suponga que tiene un valor inicial de 1; esto supone que habrá cuatro posibles entrelazamientos, en función del orden en que cada proceso asigne un valor a `turno` y después compruebe su valor en la sentencia `while`:

```
Primera posibilidad: P1 primero, después P2
P1 coloca turno a 2
P1 comprueba turno y entra en su bucle ocupado
P2 coloca turno a 1 (turno se mantendrá con ese valor)
P2 comprueba turno y entra en su bucle ocupado
P1 realiza el bucle, vuelve a comprobar turno y entra en su sección crítica

Segunda posibilidad: P2 primero, después P1
P2 coloca turno a 1
P1 comprueba turno y entra en su bucle ocupado
P1 coloca turno a 2 (turno se mantendrá con ese valor)
P1 comprueba turno y entra en su bucle ocupado
P2 realiza el bucle, vuelve a comprobar turno y entra en su sección crítica

Tercera posibilidad: P1 y P2 entrelazados
P1 coloca turno a 2
P2 coloca turno a 1 (turno se mantendrá con ese valor)
P2 entra en su bucle ocupado
P1 entra en su sección crítica

Cuarta posibilidad: P2 y P1 entrelazados
P2 coloca turno a 1
P1 coloca turno a 2 (turno se mantendrá con ese valor)
P1 entra en su bucle ocupado
P2 entra en su sección crítica
```

Las cuatro posibilidades conducen a un proceso a su sección crítica, y al otro a un bucle de espera.

En general, aunque un sencillo entrelazado puede ilustrar que un sistema falla al satisfacer su especificación, no sirve para mostrar fácilmente que todos los posibles entrelazados están conformes con la especificación. Para esto último, es preciso usar demostraciones formales y comprobación de modelos.

Curiosamente, el algoritmo anterior es equitativo en caso de contienda en el acceso a las secciones críticas, y, por ejemplo, si `P1` tuvo éxito (según el primer o tercer entrelazado), se obliga a que `P2` entre a continuación. Cuando `P1` sale de su sección crítica, baja su `indicador1`. Esto podría permitir que `P2` entrara en su sección crítica, pero incluso si no lo hace (porque no se

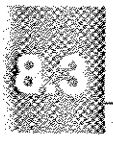
estuviera ejecuntando en ese instante), entonces P1 podría continuar, entrar, y dejar su sección no crítica, levantar indicador1, colocar turno a 2, y pasar a un bucle de espera. Permanecería allí hasta que P2 entrara, dejara su sección crítica y bajara indicador2 según su protocolo de salida.

En términos de fiabilidad, el fallo de un proceso dentro de su sección no crítica no afectará a los demás procesos. No ocurre así cuando fallan los protocolos o la sección crítica. Aquí, la finalización prematura de un proceso supondría problemas de interbloqueo activo para el resto del programa.

Se ha discutido extensamente este problema para mostrar la dificultad de implementar la sincronización entre procesos usando sólo variables compartidas y ninguna otra primitiva adicional distinta de las encontradas en los lenguajes secuenciales. Estas dificultades pueden resumirse en lo siguiente:

- Los protocolos de espera ocupada son difíciles de diseñar y comprender, y es complicado probar su corrección. (El lector podría querer considerar la generalización del algoritmo de Peterson para *n* procesos.)
- Los programas de prueba pueden ignorar entrelazamientos raros que rompen la exclusión mutua o llevan a un interbloqueo activo.
- Los bucles de espera ocupada son ineficientes.
- Una tarea no fiable (engañosa) que utilice falsamente las variables compartidas, corromperá el sistema completo.

Ningún lenguaje de programación concurrente se basa completamente en esperas ocupadas y variables compartidas; hay otros métodos y primitivas. Para sistemas con variables compartidas, los semáforos y monitores que se describen en las Secciones 8.4 y 8.6 son las construciones más usuales.



8.3 Suspender y reanudar

Uno de los problemas de los bucles de espera ocupada es que desperdician tiempo valioso de procesador. Un método alternativo es suspender (es decir, eliminar del conjunto de procesos ejecutables) al proceso solicitante si la condición por la que espera no es cierta. Considere, por ejemplo, una sencilla sincronización de condición con un indicador. Un proceso activa el indicador, y otro espera hasta que el indicador está activado y lo desactiva a continuación. Una forma sencilla de suspender (suspend) y reanudar (resume) podría ser la siguiente:

```
process P1;  (* proceso que espera *)
  ...
  if indicador = abajo do
    suspend;
```

```
  end;
  indicador := abajo;
  ...
end P1;
process P2;  (* signalling process *)
  ...
  indicador := arriba;
  resume P1; (* no tiene efecto si P1 no ha sido suspendido *)
  ...
end P2;
```

Esta aproximación estaba soportada por una de las primeras versiones de la clase `Thread` de Java:

```
public final void suspend();
                    // lanza SecurityException;
public final void resume();
                  // lanza SecurityException;
```

El ejemplo anterior se puede representar en Java así:

```
boolean indicador;
final boolean arriba = true;
final boolean abajo = false;

class PrimerHilo extends Thread {

  public void run() {
     ...
     if(indicador == abajo) {
        suspend();
     };
     indicador = abajo;
     ...
  }
};

class SegundoHilo extends Thread { // H2

  PrimerHilo H1;

  public SegundoHilo(PrimerHilo H) {
     super();
```

```
        H1 =H;
    }

    public void run() {
        ...
        indicador = arriba;
        H1.resume();
        ...
    }
}
```

Desgraciadamente, esta aproximación padece lo que se conoce como una **condición de carrera**. El hilo `H1` podría comprobar el `indicador`, y el soporte subyacente de ejecución (o sistema operativo) decidir desalojarlo y ejecutar `H2`. `H2` activa el `indicador` y `resume H1`. `H1` no está suspendido, como es obvio, por lo que `resume` no tiene efecto. Ahora, cuando `H1` vuelva a ejecutarse, piensa que `indicador` está abajo y, por tanto, se suspende el mismo.

La razón de este problema, que estaba presente en los ejemplos dados en la sección previa, es que el `indicador` es un recurso compartido que está siendo comprobado, y una acción que va a ser tomada que depende de su estatus (el proceso se está suspendiendo él mismo). Está comprobación y suspensión no es una acción atómica, y por tanto puede ser interferenciada por otros procesos. Ésta es la causa de que la versión más reciente de Java dé por obsoletos estos métodos.

Hay varias soluciones bien conocidas para resolver este problema de condición de carrera, y todas ellas ofrecen cierta forma de operación de **suspensión en dos etapas**. Esencialmente, `P1` debe anunciar que planea suspenderse próximamente. Cualquier operación de reanuación que encuentre con que `P1` no está suspendido verá aplazado su efecto, y cuando `P1` se suspenda, será reiniciado inmediatamente; es decir, anulará la suspensión.

Aunque la funcionalidad suspender y reanudar es de bajo nivel, propensa a una utilización errónea, es un mecanismo eficiente que puede usarse para construir primitivas de sincronización de nivel superior. Por ello, Ada proporciona, como parte de su Anexo para Tiempo Real, una versión segura de este mecanismo. Está basado en el concepto de un objeto `suspensión`, que puede mantener el valor `True` o el valor `False`. El Programa 8.1 proporciona la especificación del paquete.

Los cuatro subprogramas definidos por el paquete son atómicos uno con respecto a otro. En la vuelta del procedimiento `Suspend_Until_True`, el objeto suspensión referenciado es colocado a `False`.

El sencillo problema de sincronización de condición anterior puede así resolverse fácilmente.

```
with Ada.Synchronous_Task_Control;
use Ada.Synchronous_Task_Control;

...
Indicador : Suspension_Object;
...
```

Programa 8.1. Control síncrono de tareas.

```
package Ada.Synchronous_Task_Control is
   type Suspension_Object is limited private;
   procedure Set_True(S : in out Suspension_Object);
   procedure Set_False(S : in out Suspension_Object);
   function Current_State(S : Suspension_Object) return Boolean;
   procedure Suspend_Until_True(S: in out Suspension_Object);
     -- genera Program_Error si más de una tarea tratara
     -- de suspender sobre S a la vez
private
   -- no especificado por el lenguaje
end Ada.Synchronous_Task_Control;
```

```
task body P1 is
begin
  ...
  Suspend_Until_True(Indicador);
  ...
end P1;

task body P2 is
begin
  ...
  Set_True(Indicador);
  ...
end P1;
```

Los objetos suspensión se comportan casi de la misma forma que los semáforos binarios, los cuales se discuten en la Sección 8.4.4.

Aunque `suspender` y `reanudar` son útiles primitivas de bajo nivel, ningún sistema operativo o lenguaje depende únicamente de estos mecanismos para la exclusión mutua o la sincronización de condición. Si están presentes, introducen claramente un nuevo estado en el diagrama de transición de estados presentado en el Capítulo 7. El diagrama general de estados extendido para un proceso es el de la Figura 8.1.

8.4 Semáforos

Los semáforos son un mecanismo sencillo para la programación de la exclusión mutua y la sincronización de condición. Fueron diseñados originalmente por Dijkstra (1968a), y aportan los dos beneficios siguientes:

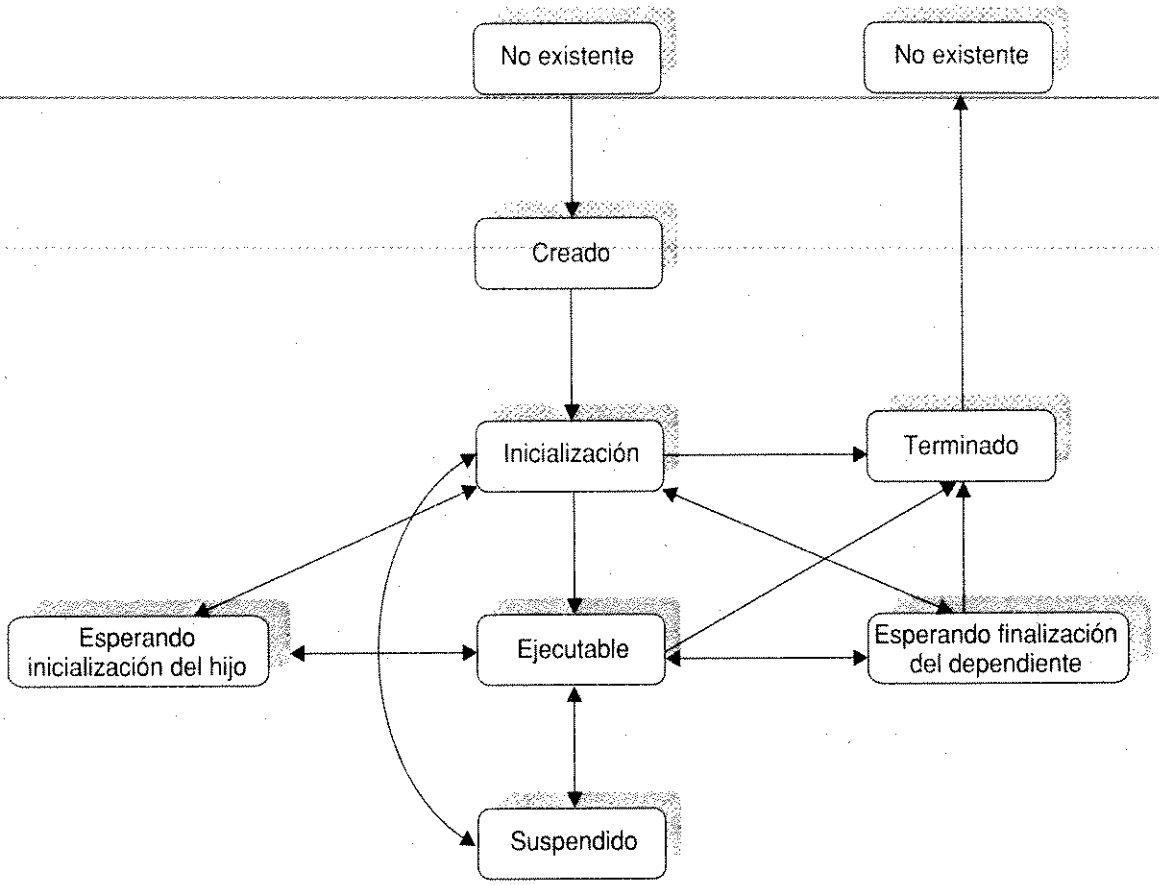


Figura 8.1. Diagrama de estados para un proceso.

(1) Simplifican los protocolos para la sincronización.

(2) Eliminan la necesidad de bucles de espera ocupados.

Un **semáforo** es una variable entera no negativa que, aparte de la inicialización, sólo puede ser manejada por dos procedimientos. Estos procedimientos fueron llamados por Dijkstra *P* y *V*, pero en este libro se denominarán `wait` y `signal`. La semántica de `wait` y `signal` es como sigue:

(1) `wait(S)` Si el valor del semáforo, `S`, es mayor que cero, entonces decrementa su valor en uno; en caso contrario, demora el proceso hasta que `S` sea mayor que cero (y entonces decrementa su valor).

(2) `signal(S)` Incrementa el valor del semáforo, `S`, en uno.

Los semáforos generalizados suelen conocerse como **semáforos de conteo**, ya que sus operaciones incrementan o decrementan un contador entero. La propiedad adicional importante de `wait` y `signal` es que sus acciones son atómicas (indivisibles). Dos procesos, que ejecuten sendas llamadas a `wait` sobre el mismo semáforo no interfieren. Además, un proceso no falla durante la ejecución de una operación de semáforo.

La sincronización de condición y la exclusión mutua pueden programarse fácilmente con semáforos. Primero, considere la sincronización de condición:

```
(* sincronización de condición *)
var sincon : semaforo; (* inicialmente 0 *)
process P1; (* proceso que espera *)
  ...
  wait (sincon);
  ...
end P1;

process P2; (* proceso que ejecuta signal *)
  ...
  signal (sincon);
  ...
end P2;
```

Cuando P1 ejecuta wait en un semáforo a 0, será demorado hasta que P2 ejecute signal. Esto pondrá sincon a 1, con lo que terminará wait; P1 continuará, y sincon se decrementa a 0. Hay que añadir que si P2 ejecuta signal primero, el semáforo se pone a 1, por lo que P1 no se demora al ejecutar wait.

La exclusión mutua es igualmente sencilla.

```
(* exclusión mutua *)
var mutex : semaforo; (* inicialmente a 1 *)
process P1;
  loop
    wait (mutex);
      <sección crítica>
    signal (mutex);
    <sección no crítica>
  end
end P1;

process P2;
  loop
    wait (mutex);
      <sección crítica>
    signal (mutex);
    <non-critical section>
  end
end P2;
```

P1 y P2 se disputan la entrada en la sección crítica (están en contienda) si ejecutan sus sentencias wait simultáneamente. Sin embargo, al ser wait atómica, uno de ellos completará la ejecución de su sentencia antes de que comience el otro. Un proceso ejecutará un wait (mutex)

con `mutex=1`, lo que le permitirá dirigirse a su sección crítica y poner `mutex` a `0`; el otro proceso ejecutará `wait (mutex)` con `mutex=0`, y será demorado. Una vez que el primer proceso salga de su sección crítica, efectuará `signal (mutex)`, lo que producirá que el semáforo se coloque a `1` de nuevo, y permitirá entrar en su sección crítica al segundo proceso (y colocar `mutex` a `0` de nuevo).

Al delimitar una sección de código con wait/signal, el valor inicial del semáforo restringirá la cantidad máxima de objetos de código que se ejecutan concurrentemente. Si el valor inicial es `0`, no entrará ningún proceso; si es `1`, podrá entrar sólo un proceso (es decir, exclusión mutua); para valores mayores que uno, se permitirá el número dado de ejecuciones concurrentes del código.

8.4.1 Procesos suspendidos

En la definición de `wait` está claro que si el semáforo es cero, el proceso invocador se demora. Ya se ha presentado y criticado un método de demora (espera ocupada). En la Sección 8.3 se introdujo un mecanismo más eficiente: la suspensión del proceso. De hecho, todas las primitivas de sincronización tratan el retraso mediante alguna forma de suspensión, y a la postre, el proceso se elimina del conjunto de procesos en estado ejecutable.

Cuando un proceso efectúa `wait` sobre un semáforo a cero, se invoca al RTSS (sistema de soporte de ejecución; *run-time support system*), el proceso se elimina del procesador , y se coloca cierta cola de procesos suspendidos (la cola de procesos suspendidos por un semáforo concreto). El RTSS debe seleccionar, entonces, otro proceso para ejecutar. Finalmente, si el programa es correcto, otros procesos ejecutarán `signal` sobre dicho semáforo. Como resultado, el RTSS eligirá uno de los procesos suspendidos que esperan por un `signal` sobre el semáforo y lo hará ejecutable de nuevo.

Con todas estas consideraciones, se puede dar una definición levemente diferente de `wait` y `signal`, más próxima a lo que debiera hacer una implementación:

```
wait(S) :-
if S > 0 then
  S:= S-1
else
  número_suspendido:= número_suspendido + 1
  se suspende el proceso invocador
signal(S) :
-if número_suspendido > 0 then
  número_suspendido:= número_suspendido - 1
  vuelve nuevamente ejecutable un proceso suspendido
else
  S:= S+1
```

Con esta definición, se evita el incrementar el semáforo si va seguido inmediatamente por su decremento.

Conviene indicar que el algoritmo anterior no define el orden en el que se liberan los procesos de su estado suspendido. Normalmente, se liberan en orden FIFO, aunque cabría decir que con un auténtico lenguaje concurrente, el programador debería asumir un orden no determinista (véase la Sección 9.4.3). Sin embargo, para un lenguaje de programación de tiempo real, la prioridad de los procesos juega un papel importante (véase el Capítulo 13).

8.4.2 Implementación

El anterior algoritmo de implementación de semáforos es bastante sencillo, aunque implica el soporte de un mecanismo de cola. Donde puede haber problemas es con el requisito de indivisibilidad de la ejecución de las operaciones `wait` y `signal`. Indivisibilidad significa que una vez que un proceso ha comenzado uno de estos procedimientos, no podrá ser detenido hasta completar la operación. Esto es fácilmente realizable con la ayuda del RTSS; se programa el planificador de modo que no desaloje a un proceso mientras esté ejecutando un `wait` o un `signal`. Se trataría de operaciones **no desalojables**.

Lamentablemente, el RTSS no siempre tiene el control total de los eventos de planificación. Aunque todas las acciones internas están bajo su influencia, las acciones externas suceden asíncronamente, y podrían obstaculizar la atomicidad de las operaciones del semáforo. Para impedir esto, el RTSS inhibirá las interrupciones durante la ejecución de la secuencia indivisible de sentencias. Así, ningún evento externo podrá interferir.

Esta inutilización de las interrupciones es adecuada para un sistema monoprocesador, pero no para uno multiprocesador. Con un sistema de memoria compartida, puede haber dos procesos paralelos ejecutando `wait` o `signal` en el mismo semáforo, hecho que no puede impedir el RTSS. En estas circustancias, es preciso bloquear el acceso a dichas operaciones mediante un «cerrojo» (lock). Se utilizan dos mecanismos de ese tipo.

En algunos procesadores, se dispone de una instrucción `test and set` (comprueba y establece). Esto permite que un proceso acceda a un bit de la siguiente forma:

(1) Si el bit es cero, se coloca a 1 y devuelve cero.

(2) Si el bit es uno, devuelve uno.

Estas acciones son de por sí indivisibles. Dos procesos paralelos intentando efectuar `wait` (por ejemplo), realizarán una operación `test and set` sobre el mismo bit de cerrojo (que inicialmente está a cero). Uno de ellos lo conseguirá, y colocará el bit a uno; el otro obtendrá un uno y, por tanto, tendrá que volver a comprobar el cerrojo posteriormente. Cuando el primer proceso complete su operación `wait`, asignará el bit a cero (es decir desbloqueará el semáforo), y el otro proceso podrá proseguir para ejecutar su operación `wait`.

Si no se dispone de instrucción `test and set`, se puede obtener un efecto similar mediante una instrucción `intercambio` (exchange). De nuevo, el cerrojo está asociado con un bit que inicialmente está a cero. Un proceso que desee ejecutar una operación de semáforo intercambiará un uno con el bit del cerrojo. Si el resultado del cerrojo es un cero, podrá continuar; si resulta un uno, es que algún proceso está activo con el semáforo, en cuyo caso deberá volver a realizar el intercambio.

Como indicó en la Sección 8.1, una primitiva software, como es un semáforo, no puede extraer la exclusión mutua de la nada. Es preciso que las posiciones de memoria exiban esencialmente la exclusión mutua con el fin de poder construir estructuras de alto nivel. De modo similar, aunque se eliminen del dominio del programador los bucles de espera ocupada mediente el uso de semáforos, puede ser necesario utilizar esperas ocupadas (como antes) para implementar las operaciones de `wait` y `signal`. *Hay que decir, sin embargo, que el uso que se acaba de hacer de las esperas ocupadas es muy efímero (el tiempo que lleva ejecutar una operación wait o signal), no siendo el caso de su utilización explícita en un programa para retrasar el acceso a secciones críticas, que podría implicar muchos segundos de bucle.*

8.4.3 Provisión de vivacidad

En la sección 8.2, se presentó la condición de error de interbloqueo activo. Desafortunada pero inevitablemente, el uso de primitivas de sincronización presenta otras situaciones de error. El **interbloqueo** (deadlock) es la más seria de dichas situaciones, e implica que un conjunto de procesos estén en un estado desde el que es imposible continuar. Esto es similar al interbloqueo activo, aunque los procesos estén suspendidos. Para ilustrar esta situación, considere dos procesos, `P1` y `P2`, intentando obtener el acceso a dos recursos no concurrentes (esto es, recursos que sólo pueden ser accedidos por un proceso cada vez) protegidos por dos semáforos, `S1` y `S2`. Si ambos procesos acceden al recurso en el mismo orden, entonces no se plantean problemas.

```
  P1                    P2
wait (S1);            wait (S1);
  wait (S2);            wait (S2);
  .                     .
  .                     .
  .                     .
  signal (S2);          signal (S2);
signal (S1);          signal (S1);
```

El primer proceso que ejecute `wait` en `S1` con éxito efectuará también con éxito `wait` en `S2`, y finalmente `signal` con los dos semáforos, lo que permitirá continuar al otro proceso. Sin embargo, el problema aparece si uno de ellos desea utilizar los recursos en el orden inverso. Por ejemplo:

```
  P1                    P2
wait (S1);            wait (S2);
  wait (S2);            wait (S1);
  .                     .
  .                     .
  .                     .
  signal (S2);          signal (S1);
signal (S1);          signal (S2);
```

En este caso, cierto entrelazamiento podría permitir que `P1` y `P2` efectuaran con éxito sus respectivos `wait` sobre `S1` y `S2` iniciales. Pero entonces, inevitablemente, ambos procesos serían suspendidos a la espera del otro semáforo, que ahora es cero.

Por la naturaleza interdependiente de la programación concurrente, si cierto subconjunto de procesos se interbloquea, más tarde o más temprano se interbloquearán todos los demás.

Las pruebas de software casi nunca eliminan más que los interbloqueos más claros; éstos podrán ser infrecuentes, pero sus resultados son devastadores. Este error no se elimina mediante semáforos, y es posible en cualquier lenguaje de programación concurrente. El diseño de lenguajes que prohiban la programación de interbloqueos es una meta deseable, pero inalcanzable. En los Capítulos 11 y 13 se tratarán las cuestiones relacionadas con la eliminación, detección y recuperación de interbloqueos.

Los **aplazamientos indefinidos** (conocidos como *lockout* o **inanición**) suponen una condición de error menos severa, que consiste en que un proceso que desea acceder a cierto recurso mediante una sección crítica, permanezca siempre en un estado de espera, porque siempre hay otros procesos que ganan el acceso al recurso antes que él. Con un sistema de semáforos, un proceso podría permanecer suspendido indefinidamente (es decir, encolado en el semáforo) según sea la forma en que el RTSS elige los procesos de esta cola cuando llega una señal. Incluso si el retraso no es realmente indefinido, sino indeterminado, podría engendrar errores en un sistema de tiempo real.

Si un proceso está libre de interbloqueo activo, interbloqueos y aplazamientos indefinidos, entonces se dice que posee **vivacidad**. Informalmente, la propiedad de vivacidad implica que si un proceso desea realizar alguna acción, entonces, eventualmente, podrá hacerlo. Concretamente, si un proceso solicita el acceso a una sección crítica, lo conseguirá en un tiempo finito.

8.4.4 Semáforos binarios y de cantidad

La definición general de semáforo conlleva un entero no negativo; por ello, su valor actual puede alcanzar cualquier número positivo soportado, aunque en todos los ejemplos dados anteriormente en este capítulo (para sincronización de condición y exclusión mutua) sólo se han utilizado los valores 0 y 1. A un semáforo que sólo toma estos dos valores se le conoce como **semáforo binario**, y su implementación es sencilla. Efectuar un `signal` sobre un semáforo binario que tiene el valor 1 no tiene efecto: el semáforo permanece con el valor 1. Si se precisa una forma más general, se puede abordar la construcción de un semáforo generalizado con dos semáforos binarios y un entero.

Otra variación de la definición normal de un semáforo es el **semáforo de cantidad**. Con esta estructura, la cantidad a ser decrementada por `wait` (e incrementada por `signal`) no está fijada en 1, sino que se da como parámetro a los procedimientos:

```
wait (S, i) :- if S >= i then
                    S:= S-i
               else
```

```
                    demora
                    S:= S-i
signal (S, i) : S:= S+i
```

Más adelante, en la Sección 8.6.2, se dará un ejemplo de utilización de un semáforo de cantidad.

8.4.5 Ejemplo de programas con semáforos en Ada

Algol-68 fue el primer lenguaje en incluir semáforos. Proporcionaba un tipo, `sema`, que era manipulado por los operadores `up` y `down`. Para ilustrar algunos programas sencillos que utilizan semáforos, se utilizará un tipo abstracto de datos para semáforos, en Ada.

```
package Paquete_Semaforo is
  type Semaforo(Inicial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaforo);
  procedure Signal (S : in out Semaforo);
private
  type Semaforo is ...
end Paquete_Semaforo;
```

Ada no soporta directamente semáforos, pero los procedimientos `wait` y `signal` pueden construirse a partir de las primitivas de sincronización de Ada; éstas no se han discutido todavía, por lo que la definición total del tipo semáforo y del cuerpo del paquete no se darán aquí (véase la Sección 8.7). Lo fundamental de los tipos abstractos de datos es, sin embargo, que pueden ser utilizados sin conocer su implementación.

El primer ejemplo es un sistema productor consumidor que utiliza un búfer limitado para pasar enteros entre las dos tareas:

```
procedure Main is
  package Bufer is
    procedure Agrega (I : Integer);
    procedure Toma (I : out Integer);
  end Bufer;
  task Productor;
  task Consumidor;

  package body Bufer is separate; -- véase más adelante
  use Bufer;

  task body Productor is
    Item : Integer;
  begin
```

```
    loop
      -- produce item
      Agrega (Item);
    end loop;
  end Productor;

  task body Consumidor is
    Item : Integer;
  begin
    loop
      Toma (Item);
      -- consume item
    end loop;
  end Consumidor;
begin
  null;
end Main;
```

El propio búfer debe protegerse frente al acceso concurrente: la agregación sobre un búfer lleno y la extracción sobre uno vacío. Esto se realiza mediante tres semáforos:

```
with Paquete_Semaforo; use Paquete_Semaforo;
separate (Main)
package body Bufer is
  Talla : constant Natural := 32;
  type Rango_Bufer is mod Size;
  Buf : array (Rango_Bufer) of Integer;
  Tope, Base : Rango_Bufer := 0;

  Mutex : Semaforo; -- por defecto es 1
  Item_Disponible : Semaforo(0);
  Espacio_Disponible : Semaforo(Inicial => Talla);

  procedure Agrega (I : Integer) is
  begin
    Wait(Espacio_Disponible);
    Wait(Mutex);
      Buf(Tope) := I;
      Tope := Tope+1;
    Signal(Mutex);
    Signal(Item_Disponible);
  end Agrega;
```

```
  procedure Toma (I : out Integer) is
  begin
     Wait(Item_Disponible);
     Wait(Mutex);
        I := Buf(Base);
        Base := Base+1;
     Signal(Mutex);
     Signal(Espacio_Disponible);
  end Toma;
end Bufer;
```

Los valores iniciales de los tres semáforos son diferentes. `Mutex` es un semáforo de exclusión mutua (mutual exclusion) ordinario, y su valor inicial es 1; `Item_Disponible` protege frente a tomar de un búfer vacío, y tiene el valor inicial 0; y `Espacio_Disponible` (inicialmente `Talla`) se utiliza para prevenir operaciones `Agrega` en un búfer lleno.

Cuando comienza el programa, cualquier tarea Consumidor que llame a `Toma` será suspendida en `Wait(Item_Disponible)`; sólo después de que una tarea Productor haya llamado a `Agrega` –y ejecutado, por tanto, `Signal(Item_Disponible)`–, continuará la tarea Consumidor.

8.4.6 Programación de semáforos utilizando C y POSIX

Aunque pocos lenguajes modernos de programación soportan semáforos directamente, muchos sistemas operativos sí lo hacen. POSIX, por ejemplo, proporciona semáforos con contador que permiten a los procesos que se ejecutan en espacios de direccionamiento distintos (o hilos con el mismo espacio de direccionamiento) sincronizarse y comunicarse utilizando memoria compartida. Hay que indicar, sin embargo, que para sincronizar y comunicar en el mismo espacio de almacenamiento es más eficiente utilizar mutexes y variables de condición (véase la Sección 8.6.3). El Programa 8.2 muestra la interfaz POSIX en C para semáforos (también se proporcionan funciones para llamar a un semáforo mediante una cadena de caracteres, pero han sido omitidas aquí). En POSIX, las operaciones estándar de semáforos *initialize*, *wait* y *signal*, se llaman `sem_init`, `sem_wait` y `sem_post`. También se proporcionan un wait no bloqueante (`sem_trywait`) y una versión temporizada (`sem_timedwait`), así como una rutina para determinar el valor actual de un semáforo (`sem-getvalue`).

Considere el ejemplo de un controlador de recursos, que aparece de muchas formas en los programas de tiempo real. Para simplicar, el ejemplo utilizará hilos en vez de procesos. Se proporcionan dos funciones: `asigna` y `desasigna`; cada una toma un parámetro que indica un nivel de prioridad asociado con la solicitud. Se supone que el hilo invocador libera el recurso a la misma prioridad con la que hizo la reserva solicitada. Para facilitar la presentación, el ejemplo no considera cómo se transfiere el recurso. Sin embargo, la solución no le protege contra condiciones de carrera (véase el Ejercicio 8.27).

Programa 8.2. Una interfaz C POSIX para semáforos.

```
#include <time.h> typedef ... sem_t;

int sem_init(sem_t *sem_ubicacion, int pshared, unsigned int valor);
   /* inicializa el semáforo en la posición sem_location con valor */
   /* si pshared es 1, el semáforo puede usarse entre procesos */
   /* o hilos */
   /* si pshared es 0, el semáforo únicamente podrá usarse entre hilos */
   /* o en el mismo proceso */

int sem_destroy(sem_t *sem_ubicacion);
   /* destruye un semáforo sin nombre en la posición sem_ubicacion */

int sem_wait(sem_t *sem_ubicacion);
   /* una operación wait estándar sobre el semáforo */

int sem_trywait(sem_t *sem_ubicacion);
   /* intenta decrementar el semáforo */
   /* retorna -1 si la llamada bloqueara el proceso invocador */

int sem_timedwait(sem_t *sem, const struct timespec *abstime);
   /* retorna -1 si el semáforo no pudiera bloquearse */
   /* durante un tiempo absoluto */

int sem_post(sem_t *sem_ubicacion);
   /* una operación signal estándar sobre el semáforo */

int sem_getvalue(sem_t *sem_ubicacion, int *valor);
   /* obtiene el valor actual del semáforo sobre una posición */
   /* apuntada por un valor; un número negativo indica el número */
   /* de hilos en espera */

   /* Todas las funciones anteriores devuelven 0 con éxito, si no -1 */
   /* Además, cuando se devuelve una condición de error */
   /* una variable compartida, errno, contiene la razón del error */
```

```
#include <semaphore.h>

typedef enum {alto, medio, bajo} prioridad_t;
typedef enum {false, true} booleano;

sem_t mutex;       /* usado para acceso mutuamente
                      excluyente a esperando y ocupado */
```

```
sem_t cond[3];    /* usado para sincronización de condición */
int esperando;    /* recuento del número de hilos que esperan
                     a cierto nivel de prioridad */
boolean ocupado;  /* indica si el recurso está en uso*/

void asigna(prioridad_t P)
{
  SEM_WAIT(&mutex); /* bloquea mutex */
  if(ocupado) {
     SEM_POST(&mutex); /* libera mutex */
     SEM_WAIT(&cond[P]); /* espera al nivel correcto de prioridad */
     /* el recurso ha sido asignado */
  }
  ocupado = true;
  SEM_POST(&mutex); /* libera mutex */
}
```

Se utiliza un único semáforo, `mutex`, para asegurar la exclusión mutua en las peticiones de asignación y desasignación. Se usan tres semáforos de sincronización de condición para la cola de los hilos que esperan a tres niveles de prioridad (`alto`, `medio` y `bajo`). La función `asigna` asigna el recurso si no está en uso (indicado por el indicador `ocupado`).

La función de desasignación simplemente realiza signal en el semáforo para el proceso de prioridad más alta que espera.

```
int desasigna(prioridad_t P)
{

  SEM_WAIT(&mutex); /* bloquea mutex */
  if(ocupado) {
     ocupado = false;
     /* libera el hilo en espera de mayor prioridad */
     SEM_GETVALUE(&cond[alto],&esperando);
     if (esperando < 0) {
        SEM_POST(&cond[alto]);
     }
     else {
        SEM_GETVALUE(&cond[medio],&esperando);
        if (esperando < 0) {
           SEM_POST(&cond[medio]);
        }
        else {
           SEM_GETVALUE(&cond[bajo],&esperando);
           if (esperando < 0) {
```

```
            SEM_POST(&cond[bajo]);
          }
          else SEM_POST(&mutex);
          /* nadie esperando, libera bloqueo */
        }
      }
      /* recurso y bloqueo pasado sobre el */
      /* hilo en espera de mayor prioridad */
      return 0;
    }
    else return -1; /* retorna error */
}
```

Una rutina de inicialización establece el indicador `ocupado` a `false`, y crea los cuatro semáforos utilizados por `asigna` y `desasigna`.

```
void inicializa() {
  prioridad_t i;

  ocupado = false;
  SEM_INIT(&mutex, 0, 1);
  for (i = alto; i<= bajo; i++) {
    SEM_INIT(&cond[i], 0, 0) ;
  };
}
```

Recuerde que, como el enlace de C a POSIX utiliza valores de retorno distintos de cero para indicar que ha ocurrido un error, es necesario encapsular cada llamada POSIX en una sentencia `if`. Esto hace que el código sea más difícil de entender (un enlace de C++ o Ada con POSIX debiera permitir la generación de excepciones en presencia de error). Consecuentemente, como con los otros ejemplos de C utilizados en este libro, `SYS_CALL` se usa para representar una llamada a `sys_call` y una recuperación de error apropiada (véase la Sección 6.1.1). Para `SEM-INIT` esto *puede incluir un reintento.*

Un hilo que desee utilizar el recurso deberá hacer las siguientes llamadas:

```
prioridad_t mi_prioridad;

...
asigna(mi_prioridad); /* espera por el recurso */
  /* utiliza el recurso */
if(desasigna(mi_prioridad) <= 0) {
  /* no puede liberar el recurso, */
  /* promete alguna operación de recuperación */
}
```

8.4.7 Críticas a los semáforos

Aunque el semáforo es una elegante primitiva de sincronización de bajo nivel, un programa de tiempo real construido sólo sobre el uso de semáforos es de nuevo propenso a errores. Basta con que se produzca la omisión o mala ubicación de un semáforo para que el programa completo falle en tiempo de ejecución. No se puede garantizar la exclusión mutua, y puede aparecer el bloqueo indefinido justo cuando el software está tratando con un suceso raro pero crítico. Se precisa una primitiva de sincronización más estructurada.

Lo que proporciona un semáforo es un medio para programar exclusión mutua sobre una sección crítica. Una aproximación más estructurada debiera proporcionar la exclusión mutua directamente. Esto es lo que ofrecen las construcciones discutidas en las Secciones 8.5 a 8.8.

Los ejemplos mostrados en la Sección 8.4.5 muestran cómo se puede construir un tipo abstracto de datos para semáforos en Ada. Sin embargo, ningún lenguaje de programación concurrente de alto nivel confía totalmente en los semáforos. Son importantes históricamente, pero podría decirse que no son adecuados para el dominio de tiempo real.

8.5 Regiones críticas condicionales

Las regiones críticas condicionales (CCR; conditional critical regions) son un intento de superar algunos de los problemas asociados con los semáforos. Una región crítica es una sección de código que está garantizado que será ejecutado en exclusión mutua. Debe compararse con el concepto de sección crítica, la cual debiera ejecutarse bajo exclusión mutua (pero cuando hay error pudiera no serlo). Claramente, la programación de una sección crítica como una región crítica satisface inmediatamente el requisito para exclusión mutua.

Las variables que deben ser protegidas de un uso concurrente son agrupadas juntas en las llamadas regiones, y son etiquetadas como recursos. Está prohibido que los procesos entren en una región en la que hay otro proceso activo. La condición de sincronización se proporciona mediante guardas en las regiones. Cuando un proceso desea entrar en una región crítica, evalúa la guarda (bajo exclusión mutua): si la guarda se evalúa a cierto podrá entrar, pero si es falsa el proceso se demorará. Como con los semáforos, el programador no supone ningún orden de acceso si hay más de un proceso demorado intentando entrar en la misma región crítica (por cualquier razón).

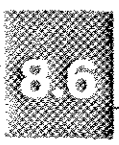
Para ilustrar el uso de las CCR, se proporciona a continuación un esbozo del programa del búfer limitado.

```
program bufer_eg;
  type bufer_t is record
    huecos : array(1..N) of character;
    talla : integer range 0..N;
    tope, base : integer range 1..N;
  end record;
```

```
bufer : bufer_t;

resource buf : bufer;

process productor;
  ...
  loop
    region buf when bufer.talla < N do
      -- poner caracter en el búfer, etc
    end region
    ...
  end loop;
end

process consumidor;
  ...
  loop
    region buf when buffer.talla > 0 do
      -- toma caracter del búfer, etc
    end region
    ...
  end loop;
end
end
```

Un posible problema de prestaciones con CCR es que los procesos deben reevaluar sus guardas cada vez que se abandona una CCR que llama al recurso. Un proceso suspendido debe volver a ejecutable de nuevo con el fin de comprobar la guarda, y, si todavía es falsa, volver al estado suspendido.

Edison (Brinch-Hansen, 1981) dispone de una versión de CCR. Este lenguaje se diseñó para aplicaciones embebidas sobre sistemas multiprocesador. Cada procesador ejecuta un solo proceso, de forma que, si es preciso, puede evaluar repetidamente sus guardas, aunque puede causar un excesivo tráfico en la red.



8.6 Monitores

El problema principal con las regiones condicionales es que pueden estar dispersas a lo largo de todo el programa. Los monitores están pensados para atenuar este problema proporcionando regiones de control más estructuradas. También pueden utilizar una forma de sincronización de condición que es más eficiente de implementar.

Las regiones críticas consideradas se escriben como procedimientos, y están encapsuladas en un único módulo, llamado monitor. Como módulo, se ocultan todas las variables que deben ser accedidas bajo exclusión mutua; adicionalmente, como monitor, todas las llamadas a procedimientos del módulo tienen garantizada su ejecución con exclusión mutua.

Los monitores surgen como un refinamiento de las regiones críticas condicionales; el diseño inicial y el análisis de la estructura fueron emprendidos por Dijkstra (1968b), Brinch-Hansen (1973) y Hoare (1974). Pueden encontrarse en numerosos lenguajes de programación, incluyendo Modula-1, Pascal concurrente y Mesa.

Continuando, a efectos de comparación, con el ejemplo del búfer limitado, un monitor búfer debiera tener la siguiente estructura:

```
monitor bufer;
  export agrega, toma;
  var (* declaración de las variables necesarias *)

  procedure agrega (I : integer);
     ...
  end;

  procedure toma (var I : integer);
     ...
  end;
begin
  (* inicialización de las variables del monitor *)
end
```

Con lenguajes como Modula-2 y Ada, es natural programar el búfer como un módulo distinto (package). Ocurre lo mismo al programarlo como monitor, aunque la diferencia entre un módulo y un monitor está en que, en este último caso, las llamadas concurrentes para agregar o tomar (en el caso anterior) son secuencializadas por definición, haciendo innecesarios los semáforos para la exclusión mutua.

Aunque el monitor proporciona exclusión mutua, existe todavía una necesidad de sincronización de condición en él. En teoría, podrían seguir utilizándose semáforos, pero normalmente se incluye una primitiva de sincronización más sencilla. En los monitores de Hoare (Hoare, 1974), esta primitiva se llama **variable de condición**, y se maneja mediante dos operadores que, por sus similitudes con la estructura de semáforo, serán llamados de nuevo `wait` y `signal`. Cuando un proceso lanza una operación `wait`, se bloquea (suspendido) y se ubica en una cola asociada con esa variable de condición (comparable con un `wait` en un semáforo con valor cero). Sin embargo, hay que indicar que un `wait` sobre una variable de condición *siempre* bloquea, a diferencia de un `wait` en un semáforo. Cuando un proceso bloqueado libera su bloqueo mutuamente exclusivo en el monitor, permitirá entrar a otro proceso. Cuando un proceso ejecuta una operación `signal`, liberará un proceso bloqueado. Si no hay ningún proceso bloqueado en la variable es-

pecificada, entonces `signal` *no tiene efecto*. De nuevo, hay que señalar el contraste con `signal` en un semáforo, que siempre tiene un efecto en el semáforo. Evidentemente, la semántica de `wait` y `signal` para monitores es más similar a la de suspender y reanudar. El ejemplo del búfer limitado tiene este aspecto:

```
monitor bufer;
  export agrega, toma;
  const talla = 32;
  var buf : array[0...talla-1] of integer;
     tope, base : 0..talla-1;
     EspacioDisponible, ItemDisponible : condition;
     NumeroEnBufer : integer;

  procedure agrega (I : integer);
  begin
     if NumeroEnBufer = talla then
        wait(EspacioDisponible);
     buf[tope] := I;
     NumeroEnBufer := NumeroEnBufer+1;
     tope := (tope+1) mod talla;
     signal(ItemDisponible)
  end agrega;

  procedure toma (var I : integer);
  begin
     if NumeroEnBufer = 0 then
        wait(ItemDisponible);
     I := buf[base];
     base := (base+1) mod talla;
     NumeroEnBufer := NumeroEnBufer-1;
     signal(EspacioDisponible)
  end toma;

begin (* inicializacion *)
  NumeroEnBufer := 0;
  tope := 0;
  base := 0
end;
```

Si un proceso invoca, por ejemplo, `toma` cuando no hay nada en el búfer, entonces será suspendido en `ItemDisponible`. Sin embargo, un proceso que añade un item realizará un signal sobre el proceso suspendido cuando esté disponible un item.

Las semántica para `wait` y `signal` dadas anteriormente no son completas; dos o más procesos podrían llegar a estar activos en un monitor. Esto podría ocurrir siguiendo cierta operación `signal` que liberara un proceso bloqueado. Tanto el proceso liberado como el que lo liberó se encontrarán en ejecución dentro del monitor. Para prohibir esta actividad claramente indeseable, debe modificarse la semántica de `signal`. En los lenguajes se utilizan cuatro aproximaciones diferentes:

(1) Se permite un signal sólo como la última acción de un proceso antes de dejar el monitor (éste es el caso del ejemplo anterior del búfer limitado).

(2) Una operación signal tiene el efecto colateral de ejecutar una sentencia return; es decir, el proceso es forzado a dejar el monitor.

(3) Una operación signal que desbloquea otro proceso tiene el efecto de bloquearse a sí misma, y sólo se desbloqueará cuando quede libre el monitor.

(4) Una operación signal que desbloquea otro proceso no bloquea, y el proceso liberado debe competir por el acceso al monitor una vez que acaba el proceso que lanzó el signal.

En el caso (3), propuesto por Hoare en su trabajo original sobre monitores, los procesos que bloqueados por una acción signal se ponen en una «cola de dispuestos», y son elegidos, cuando el monitor es liberado, con preferencia sobre los procesos bloqueados en la entrada. En el caso (4), el proceso liberado es el que se coloca en la «cola de dispuestos».

Debido a la importancia de los monitores, se describirán brevemente dos lenguajes que soportan dicha estructura.

8.6.1 Modula-1

Modula-1 (como se le conoce ahora) es el precursor de Modula-2 y Modula-3, pero tiene un modelo de procesos completamente diferente. Emplea declaración explícita de procesos (no corrutinas) y monitores, que son denominados **módulos de interfaz**. De forma algo confusa, las variables de condición se llaman señales, y son activadas por tres procedimientos.

(1) El procedimiento `wait(s,r)` retrasa el proceso invocador hasta que recibe la señal `s`. Al demorar el proceso, se le proporciona una prioridad (o rango de retraso, `r`), donde `r` debe ser una expresión entera positiva cuyo valor por defecto es 1.

(2) El procedimiento `send(s)` envía la señal `s` al proceso con la prioridad más alta que ha estado esperando por `s`. Si hay varios procesos con la misma prioridad esperando, entonces recibirá la señal el que ha estado esperando más tiempo. El proceso que ejecuta el `send` es suspendido. Si no hay procesos esperando, la llamada no tiene efecto.

(3) La función booleana `awaited(s)` produce el valor verdadero si hay al menos un proceso bloqueado en `s`; y falso en caso contrario.

A continuación, se da un ejemplo de una interfaz de módulo en Modula-1. Realiza el requisito de control de recurso que fue programado con semáforos anteriormente:

```
INTERFACE MODULE control_recurso;

  DEFINE asigna, desasigna; (* lista de exportaciones *)

  VAR ocupado : BOOLEAN;
      libre : SIGNAL;

  PROCEDURE asigna;
  BEGIN
     IF ocupado THEN WAIT(libre) END;
     ocupado := TRUE;
  END;

  PROCEDURE desasigna;
  BEGIN
     ocupado := FALSE;
     SEND(libre)
  END;

BEGIN (* módulo de inicialización *)
  ocupado := FALSE
END.
```

Hay que tener en cuenta que podría haberse insertado

```
if AWAITED(libre) then SEND(libre)
```

en `deallocate`, pero como el efecto de `SEND(libre)` es nulo, cuando `AWAITED(libre)` es falso, no se consigue nada haciendo la comprobación.

8.6.2 Mesa

Hasta ahora, se ha supuesto que cuando se desbloquea un proceso suspendido es porque la condición que produjo dicho bloqueo deja de ser cierta. Por ejemplo, el proceso que está bloqueado esperando la liberación del recurso (es decir, no ocupado) puede suponer que está disponible cuando empieza a ejecutarse de nuevo. De forma similar, un proceso retrasado dentro de un monitor de búfer limitado continúa con sus acciones una vez que se convierte de nuevo ejecutable.

En Mesa (Lampson y Redell, 1980), se considera una aproximación diferente. Hay una operación «wait», pero los procesos no pueden suponer que, cuando son despertados, la condición que produjo el bloqueo se ha eliminado. La operación «notify» (notifica), comparable a signal, simplemente indica que el proceso bloqueado debiera reevaluar la condición. Mesa también permite una operación «broadcast» (difunde), que notifica a todos los procesos que esperan por una condición concreta. Estos procesos son despertados uno a uno para mantener el uso exclusivo del monitor. Un proceso que ejecute una operación de notify o broadcast no es bloqueado.

Hay tres clases de procedimientos permitidos en un monitor Mesa: procedimientos de entrada, procedimientos internos, y procedimientos externos. Los procedimientos de entrada pueden operar con el bloqueo del monitor. Los procedimientos internos solo pueden ser llamados desde procedimientos de entrada. (En Modula-1, un procedimiento interno podría, simplemente, no aparecer en la lista de definición). Los procedimientos externos pueden ser llamados sin el bloqueo del monitor, y se utilizan para ver el estado actual del monitor. No pueden cambiar variables, llamar a procedimientos internos, o utilizar una variable de condición. Estas restricciones se comprueban en tiempo de compilación.

Por poner un ejemplo de un monitor en Mesa, considere un refinamiento del problema de reserva de recursos. En lugar de tener un único recurso que está ocupado o vacío, el monitor debe controlar el acceso a N instancias del recurso. Una solicitud de reserva pasa, como parámetro, el número de instancias ($\leq N$) requeridas. Para simplicar, este parámetro de solicitud no se comprueba en los procedimientos «asigna» o «desasigna». Para obtener una estructura segura, sería necesario comprobar que un proceso que devuelve un conjunto de recursos retenidos, los devuelve realmente. El control seguro de recursos se considera de nuevo en el Capítulo 11. A continuación, se presenta el código en Mesa; se incluye el procedimiento `libera_recursos` para mostrar un procedimiento externo. El lector debiera ser capaz de comprender este módulo sin más conocimientos de Mesa (hay que indicar que el operador de asignación en Mesa es <-).

```
Control_Recursos : monitor =
begin
  const N = 32;
  libre : condition;
  recursos_libres : positive <- N;

  asigna : entry procedure[talla : positive] =
    begin
      do
        if talla <= recursos_libres then exit; -- sale del bucle
        wait libre;
      endloop;
      recursos_libres <- recursos_libres - talla;
  end;

  desasigna : entry procedure[talla : positive] =
  begin
    libreslocal[talla];
    broadcast libre;
  end;

  libreslocal : internal procedure[S : positive] =
  begin
```

```
    recursos_libres <- recursos_libres + S;
  end;
  libera_recursos : external procedure return[p : positive] =
  begin
    p <- recursos_libres;
  end;
end;
```

Como una operación de desasignación podría liberar suficientes recursos para que un número de procesos bloqueados en la reserva continúe, se realiza una multidifusión. Todos los procesos bloqueados (en `libre`) se ejecutarán por turno, y o bien obtienen los recursos, si `talla` es ahora menor o igual que `recursos_libres`, o bien vuelven a ser bloqueados.

Si Mesa no soportara una funcionalidad de difusión, entoces el programador debería mantener la cuenta del número de los procesos bloqueados y desbloquear por turno los procesos en la cadena. Esto se debe a la semántica de la operación notify, que no bloquea. En Modula-1 esto se obtiene implícitamente, porque la operación send produce un cambio de contexto (context switch) en el proceso despertado.

```
PROCEDURE asigna(talla : INTEGER);
BEGIN
  WHILE talla > recursos_libres DO
    WAIT(libre);
    SEND(libre)
  END;
  recursos_libres := recursos_libres - talla
END;

PROCEDURE desasigna(talla : INTEGER);
BEGIN
  recursos_libres := recursos_libres + talla;
  SEND(libre)
END;
```

Si un lenguaje soportara los semáforos de cantidad, entonces los procedimientos de reserva y liberación serían muy sencillos:

```
procedure asigna(talla : integer);
begin
  wait(QS, talla)
end;

procedure desasigna(talla : integer);
begin
```

```
  signal(QS, talla)
end;
```

donde QS es un semáforo de cantidad inicializado con el número total de recursos en el sistema.

8.6.3 Mutexes y variables de condición POSIX

En la Sección 8.4.6, se describieron los semáforos POSIX como un mecanismo utilizable entre procesos y entre hilos. Si se soporta la extensión de hilos para POSIX, entonces utilizar semáforos para comunicación y sincronización entre hilos en el mismo espacio de direccionamiento es caro y está poco estructurado. Los **mutexes** y las **variables de condición**, cuando se combinan, proporcionan la funcionalidad de un monitor con una interfaz de procedimiento. El Programa 8.3 define la interfaz C a la excepción de las funciones que manipulan los atributos de los objetos.

Programa 8.3. Una interfaz C para mutexes y variables de condición POSIX.

```
#include <time.h>

typedef ... pthread_mutex_t;
typedef ... pthread_mutexattr_t;
typedef ... pthread_cond_t;
typedef ... pthread_condattr_t;

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
   /* inicializa un mutex con ciertos atributos */
int pthread_mutex_destroy(pthread_mutex_t *mutex);
   /* destruye un mutex */
   /* comportamiento indefinido si el mutex está bloqueado */

int pthread_mutex_lock(pthread_mutex_t *mutex);
   /* bloquea el mutex; si ya lo estaba, suspende al hilo llamador */
   /* el propietario del mutex es el hilo que lo boqueó */
int pthread_mutex_trylock(pthread_mutex_t *mutex);
   /* como el bloqueo, pero devuelve error si mutex estaba bloqueado */
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *abstime);
   /* como el bloqueo, pero devuelve error si el bloqueo no puede */
   /* ser obtenido en un tiempo limite */

int pthread_mutex_unlock(pthread_mutex_t *mutex);
   /* desbloquea mutex si lo llama el hilo propietario */
   /* comportamiento indefinido si el hilo no es el propietario */
```

(Continuación)

```
   /* comportamiento indefinido si el mutex no está bloqueado */
   /* con éxito, produce que un hilo bloqueado sea liberado */

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
   /* inicializa una variable de condición con ciertos atributos */
int pthread_cond_destroy(pthread_cond_t *cond);
   /* destruye una variable de condición */
   /* comportamiento indefinido si hay hilos esperando */
   /* por la variable de condición */

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
   /* llamado por el hilo que posee un mutex bloqueado */
   /* comportamiento indefinido si el mutex no está bloqueado */
   /* atómicamente bloquea el hilo en la variable de condición y */
   /* libera el bloqueo en el mutex */
   /* un retorno con éxito indica que el mutex ha sido bloqueado */
int pthread_cond_timedwait(pthread_cond_t *cond,
           pthread_mutex_t *mutex, const struct timespec *abstime);
   /* igual que pthread_cond_wait, excepto que se devuelve un error */
   /* si expira el tiempo límite */

int pthread_cond_signal(pthread_cond_t *cond);
   /* desbloquea como mínimo un hilo bloqueado */
   /* sin efecto si no hay hilos bloqueados */
   /* los hilos desbloqueados compiten automáticamente */
   /* por el mutex asociado */
int pthread_cond_broadcast(pthread_cond_t *cond);
   /* desbloquea todos los hilos bloqueados */
   /* sin efecto si no hay hilos bloqueados */
   /* los hilos desbloqueados compiten automáticamente */
   /* por el mutex asociado */

/* Todas las funciones anteriores devuelven 0 si tienen éxito */
```

Cada monitor tiene una variable `mutex` asociada (inicializada), y todas las operaciones en el monitor (regiones críticas) se delimitan con llamadas para bloquear (`pthread_mutex_lock`) y desbloquear (`pthread_mutex_unlock`) el `mutex`. La sincronización de condición viene dada por las variables de condición asociadas con el `mutex`. Hay que indicar que, cuando un hilo espera en una variable de condición (`pthread_cond_wait`, `pthread_cond_timedwait`), su bloqueo en el `mutex` asociado se libera. También, cuando se vuelve correctamente de

la espera condicional, de nuevo mantiene el bloqueo. Sin embargo, como más de un hilo puede ser liberado (incluso por `pthread_cond_signal`), el programador debe comprobar de nuevo la condición que le produjo la espera inicial.

Consideremos el siguiente búfer limitado entero utilizando mutexes y variables de condición. El búfer consta de un `mutex`, dos variables de condición (`bufer_no_lleno` y `bufer_no_vacio`), una cuenta del número de items en el búfer, el búfer mismo, y las posiciones del primer y último item del búffer. La rutina `agrega` bloquea el búfer, y, mientras el búfer está lleno, espera en la variable de condición `bufer_no_lleno`. Cuando el búfer tiene espacio, el item de datos entero se coloca en el búfer, se desbloquea el `mutex`, y se envía la señal `bufer_no_vacio`. La rutina `toma` tiene una estructura similar.

```
#include <pthreads.h>

#define TALLA_BUF 10

typedef struct {
  pthread_mutex_t mutex;
  pthread_cond_t bufer_no_lleno;
  pthread_cond_t bufer_no_vacio;
  int cuenta, primero, último;
  int buf[TALLA_BUF];
  } bufer;

int agrega(int item, bufer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);
  while(B->cuenta == TALLA_BUF)
     PTHREAD_COND_WAIT(&B->bufer_no_lleno, &B->mutex);
  /* pon datos en el búfer y actualiza cuenta y último */
  PTHREAD_COND_SIGNAL(&B->bufer_no_vacio);
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  return 0;
}

int toma(int *item, bufer *B ) {
  PTHREAD_MUTEX_LOCK(&B->mutex);
  while(B->cuenta == 0)
     PTHREAD_COND_WAIT(&B->bufer_no_vacio, &B->mutex);
  /* toma los datos del búfer y actualiza cuenta y primero */
  PTHREAD_COND_SIGNAL(&B->bufer_no_lleno);
  PTHREAD_MUTEX_UNLOCK(&B->mutex);
  return 0;
}

/* tambien se precisa de una función inicializa() */
```

Aunque los mutexes y las variables de condición actúan como un tipo de monitor, su semántica difiere cuando se libera un hilo desde una espera condicional y otros hilos están intentando acceder a la sección crítica. Con POSIX, no está especificado qué hilo lo consigue, a menos que se utilice una planificación basada en prioridad (véase la Sección 13.14.2). En la mayoría de los monitores se da preferencia al hilo liberado.

8.6.4 Llamadas anidadas al monitor

Hay muchos problemas asociados con el uso de monitores, pero uno de los que más atención han recibido es el de las implicaciones semánticas de la llamada a un procedimiento monitor desde otro monitor.

> La controversia está en lo que debería hacerse si un proceso que ha hecho una llamada anidada al monitor es suspendido en otro monitor. Debería renunciarse a la exclusión mutua de la última llamada al monitor, debido a la semántica de wait y operaciones equivalentes. Sin embargo, no se debe renunciar a la exclusión mutua por parte de los procesos en los monitores desde los que se han hecho las llamadas anidadas. Los procesos que intenten invocar a procedimientos en esos monitores serán bloqueados. Esto tiene implicaciones en cuanto a las prestaciones, puesto que el bloqueo disminuirá la cantidad de concurrencia exhibida por el sistema. (Lampson y Redell, 1980)

Se han sugerido varias aproximaciones al problema del monitor anidado. La más popular, adoptada por Java, POSIX y Mesa, es mantener el bloqueo. Otras aproximaciones son prohibir llamadas anidadas conjuntas a procedimientos (como en Modula-1), o proporcionar construcciones que especifiquen que ciertos procedimientos del monitor pueden liberar su bloqueo de exclusión mutua durante las llamadas remotas.

8.6.5 Críticas a los monitores

El monitor proporciona una solución estructurada y elegante a problemas de exclusión mutua como el del búfer limitado. Sin embargo, no soluciona bien las sincronizaciones de condición fuera del monitor. En Modula-1, por ejemplo, las señales son una característica general de programación, y su uso no está restringido dentro de un monitor. Por tanto, un lenguaje basado en monitores presenta una mezcla de primitivas de alto y bajo nivel. Todas las críticas relacionadas con el uso de semáforos se aplican igualmente (incluso más, si cabe) a las variables de condición.

Además, aunque los monitores encapsulan todas las entradas relacionadas con un recurso, y proporcionan la importante exclusión mutua, su estructura interna puede ser todavía difícil de leer, debido al uso de variables de condición.

8.7 Objetos protegidos

Las críticas a los monitores se centran en el uso de las variables de condición. Sustituyendo esta aproximación a la sincronización por el uso de guardas, se obtiene una abstracción más estructurada. Esta forma de monitor se denominará **objeto protegido**. Ada es el único lenguaje importante que proporciona este mecanismo que aquí será descrito en relación con Ada.

En Ada, un objeto protegido encapsula elementos de datos, y sólo se permite el acceso a ellos mediante subprogramas protegidos o entradas protegidas. El lenguaje garantiza que estos programas y entradas serán ejecutados de manera que se asegure que los datos son actualizados mediante exclusión mutua. La sincronización de condición se proporciona mediante expresiones booleanas en entradas (se trata de guardas, pero en Ada se denominan **barreras**), que deben evaluar a cierto antes que a una tarea se le permita su entrada. Por consiguiente, los objetos protegidos son más bien como monitores o regiones críticas condicionales. Proporcionan la funcionalidad estructurada de los monitores con el mecanismo de sincronización de alto nivel de las regiones críticas condicionales.

Una unidad protegida puede declararse como un tipo o como una simple instancia; tiene una especificación y un cuerpo (aquí se declara de forma similar a una tarea). Su especificación puede contener funciones, procedimientos y entradas (entry).

La declaración siguiente ilustra cómo se pueden utilizar los tipos protegidos para proporcionar exclusión mutua simple:

```
-- sobre un simple entero
protected type Entero_Compartido(Valor_Inicial : Integer) is
  function Lee return Integer;
  procedure Escribe(Nuevo_Valor : Integer);
  procedure Incrementa(Por : Integer);
private
  El_Dato : Integer := Inicial_Valor;
end Entero_Compartido;

Mi_Dato : Entero_Compartido(42);
```

El tipo protegido anterior encapsula un entero compartido. La declaración de objeto `Mi_Dato` declara una instancia del tipo protegido y pasa el valor inicial al dato encapsulado. El dato encapsulado sólo puede ser accedido por tres subprogramas: `Lee`, `Escribe` e `Incrementa`.

Un procedimiento protegido proporciona acceso mutuamente excluyente de lectura/escritura a los datos encapsulados. En este caso, las llamadas concurrentes al procedimiento `Escribe` o `Incrementa` serán ejecutadas en exclusión mutua, es decir, sólo se podrá ejecutar una de ellas cada vez.

Las funciones protegidas proveen acceso concurrente de sólo lectura para datos encapsulados. En el ejemplo anterior, esto significa que se pueden ejecutar simultáneamente muchas llamadas a Lee. Sin embargo, las llamadas a funciones protegidas siguen siendo mutuamente excluyentes con las llamadas a procedimientos protegidos. Una llamada Lee no puede ser ejecutada si hay pendiente una llamada a procedimiento; tampoco un procedimiento podrá ser ejecutado si hay pendiente una o más llamadas a función.

El cuerpo de Entero_Compartido es, simplemente:

```
protected body Entero_Compartido is
  function Lee return Integer is
  begin
     return El_Dato;
  end Lee;

  procedure Escribe(Nuevo_Valor : Integer) is
  begin
     El_Dato := Nuevo_Valor;
  end Escribe;

  procedure Incrementa(Por : Integer) is
  begin
     El_Dato := El_Dato + Por;
  end Incrementa;
end Entero_Compartido;
```

Una entrada protegida es semejante a un procedimiento protegido en el que se garantiza la exclusión mutua en la ejecución, y tiene acceso de lectura/escritura sobre los datos encapsulados. Sin embargo, una entrada protegida está guardada por una expresión booleana (o barrera) dentro del cuerpo del objeto protegido; si esta barrera se evalúa a falso cuando se realiza la llamada a la entrada, se suspende la tarea invocadora hasta que la barrera se evalúe a cierto y no existan otras tareas activas actualmente dentro del objeto protegido. Por lo tanto, las llamadas a entradas protegidas se *pueden utilizar para implementar sincronización* de condición.

Consideremos un búfer limitado compartido entre varias tareas. La especificación del búfer es:

```
-- un búfer limitado

Talla_Bufer : constant Integer := 10;
type Indice is mod Talla_Bufer ;
subtype Cuenta is Natural range 0 .. Talla_Bufer;
type Bufer is array (Indice) of Item_Dato;

protected type Bufer_Limitado is
```

```
  entry Extrae(Item: out Item_Dato);
  entry Pon(Item: in Item_Dato);
private
  Primero : Indice := Indice'First;
  Ultimo : Index := Indice'Last;
  Numero_En_Bufer : Cuenta := 0;
  Buf : Bufer;
end Bufer_Limitado;

Mi_Bufer : Bufer_Limitado;
```

Se han declarado dos entradas, que representan la interfaz pública del búfer. Los elementos de datos declarados en la parte privada son aquellos elementos que deben ser accedidos bajo exclusión mutua. En este caso, el búfer es un array, y es accedido mediante dos índices; hay también un contador que indica el número de elementos en el búfer.

El cuerpo de este tipo protegido se proporciona a continuación:

```
protected body Bufer_Limitado is

  entry Extrae(Item: out Item_Dato)
    when Numero_En_Bufer /= 0 is
  begin
    Item := Buf(Primero);
    Primero := Primero + 1; -- es mod
    Numero_En_Bufer := Numero_En_Bufer - 1;
  end Extrae;

  entry Pon(Item: in Item_Dato)
    when Numero_En_Bufer /= Talla_Bufer is
  begin
    Ultimo := Ultimo + 1; -- es mod
    Buf(Ultimo) := Item;
    Numero_En_Bufer := Numero_En_Bufer + 1;
  end Put;

end Buffer_Limitado;
```

La entrada `Extrae` es guardada por la barrera «**when** `Numero_En_Bufer /= 0`»; sólo cuando esto evalúa a verdadero una tarea, puede ejecutar la entrada `Extrae`; algo similar ocurre con la entrada `Pon`. Las barreras definen una precondición; sólo cuando evalúan a verdadero la entrada puede ser aceptada.

Aunque las llamadas a un objeto protegido pueden ser retrasadas porque el objeto esté en uso (lo que implica que no pueden ser ejecutadas con el acceso de lectura o lectura/escritura solici-

tado), Ada no ve la llamada como suspendida. Las llamadas que son retrasadas debido a que una barrera en la entrada es falsa son consideradas, sin embargo, como suspendidas, y son colocadas en una cola. La razón de esto es que:

- Se supone que las operaciones protegidas duran poco.
- Una vez comenzada una operación protegida, no puede ser suspendida su ejecución (todas las llamadas que se suspenderían potencialmente están prohibidas y activan excepciones); sólo pueden ser reencoladas (véase la Seccion 11.4).

En este caso, una tarea no debiera ser demorada durante un periodo significativo mientras intenta acceder al objeto protegido (no sólo por razones asociadas con el orden de planificación). Una vez que una llamada a procedimiento (o función) ha conseguido acceso, comenzará inmediatamente a ejecutar el subprograma; una llamada a una entrada evaluará la barrera, y, naturalmente, será bloqueada si la barrera es falsa. En la Sección 13.14.1, la estrategia de implementación requerida por el Anexo de Sistemas de Tiempo Real se considera que garantiza que una tarea nunca sea demorada cuando intenta conseguir el acceso a un objeto protegido.

8.7.1 Llamadas de entrada y barreras

Para realizar una llamada a un objeto protegido, una tarea simplemente nombra el objeto y el subprograma o entrada requerida. Por ejemplo, para colocar algún dato en el búfer limitado anterior, se precisa la llamada de la tarea a

```
Mi_Bufer.Pon(Algun_Item);
```

En cualquier momento, una entrada protegida puede estar abierta o cerrada. Está abierta si, cuando se comprueba, la expresión booleana evalúa a cierto; en caso contrario está cerrada. Generalmente, las barreras de entrada protegida de un objeto protegido son evaluadas cuando:

(a) Una tarea llama a una de sus entradas protegidas y la barrera asociada referencia a una variable o atributo que podría haber cambiado desde que la barrera fue evaluada por última vez.

(b) Una tarea abandona un procedimiento protegido o una entrada protegida y hay tareas encoladas en entradas cuyas barreras referencian variables o atributos que podrían haber cambiado desde que las barreras fueran evaluadas por última vez.

Las barreras no son evaluadas como resultado de una llamada a una función protegida. Observe que no es posible que dos tareas estén activas en una entrada o procedimiento protegido, ya que las barreras sólo son evaluadas cuando una tarea deja el objeto.

Cuando una tarea llama a una entrada o a un subprograma protegidos, el objeto protegido podría estar ya bloqueado: si una o más tareas están ejecutando funciones protegidas dentro del objeto protegido, se dice que el objeto tiene un **bloqueo de lectura** activo; si una tarea está ejecutando un procedimiento o entrada compartidos, se dice que el objeto tiene un **bloqueo de lectura/escritura** activo.

Si hay más de una tarea que llama a la misma barrera cerrada, entonces las llamadas son encoladas por defecto de la forma «primera en llegar, primera en ser servida» (FCFS). Sin embargo, este valor por defecto puede cambiarse (véase la Sección 13.14.1).

Ahora se darán dos nuevos ejemplos. Considere primero el sencillo controlador de recursos visto anteriormente para otros lenguajes. Cuando únicamente se solicita (y se libera) un único recurso, el código es sencillo:

```
protected Control_Recursos is
  entry Asigna;
  procedure Desasigna;
private
  Libre : Boolean := True;
end Control_Recursos;

protected body Control_Recursos is

  entry Asigna when Libre is
  begin
    Libre := False;
  end Asigna;
  procedure Desasigna is
  begin
    Libre := True;
  end Desasigna;

end Control_Recursos;
```

El recurso está inicialmente disponible, y por tanto el indicador `Libre` es cierto. Una llamada a `Asigna` cambia el indicador, y por tanto cierra la barrera; todas las llamadas posteriores a `Asigna` serán bloqueadas. Cuando se invoca `Desasigna`, se abre la barrera. Esto permitirá que una de las tareas que esperan prosiga, ejecutando el cuerpo de `Asigna`. El efecto de esta ejecución es cerrar la barrera de nuevo, y por tanto no serán posibles otras ejecuciones del cuerpo de la entrada (hasta que haya otra llamada a `Desasigna`).

Por raro que parezca, el controlador general de recursos (donde se solicitan o liberan grupos de recursos) no es fácil de programar utilizando solamente guardas. Las razones de esto serán explicadas en el Capítulo 11, en el que se considera el control de recursos con cierto detalle.

Cada cola de entrada tiene un atributo que indica cuántas tareas están bloqueadas actualmente. Esto se utiliza en el ejemplo siguiente. Supongamos que una tarea desea difundir totalmente un valor (de tipo `Mensaje`) a un número de tareas en espera. Las tareas en espera llamarán a una entrada `Recibe`, que sólo se abre cuando ha llegado un mensaje. En ese momento, todas las tareas en espera serán liberadas.

Aunque ahora pueden continuar todas las tareas, deben pasar a través del objeto protegido por estricto turno (dado que sólo puede haber una activa en el objeto). La última tarea que queda fuera debe entonces activar la barrera de nuevo, de forma que las posteriores llamadas a `Recibe` serán bloqueadas hasta que sea difundido un nuevo mensaje. Esta desactivación y activación de las barreras puede ser comparada con el uso de variables de condición que no tienen efecto duradero (dentro del monitor) una vez que han salido todos los procesos. El código para el ejemplo de la difusión total es como sigue (fíjese en que el atributo `Count` indica el número de tareas encoladas en una entrada):

```
protected type Difunde is
  entry Recibe(M : out Mensaje);
  procedure Envia(M : Mensaje);
private
  Nuevo_Mensaje : Mensaje;
  Mensaje_Llegado : Boolean := False;
end Difunde;

protected body Difunde is

  entry Recibe(M : out Mensaje) when Mensaje_Llegado is
  begin
    M := Nuevo_Mensaje;
    if Recibe'Count = 0 then
      Mensaje_Llegado := False;
    end if;
  end Recibe;

  procedure Envia(M : Mensaje) is
  begin
    if Recibe'Count > 0 then
      Mensaje_Llegado := True;
      Nuevo_Mensaje := M;
    end if;
  end Envia;

end Difunde;
```

Como puede que no haya tareas esperando por el mensaje, el procedimiento `Envia` tiene que comprobar el atributo `Count`. Sólo si es mayor que cero desactivará la barrera (y registrará un nuevo mensaje).

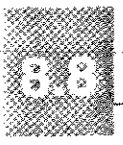
Por último, esta sección proporciona una implementación completa en Ada del paquete semáforo dado en la Sección 8.4.5. Esto muestra que los objetos protegidos no son sólo una excelente abstracción estructurada, sino que tienen la misma potencia expresiva de los semáforos.

```
package Paquete_Semaforo is
  type Semaforo(Initial : Natural := 1) is limited private;
  procedure Wait (S : in out Semaforo);
  procedure Signal (S : in out Semaforo);
private
  protected type Semaforo(Inicial : Natural := 1) is
    entry Wait_Imp;
    procedure Signal_Imp;
  private
    Valor : Natural := Inicial;
  end Semaforo;
end Paquete_Semaforo;

package body Paquete_Semaforo is
  protected body Semaforo is
    entry Wait_Imp when Valor > 0 is
    begin
      Valor := Valor - 1;
    end Wait_Imp;

    procedure Signal_Imp is
    begin
      Valor := Valor + 1;
    end Signal_Imp;
  end Semaforo;

  procedure Wait(S : in out Semaforo) is
  begin
    S.Wait_Imp;
  end Wait;

  procedure Signal(S : in out Semaforo) is
  begin
    S.Signal_Imp;
  end Signal;
end Semaforo_Package;
```

8.7.2 Objetos protegidos y tipos etiquetados

Podría decirse que Ada no integra totalmente sus modelos de concurrencia y de programación orientada a objetos. Por ejemplo, ni las tareas ni los objetos protegidos son extensibles. Está fuera del alcance de este libro considerar con detalle la mejor forma de utilizar las características de

Ada para conseguir la mayor integración posible –véase Burns y Wellings (1998)–. Además, véase Wellings et al. (2000) para una visión de conjunto sobre cómo añadir tipos protegidos extensibles en Ada.



Métodos sincronizados

En muchos aspectos, los objetos protegidos de Ada son como objetos en un lenguaje de programación orientado a objetos basado en clases. La principal diferencia, naturalmente, es que no soportan la relación de herencia. Java, que tiene la concurrencia totalmente integrada y un modelo orientado al objeto, proporciona un mecanismo en el que los monitores pueden ser implementados en el contexto de clases y objetos.

En Java, hay un bloqueo asociado con cada objeto. No puede accederse directamente a este bloqueo desde la aplicación, pero es afectado por:

- El modificador del método `synchronized`.
- La sincronización de bloque.

Cuando un método se etiqueta con el modificador `synchronized`, el acceso al metodo sólo se puede realizar una vez que se ha obtenido el bloqueo asociado con el objeto. En este caso, los métodos tienen acceso mutuamente exclusivo a los datos encapsulados por el objeto, si esos datos son accedidos solamente por otros métodos sincronizados. Los métodos no sincronizados no precisan el bloqueo, y pueden ser llamados, por tanto, en cualquier instante. Así pues, para obtener exclusión mutua total, cada método tiene que ser etiquetado como `synchronized`. Un entero compartido se representa, por tanto, por:

```
class EnteroCompartido
{
  private int elDato;

  public EnteroCompartido(int valorInicial)
  {
    elDato = valorInicial;
  }

  public synchronized int lee()
  {
    return elDato;
  };

  public synchronized void escribe(int valorNuevo)
  {
```

```
    elDato = valorNuevo;
  };
  public synchronized void incrementaEn(int cuanto)
  {
    elDato = elDato + cuanto;
  };
}

EnteroCompartido miDato = new EnteroCompartido(42);
```

La sincronización de bloque proporciona un mecanismo con el que etiquetar un bloque como sincronizado. La palabra `synchronized` toma como parámetro un objeto cuyo bloqueo necesita conseguir antes de poder continuar. Por lo tanto, los métodos sincronizados son implementables efectivamente como:

```
public int lee()
{
  synchronized(this){
    return elDato;
  }
}
```

donde **this** es la forma en Java para obtener el objeto actual.

Usado con esta total generalidad, el bloque sincronizado puede socavar una de las ventajas de los mecanismos tipo monitor: la de encapsular las restricciones de sincronización asociadas con un objeto en un único lugar del programa. Esto se debe a que no es posible comprender la sincronización asociada con un objeto concreto mirando al objeto mismo, cuando otros objetos pueden llamar a ese objeto en una sentencia sincronizada. Sin embargo, con un uso cuidadoso de esta funcionalidad se aumenta el modelo básico, y se permite programar restricciones de sincronización más expresivas, como se mostrará en breve.

Aunque los métodos o bloqueos sincronizados permiten el acceso mutuamente excluyente a los datos en un objeto, esto no es adecuado si el dato es *estático*. Los datos estáticos son datos compartidos entre todos los objetos creados a partir de una clase. Para obtener acceso mutuamente exluyente, estos datos requieren que todos los objetos estén bloqueados.

En Java, las propias clases son también objetos, y por tanto hay un bloqueo asociado con la clase. Este bloqueo puede ser accedido etiquetando un método estático con el modificador synchronized, o bien identificando el objeto de la clase en un bloque de sentencias synchronized. Lo último puede ser obtenido de la clase Object asociada con el objeto. Hay que tener en cuenta, sin embargo, que este bloqueo de clase no se obtiene cuando se realiza sincronización en el objeto. Por tanto, para obtener exclusión mutua sobre una variable estática se requiere lo siguiente (por ejemplo):

```
class VariableCompartidaEstatica
```

```
{
  private static int compartida;
  ...

  public synchronized int Lee()
  {
    synchronized(this.getClass())
    {
      return compartida;
    };
  }

  public synchronized static void Escribe(int I)
  {
    compartida = I;
  };
}
```

8.8.1 Espera y notificación

Conseguir sincronización condicional requiere soporte adicional. Éste llega de los métodos proporcionados en la clase object predefinida:

```
public void wait();
            // lanza IllegalMonitorStateException
public void notify();
            // lanza IllegalMonitorStateException
public void notifyAll();
            // lanza IllegalMonitorStateException
```

Estos métodos están diseñados para ser utilizados sólo desde métodos que mantienen el objeto bloqueado (por ejemplo los que están sincronizados). Si son llamados sin el bloqueo, se lanza la excepción `IllegalMonitorStateException`.

El método `wait` siempre bloquea al hilo invocador y libera el bloqueo asociado con el objeto. Si la llamada se realiza dentro de un monitor anidado, sólo el bloqueo más interior es asociado con el `wait` y es liberado.

El método `notify` despierta a un hilo que espera; no está definido por el lenguaje Java a cuál se despierta (sin embargo, sí que lo está en Java para tiempo real; véase la Sección 13.14.3). Hay que tener en cuenta que `notify` no libera el bloqueo, y por tanto el hilo despertado debe esperar todavía hasta obtener el bloqueo antes de poder continuar. Para despertar a **todos** los hilos que esperan es preciso utilizar el método `notifyAll`; de nuevo, esto no libera el bloqueo, y todos los hilos despertados deben esperar y competir por el bloqueo cuando llega a estar libre. Si ningún hilo está esperando, entonces `notify` y `notifyAll` no tienen efecto.

Un hilo que espera puede ser también despertado si es interrumpido por otro hilo. En este caso, se lanza `InterruptedException`. La condición no será considerada en este capítulo (la excepción tendrá permitido propagarse), pero se verá íntegramente en la Sección 10.9.

Aunque parece que Java proporciona las funcionalidades equivalentes a otros lenguajes que soportan monitores, hay una diferencia importante. *No* hay variables de condición explícitas. Por tanto, cuando un hilo es despertado, no se puede suponer necesariamente que su «condición» sea verdadera, ya que todos los hilos son despertados potencialmente sin tener en cuenta las condiciones por las que estén esperando. Para muchos algoritmos esta limitación no es un problema, ya que las condiciones por las que están esperando las tareas son mutuamente excluyentes. Por ejemplo, el búfer limitado tiene tradicionalmente dos variables de condición: `EsperaBufferNoLleno` y `EsperaBufferNoVacio`. Sin embargo, si un hilo está esperando por una condición, ningún otro hilo puede estar esperando por otra condición. Por tanto, el hilo puede suponer que cuando depierta, el búfer está en el estado apropiado.

```
public class BuferLimitado
{
  private int bufer[];
  private int primero;
  private int ultimo;
  private int numeroEnBufer = 0;
  private int talla;

  public BuferLimitado(int longitud)
  {
     talla = longitud;
     bufer = new int[talla];
     ultimo = 0;
     primero = 0;
  };

  public synchronized void pon(int item) throws InterruptedException
  {
     if (numeroEnBufer == talla) {
        wait();
     };
     ultimo = (ultimo + 1) % talla ; // % is módulo
     numeroEnBufer++;
     bufer[ultimo] = item;
     notify();
  };

  public synchronized int dame() throws InterruptedException
```

```
  {
    if (numeroEnBufer == 0) {
      wait();
    };
    primero = (primero + 1) % talla ; // % es módulo
    numeroEnBufer--;
    notify();
    return bufer[primero];
  };
}
```

Naturalmente, si se utiliza `notifyAll` para despertar hilos, dichos hilos deben reevaluar siempre sus condiciones antes de continuar.

Uno de los problemas estándar de control de concurrencia es el de los **lectores-escritores**. Según este problema, muchos lectores y muchos escritores están intentando acceder a una gran estructura de datos. Los lectores pueden leer concurrentemente, ya que no alteran los datos; sin embargo, los escritores requieren exclusión mutua sobre los datos, tanto respecto a otros escritores como frente a los lectores. Hay diferentes variaciones de este esquema; la considerada aquí es aquélla en la que se da siempre prioridad a los escritores que esperan. Por tanto, tan pronto como se dispone de un escritor, todos los nuevos lectores serán bloqueados hasta que todos los escritores hayan finalizado. Naturalmente, en situaciones extremas esto puede llevar a la inanición de los lectores.

La solución al problema de los lectores-escritores utilizando monitores estándar requiere cuatro procedimientos de monitor: `iniciaLectura`, `detieneLectura`, `iniciaEscritura` y `detieneEscritura`. Los lectores están estructurados:

```
iniciaLectura();
  // lee la estructura de datos
detieneLectura();
```

De forma similar, los escritores están estructurados:

```
iniciaEscritura();
  // escribe la estructura de datos
detieneEscritura();
```

El código que está dentro del monitor proporciona la sincronización necesaria utilizando dos variables de condición: `OkParaLeer` y `OkParaEscribir`. En Java, esto no puede ser expresado directamente dentro de un único monitor. A continuación, se consideran dos aproximaciones para resolver este problema.

La primera aproximación utiliza una única clase:

```
public class LectoresEscritores
{
```

```
    private int lectores = 0;
    private int escritoresEsperando = 0;
    private boolean escribiendo = false;

    public synchronized void iniciaEscritura() throws InterruptedException
    {
        while(lectores > 0 || escribiendo)
        {
            escritoresEsperando++;
            wait();
            escritoresEsperando--;
        }
        escribiendo = true;
    }

    public synchronized void detieneEscritura()
    {
        escribiendo = false;
        notifyAll();
    }

    public synchronized void iniciaLectura() throws InterruptedException
    {
        while(escribiendo || escritoresEsperando > 0) wait();
        lectores++;
    }

    public synchronized void detieneLectura()
    {
        lectores--;
        if(lectores == 0) notifyAll();
    }
}
```

En esta solución, despertando después de la petición wait, el hilo debe reevaluar las condiciones bajo las que puede continuar. Aunque esta aproximación permitirá múltiples lectores o un único escritor, se pude decir que es ineficiente, ya que se despiertan todos los hilos cada vez que hay datos disponibles. Muchos de esos hilos, cuando consiguen finalmente el acceso al monitor, se encontrarán con que todavía no pueden continuar, y por tanto tendrán que esperar de nuevo.

Una solución alternativa, sugerida por Lea (1997), es utilizar otra clase para implementar una variable de condición. Considere:

```
public class ConditionVariable {
```

```
  public boolean wantToSleep = false;
}
```

La aproximación general es crear instancias de estas variables dentro de otra clase y usar sincronización en bloque. Para impedir la espera en una llamada anidada al monitor, se utiliza el indicador `wantToSleep` («desea dormir») para indicar que el monitor quiere esperar en la variable de condición.

```
public class LectoresEscritores
{
  private int lectores = 0;
  private int lectoresEsperando = 0;
  private int escritoresEsperando = 0;
  private boolean escribiendo = false;

  ConditionVariable OkParaLeer     = new ConditionVariable();
  ConditionVariable OkParaEscribir = new ConditionVariable();

  public void iniciaEscritura() throws InterruptedException
  {
    synchronized(OkParaEscribir) // bloquea la variable de condicion
    {
      synchronized(this) // bloquea el monitor
      {
        if(escribiendo | lectores > 0) {
          escritoresEsperando++;
          OkParaEscribir.wantToSleep = true;
        } else {
          escribiendo = true;
          OkParaEscribir.wantToSleep = false;
        }
      } // libera el monitor
      if(OkParaEscribir.wantToSleep) OkParaEscribir.wait();
    }
  }

  public void detieneEscritura()
  {
    synchronized(OkParaLeer)
    {
      synchronized(OkParaEscribir)
      {
```

```
            synchronized(this)
            {
               if(escritoresEsperando > 0) {
                  escritoresEsperando--;
                  OkParaEscribir.notify();
               } else {
                  escribiendo = false;
                  OkParaLeer.notifyAll();
                  lectores = LectoresEsperando;
                  LectoresEsperando = 0;
               }
            }
         }
      }
   }

   public void iniciaLectura() throws InterruptedException
   {
      synchronized(OkParaLeer) {
         synchronized(this)
         {
            if(escribiendo | escritoresEsperando > 0) {
               lectoresEsperando++;
               OkParaLeer.wantToSleep = true;
            } else {
               lectores++;
               OkParaLeer.wantToSleep = false;
            }
         }
         if(OkParaLeer.wantToSleep) OkParaLeer.wait();
      }
   }

   public void detieneLectura()
   {
      synchronized(OkParaEscribir)
      {
         synchronized(this)
         {
            lectores--;
            if(lectores == 0 & escritoresEsperando > 0) {
               escritoresEsperando--;
```

```
                escribiendo = true;
                OkParaEscribir.notify();
            }
        }
    }
}
```

Cada variable de condición se representa por una instancia de la clase `ConditionVariable` declarada dentro de la clase que está actuando como monitor. Las condiciones se evalúan mientras se mantiene el bloqueo del monitor y el bloqueo sobre cualquier variable de condición que sea notificada o esperada dentro del procedimiento del monitor. Para asegurar que no ocurra interbloqueo, estos bloqueos se debieran obtener siempre en el mismo orden.

8.8.2 Herencia y sincronización

La combinación del paradigma orientado a objetos con mecanismos de programación concurrente podría producir la llamada **anomalía de herencia** (Matsuoka y Yonezawa, 1993). Se produce una anomalía de la herencia si la sincronización entre operaciones de una clase no es local, sino que depende del conjunto total de operaciones presentes en la clase. Cuando una subclase añade nuevas operaciones, puede cambiar necesariamente la sincronización definida en la clase padre para tener en cuenta estas nuevas operaciones (Matsuoka y Yonezawa, 1993).

Por ejemplo, considérese el búfer limitado presentado anteriormente en esta sección. Con Java, el código que comprueba las condiciones (`BuferNoLleno` y `BuferNoVacio`) está embebido en los métodos. En el Capítulo 11 se mostrará que, en general, esta aproximación tiene muchas ventajas, ya que permite a los métodos acceder a los parámetros del método, así como a los atributos del objeto. Sin embargo, esto produce problemas cuando se considera la herencia. Supongamos que el búfer limitado está en una subclase de modo que todos los accesos se puedan prohibir. Se añaden dos nuevos métodos: `prohibeAcceso` y `permiteAcceso`. Una extensión simple podría incluir el código siguiente:

```
public class ErrorAcceso extends Exception{};
public class BuferLimitadoControlado extends BuferLimitado
{
  // Código Incorrecto

  boolean prohibido;

  BuferLimitadoControlado(int longitud)
  {
     super(longitud);
     prohibido = false;
  };
```

```
  public synchronized void prohibeAcceso() throws InterruptedException
  {
    if (prohibido) wait();
    prohibido = true;
  }

  public synchronized void permiteAcceso() throws ErrorAcceso
  {
    if (!prohibido) throw new ErrorAcceso();
    prohibido = false;
    notify();
  }
}
```

El problema principal de esta extensión es que no tiene en cuenta el hecho que la superclase también realiza peticiones `wait` y `notify`. Por tanto, los métodos `pon` y `dame` podrían despertar como resultado de una petición `notify` en el método `allowAccess`.

El problema principal es que el código de los métodos de la superclase necesita ser modificado para reflejar las nuevas condiciones de sincronización. Ésta es la base de la anomalía de la herencia. De forma ideal, el código de `pon` y de `dame` debiera ser reutilizable completamente. Naturalmente, si se hubiera sabido cuando se escribió que se iba a producir una versión controlada, se debería haber elegido un diseño diferente para facilitar la extensión. Si embargo, no es posible anticipar todas las extensiones.

Una aproximación que se puede tomar es encapsular *siempre* las comprobaciones de condición en los bucles `while` con independencia de que parezca o no necesario hacerlo, y utilizar siempre `notifyAll`. Por tanto, si el búfer se escribe como sigue:

```
public class BuferLimitado
{
  private int bufer[];
  private int primero;
  private int ultimo;
  private int numeroEnBufer = 0;
  private int talla;

  public BuferLimitado(int longitud)
  {
    talla = longitud;
    bufer = new int[talla];
    ultimo = talla - 1;
    primero = talla - 1;
  };
```

```
public synchronized void pon(int item) throws InterruptedException
  {
    while (numeroEnBufer == talla) {
      wait();
    };
    ultimo = (ultimo + 1) % talla ;
    numeroEnBufer++;
    bufer[ultimo] = item;
    notifyAll();
  };

  public synchronized int dame() throws InterruptedException
  {
    while (numeroEnBuffer == 0) {
      wait();
    };
    primero = (primero + 1) % talla ;
    numeroEnBufer--;
    notifyAll();
    return bufer[primero];
  };
}
```

entoces la subclase podría escribirse así:

```
public class BuferLimitadoBloqueable extends BuferLimitado
{
  boolean prohibido ;

  // Código Incorrecto

  BuferLimitadoBloqueable(int longitud)
  {
    super(longitud);
    prohibido = false;
};

  public synchronized void prohibeAcceso() throws InterruptedException
  {
    while (prohibido) wait();
    prohibido = true;
  }
```

```
public synchronized void permiteAcceso() throws ErrorAcceso
  {
    if (!prohibido) throw new ErrorAcceso();
    prohibido = false;
    notifyAll();
  }

  public synchronized void pon(int item) throws InterruptedException
  {
    while(prohibido) wait();
    super.pon(item);
  }

  public synchronized int dame() throws InterruptedException
  {
    while(prohibido) wait();
    return(super.dame());
  }
}
```

Lamentablemente, hay un sutil error en esta aproximación. Consideremos el caso de un productor que está intentando poner datos en un búfer lleno. No está prohibido acceder al búfer, por lo que se llama a `super.pon(item)`, donde la llamada es bloqueada a la espera de la condición `BuferNoVacio`. Ahora, el acceso al búfer está prohibido. Otras llamadas adicionales a `dame` y `pon` se mantienen ignoradas en los métodos de la subclase, así como otras llamadas al método `prohibeAcceso`. Ahora, se hace una llamada a `permiteAcceso`; esto hace que todos los hilos que esperan sean liberados. Supongamos que el orden en el que los hilos liberados adquieren el bloqueo del monitor es: hilo consumidor, hilo que intenta prohibir el acceso al búfer, hilo productor. El consumidor encuentra que no está prohibido el acceso al búfer y toma datos del mismo. Lanza una petición `notifyAll`, pero no hay hilos esperando actualmente. El siguiente hilo que se ejecuta prohibe ahora el acceso al búfer. El hilo productor se ejecuta a continuación, y coloca un item en el búfer, ¡aunque el acceso está prohibido!

Aunque este ejemplo pueda parecer artificial, ilustra los sutiles errores que pueden ocurrir debido a la anomalía de la herencia.

Resumen

Las interacciones entre procesos requieren que los sistemas operativos y los lenguajes de programación concurrente soporten la sincronización y la comunicación entre procesos. La comunicación se puede basar en variables compartidas o en paso de mensajes. En este capítulo se han tratado las variables compartidas, las múltiples dificultades de actualización, y las sincronizacio-

nes que necesita las exclusión mutua para contrarestar estas dificultades. En esta discusión se han introducido los siguientes conceptos:

- Sección crítica: código que debe ser ejecutado bajo exclusión mutua.
- Sistema productor-consumidor: dos o más procesos que intercambian datos mediante un búfer finito.
- Espera ocupada: un proceso que comprueba continuamente una condición para ver si ya es capaz de continuar.
- Interbloqueo activo: una condición de error en la que uno o más procesos tienen prohibido progresar mientras consumen ciclos de proceso.

Se han utilizado ejemplos para mostrar la dificultad de programar la exclusión mutua utilizando sólo variables compartidas. Se han introducido los semáforos para simplificar estos algoritmos y eliminar la espera ocupada. Un semáforo es un entero no negativo que sólo puede ser actualizado mediante los procedimientos `wait` y `signal`. Las ejecuciones de dichos procedimientos son atómicas.

La provisión de una primitiva de semáforo tiene como consecuencia la introducción de un nuevo estado para un proceso, llamado suspendido. También introduce dos nuevas condiciones de error:

- Interbloqueo: un conjunto de procesos suspendidos que no pueden continuar.
- Aplazamiento indefinido: un proceso que es incapaz de continuar en tanto en cuanto los recursos no estén disponibles para él (también llamado inanición; *lockout* o *starvation*).

Los semáforos pueden ser criticados por ser su utilización de demasiado bajo nivel y propensa a errores. Siguiendo su desarrollo, se han introducido cuatro primitivas más estructuradas:

- Regiones críticas condicionales.
- Monitores.
- Objetos protegidos.
- Métodos sincronizados.

Los monitores son una característica importante del lenguaje, y se utilizan en Modula-1, Pascal concurrente y Mesa. Constan de un módulo, cuya entrada está garantizado (por definición) que se hará bajo exclusión mutua. Dentro del cuerpo de un monitor, un proceso puede ser suspendido si las condiciones no son apropiadas para que continúe. Esta suspensión se consigue utilizando una variable de condición. Cuando un proceso suspendido es despertado (por una operación `signal` en la variable de condición), es imprescindible que esto no produzca que dos procesos estén activos en el módulo al mismo tiempo. Para estar seguro que esto no sucede, un lenguaje debe prescribir alguna de las acciones siguientes:

(1) Que la operación `signal` sólo pueda ser ejecutada como la última acción de un proceso en un monitor.

(2) Que la operación `signal` tenga el efecto secundario de forzar la señalización del proceso para salir del monitor.

(3) Que el proceso señalizador sea él mismo suspendido si provoca que otro proceso llegue a estar activo en el monitor.

(4) Que el proceso señalizador no sea suspendido, y que el proceso libre deba competir por el acceso al monitor una vez que el proceso señalizador salga.

Se puede implementar una forma de monitor utilizando una interfaz de procedimientos. Dicha posibilidad se proporciona mediante mutexes y variables de condición en POSIX.

Aunque los monitores proporcionan una estructura de alto nivel para la exclusión mutua, habrá que programar algunas otras sincronizaciones mediante variables de condición de bajo nivel. Esto proporciona una desafortunada mezcla de primitivas en el diseño del lenguaje. Los objetos protegidos de Ada proporcionan las ventajas de estructuración de los monitores y los mecanismos de sincronización de alto nivel de las regiones críticas condicionales.

Integrar concurrencia con POO es un proceso plagado de dificultades. Ada intenta evitar el problema proporcionando funcionalidades separadas para POO y tipos protegidos. Java, sin embargo, trata el problema proporcionando métodos miembro sincronizados para las clases. Utilizando esta funcionalidad (junto con la sentencia `synchronized` y las primitivas `wait` y `notify`), proporciona una funcionalidad flexible de tipo monitor basada en orientación al objeto. Desafortunadamente, en esta aproximación se presenta la anomalía de la herencia.

El siguiente capítulo considera las primitivas de sincronización y comunicación basadas en mensajes. Los lenguajes que las utilizan han elevado, de hecho, el monitor a un proceso activo por su propio derecho. Como un proceso sólo puede hacer una cosa cada vez, la exclusión mutua está grarantizada. Los procesos no se comunican ya mediante variables compartidas, sino directamente. Es posible, por tanto, construir primitivas sencillas de alto nivel que combinen comunicación y sincronización. Este concepto fue considerado inicialmente por Conway (1963), y ha sido empleado en lenguajes de programación de tiempo real de alto nivel. Forma la base de las citas, tanto en Ada como en occam2.

Lecturas complementarias

Andrews, G. A. (1991), *Concurrent Programming Principles and Practice*, Redwood City, CA: Benjamin/Cummings.

Ben-Ari, M. (1990), *Principles of Concurrent and Distributed Programming*, New York: Prentice Hall.

Burns, A., y Davies, G. (1993), *Concurrent Programming*, Reading: Addison-Wesley.

Burns, A., y Wellings, A. J. (1995), *Concurrency in Ada*, Cambridge: Cambridge University Press.

Butenhof, D. R. (1997), *Programming With POSIX Threads*, Reading, MA: Addison-Wesley.

Hyde, P. (1999), *Java Thread Programming*, Indianapolis, IN: Sams Publishing.

Lea, D. (1999), *Concurrent Programming in Java: Design Principles and Patterns*, Reading, MA: Addison-Wesley.

Nichols, B., Buttlar, D., y Farrell, J. (1996), *POSIX Threads Programming*, Sebastopol, CA: O'Reilly.

Oaks, A., y Wong, H. (1997), *Java Threads*, Sebastopol, CA: O'Reilly.

Silberschatz, A., y Galvin, P. A. (1998), *Operating System Concepts*, New York: John Wiley & Sons.

Wirth, N. (1977), «Modula: a Language for Modular Multiprogramming», *Software Practice and Experience*, 7(1), 3-84.

Ejercicios

8.1 Muestre cómo puede modificarse el algoritmo de Peterson, dado en la Sección 8.2, para permitir que un proceso de prioridad alta tenga preferencia sobre otro de prioridad baja cuando hay espera ocupada.

8.2 Considere un item de datos que es compartido entre un único productor y un único consumidor. El productor es una tarea periódica que lee un sensor y escribe el valor en el item de datos compartido. El consumidor es una tarea esporádica que desea extraer el último valor colocado en el item de datos compartido por el productor.

El siguiente paquete intenta proporcionar un algoritmo genérico con el que se puedan comunicar el productor y el consumidor de forma segura, sin necesidad de exclusión mutua o espera ocupada.

```
generic
   type Datos is private;
   Valorinicial : Datos;
package Algoritmosimpson is
   procedure Escribe(Item: Datos); -- no bloqueante
   procedure Lee (Item : out Data); -- no bloqueante
end Algoritmosimpson;

package body Algoritmosimpson is

   type Hueco is (Primero, Segundo);
```

```
   Cuatrohuecos : array (Hueco, Hueco) of Datos :=
                  (Primero => (Valorinicial,Valorinicial),
                   Segundo => (Valorinicial,Valorinicial));

   Proximohueco : array(Hueco) of Hueco := (Primero, Primero);

   Ultimo : Hueco := Primero;
   Lectura : Hueco := Primero;

   procedure Lee(Item : Dato) is
      Par, Indice : Hueco;
   begin
      if Lectura = Primero then
         Par := Segundo;
      else
         Par := Primero;
      end if;
      if Latest = Primero then
         Indice := Segundo;
      else
         Indice := Primero;
      end if;

      Cuatrohuecos(Par, Indice) := Item;
      Proximohueco(Par) := Indice;
      Ultimo := Par;
   end Escribe;

   procedure Lee(Item : out Dato) is
      Par, Indice : Hueco;
   begin
      Par := Ultimo;
      Lectura := Par;
      Indice := Proximohueco(Par);
      Item := Cuatrohuecos(Par, Indice);
   end Lee;
end AlgoritmoSimpson;
```

Describir cuidadosamente cómo trabaja este algoritmo, explicando qué sucede cuando:

(1) Un `Escribe` es seguido por un `Lee`.

(2) Un `Lee` es desalojado por un `Escribe`.

(3) Un `Escribe` es desalojado por un `Lee`.

(4) Un `Lee` es desalojado por más de un `Escribe`.

(1) Un `Escribe` es desalojado por más de un `Lee`.

Comentar cómo conjuga el algoritmo comunicación segura y novedad de los datos.

Muchos compiladores optimizan el código de modo que las variables accedidas frecuentemente se mantienen en registros locales de la tarea. En este contexto, se justifican comentarios como que el algoritmo anterior precisa de comunicación segura. ¿Es preciso realizar algunos cambios para conseguir que funcione en todas las implementaciones de Ada 95?

8.3 Considere una estructura de datos compartida que pueda ser tanto leída como escrita. Mostrar cómo se pueden utilizar los semáforos para permitir muchos lectores concurrentes o un único escritor, pero no ambos a la vez.

8.4 Muestre cómo se puede implementar la «región crítica condicional» de Hoare utilizando semáforos.

8.5 Muestre cómo los monitores de Hoare (o de Mesa o de Modula-1) pueden ser implementados utilizando semáforos.

8.6 Muestre cómo pueden ser implementados los semáforos binarios utilizando un único módulo de interfaz de Modula-1. ¿Cómo debiera ser modificada la solución para vérselas con un semáforo general?

8.7 Considérese la planificación de una cola de solicitudes para transferir hacia/desde un disco. Con el fin de minimizar el movimiento de cabeza del disco, todas las solicitudes para un cilindro particular se sirven en una pasada. El planificador barre el disco arriba y abajo sirviendo solicitudes en cada cilindro por turno. Escribir un módulo de interfaz Modula-1 que proporcione la sincronización necesaria. Suponemos que hay dos procedimientos de interfaz: uno para solicitar el acceso a un cilindro particular, y otro para liberar el cilindro. Los procesos de usuario llaman a procedimientos del siguiente módulo del controlador del disco:

```
module controlador_disco;

   define lee, escribe;

   procedure lee(var datos:data_t; dir:disk_address);
   begin
      (* calcular la dirección del cilindro *)
      planificadorcabezasdisco.solicita(cilindro);
      (* lectura de datos *)
      planificadorcabezasdisco.libera(cilindro);
   end lee;

   (* de forma similar para escritura *)
end controlador_disco;
```

8.8 Escriba un módulo de interfaz Modula-1 para controlar el acceso de lectura/escritura de un archivo en una base de datos. Con el fin de mantener la integridad del archivo, sólo un proceso puede actualizarlo cada vez; sin embargo, cualquier número de procesos puede leer el archivo mientras el archivo no está siendo actualizado. Además, las solicitudes para actualizar el archivo sólo deben ser permitidas cuando no hay lectores. Sin embargo, una vez se haya recibido una solicitud para actualizar el archivo todas las solicitudes posteriores para leer el archivo deben ser bloqueadas hasta que no haya mas actualizaciones.

El módulo de interfaz debe definir cuatro procedimientos: `inicialectura`, `detienelectura`, `iniciaescritura` y `detieneescritura`. Suponga que el módulo de interfaz se utiliza de la siguiente forma:

```
module acceso_archivo;
   define leearchivo, escribearchivo;
   use inicialectura, detienelectura, iniciaescritura, detieneescritura;

   procedure leearchivo;
   begin
      inicialectura;
         lee el archivo
      detienelectura;
   end leearchivo;

   procedure escribearchivo;
   begin
      iniciaescritura;
         escribe el archivo
      detieneescritura;
   end escribearchivo;
end acceso_archivo.
```

8.9 Se está utilizando un computador para controlar el flujo del tráfico a través de un túnel de carretera de *una sola vía*. Por razones de seguridad, no debe haber nunca mas de *aproximadamente N* vehículos en el túnel cada vez. Los semáforos de la entrada controlan la entrada de tráfico, y los detectores de vehículos de la entrada y de salida se usan para medir el flujo del tráfico. Esboce la estructura de *dos* PROCESS Modula-1 y un INTERFACE MODULE que controlen entre ellos el flujo del tráfico. El primer proceso monitoriza el detector de salida de vehículos y el segundo el detector de entrada. El INTERFACE MODULE controla los semáforos. Se puede suponer que las siguientes funciones ya han sido definidas y están en el alcance:

```
procedure COCHES_SALIDOS: NATURAL;
begin
   (* devuelve el número de coches que han dejado *)
   (* el túnel desde que la función fue llamada por última vez *)
end COCHES_SALIDOS;
```

```
procedure COCHES_ENTRADOS : NATURAL;
begin
   (* devuelve el número de coches que han entrado en *)
   (* el túnel desde que la función fue llamada por última vez *)
end COCHES_ENTRADOS;

procedure ESTABLECE_LUCES( COL : COLOR);
begin
   (* coloca los semáforos a COL. *)
   (* COLOR es un tipo enumerado en el alcance definido como *)
   (* type COLOR = (ROJO, VERDE); *)
end ESTABLECE_LUCES;

procedure RETRASA_10_SEGUNDOS;
begin
   (* retrasa al proceso llamador 10 segundos *)
end RETRASA_10_SEGUNDOS;
```

Su solución tendría que leer los sensores (mediante las funciones COCHES_SALIDOS y COCHES_ENTRADOS) cada 10 segundos hasta que el tunel esté lleno o vacío. Cuando el túnel estuviera lleno (y por tanto las luces están en rojo), la tarea que monitoriza la entrada *no* debiera llamar continuamente a la función COCHES_ENTRADOS. De forma similar, cuando el túnel estuviera vacío, la tarea que monitoriza la salida *no* debiera llamar continuamente a la función COCHES_SALIDOS. Además, las tareas no deberían hacer espera ocupada. *No* se deben hacer hipótesis sobre el planificador de tareas de Modula-1.

8.10 Una de las críticas a los monitores es que la sincronización de condición no está estructurada y es de demasiado bajo nivel. Explique qué se quiere decir con esto. Una primitiva de sincronización de monitor de alto nivel debe tener la forma

```
WaitHasta expresion_booleana;
```

donde el proceso es demorado hasta que la expresión booleana se evalúa a cierto. Por ejemplo:

```
WaitHasta x < y +5;
```

debiera retrasar el proceso hasta que `x < y +5`.

Aunque esta forma de sincronización de condición es más estructurada, no se encuentra en la mayoría de los lenguajes que soportan monitores. Explique por qué es así. ¿Bajo qué circustancias debieran ser invalidas las objeciones a la funcionalidad de sincronización de alto nivel anterior? Mostrar cómo el problema del búfer limitado puede ser resuelto utililizando la primitiva de sincronización **`WaitHasta`** dentro de un monitor.

8.11 Considere un sistema de tres procesos fumadores de cigarrillos y un proceso agente. Cada fumador elabora continuamente un cigarrillo y lo fuma. Para hacer y encender un cigarri-

llo se necesitan tres ingredientes: tabaco, papel y cerillas. Uno de los procesos tiene papel, otro tabaco, y el tercero cerillas. El agente tiene una provisión infinita de los tres ingredientes. El agente coloca dos ingredientes, elegidos aleatoriamente, en una mesa. El fumador que tiene el ingrediente restante puede hacer un cigarrillo y fumarlo. Una vez que ha acabado de fumar, el fumador envía una señal al agente, que pone entonces otros dos de los tres ingredientes en la mesa, y el ciclo se repite.

Esboce la estructura de un monitor que sincronice al agente con los tres fumadores.

8.12 Contraste las funcionalidades internas proporcionadas por el núcleo de Unix (Bach, 1986) para exclusión mutua y sincronización de condición con las correspondientes funcionalidades proporcionadas por los semáforos.

8.13 Muestre cómo pueden usarse las funcionalidades internas proporcionadas por Unix (Bach, 1986) para controlar el acceso a una estructura de datos compartida. Suponga que hay muchos lectores y muchos escritores. Sin embargo, aunque muchos lectores deben tener permitido el acceso a la estructura de datos simultáneamente, sólo está permitido que un escritor acceda cada vez. Además, no está permitida una mezcla de lectores y escritores. La solución debe dar, además, prioridad a los lectores.

8.14 Muestre cómo pueden implementarse las operaciones sobre un semáforo en el núcleo de un sistema operativo para un sistema con un único procesador *sin* espera ocupada. ¿Qué funcionalidad hardware requiere su solución?

8.15 Muestre cómo pueden usarse los mutexes y las variables de condición de POSIX para implementar un estructura de datos compartida que pueda ser tanto leída como escrita. Permita muchos lectores a la vez o un único escritor, pero no las dos cosas.

8.16 Muestre cómo pueden usarse los mutexes y las variables de condición POSIX para implementar un controlador de recursos.

8.17 Complete el Ejercicio 8.8 utilizando objetos protegidos de Ada para sincronización de procesos.

8.18 Compare y contraste las funcionalidades proporcionadas por los mutexes y las variables de condición con aquéllas proporcionadas por los objetos protegidos de Ada.

8.19 Rehaga el Ejercicio 8.11 utlizando objetos protegidos.

8.20 Implemente semáforos de cantidad utilizando objetos protegidos.

8.21 Muestre cómo pueden usarse uno o más objetos protegidos Ada para implementar monitores de Hoare.

8.22 Explique la sincronización impuesta por el siguiente objeto protegido Ada.

```
protected type Barrera(Requerido : Positive) is
   entry Espera;
private
```

```
   Liberacion : Boolean := False;
end Barrera;
protected body Barrera is

   entry Espera when Espera'Count = Requerido or Liberacion is
   begin
      if Espera'Count = 0 then
         Liberacion = False;
      else
         Liberacion := True;
      end if;
   end Espera;
end Barrera;
```

El siguiente paquete proporciona un enlace simplificado en Ada para los mutexes y las variables de condición de POSIX. Todos los mutexes y variables de condición son inicializados con los atributos por defecto.

```
package Pthreads is
   type Mutex_T is limited private;
   type Cond_T is limited private;

   procedure Mutex_Initialise (M: in out Mutex_T);
   procedure Mutex_Lock(M: in out Mutex_T);
   procedure Mutex_Trylock(M: in out Mutex_T);
   procedure Mutex_Unlock(M: in out Mutex_T);

   procedure Cond_Initialise(C: in out Cond_T);
   procedure Cond_Wait(C: in out Cond_T;
                       M :in out Mutex_T);

   procedure Cond_Signal(C: in out Cond_T);
   procedure Cond_Broadcast(C: in out Cond_T);
private
   ...
end Pthreads;
```

Muestre cómo pueden implementarse las barreras, tal y como se han definido anteriormente, mediante este paquete. *No* utilice ninguna de las funciones Ada de sincronización y comunicación.

8.23 Utilizando el paquete `Pthreads` de Ada visto en la pregunta anterior, rehaga la respuesta para el Ejercicio 8.7 utilizando Ada. No utilice las funcionalidades de sincronización de Ada. Suponga que hay dos procedimientos interfaz: uno para solicitar el acceso a un cilindro particular, y otro para liberar el cilindro.

```
package Planificador_Cabezas_Disco is
   Size_Of_Disk : constant := ...;
   type Direccion_Disco is range 1 .. Talla_De_Disco;
   type Direccion_Cilindro is mod 1 ..20;
   procedure Solicita(Dest: Direccion_Cilindro);
   procedure Libera(Dest: Direccion_Cilindro);
end Planificador_Cabezas_Disco
```

Los usuarios invocan procedimientos del siguiente paquete del manejador de disco:

```
with Planificador_Cabezas_Disco; use Disco_Cabezas_Disco;
package Controlador_Disco is
   procedure Lee(Datos: out Datos_T; Dir : Direccion_Disco);
   procedure Escribe(Datos: in Datos_T; Dir : Direccion_Disco);
end Controlador_Disco;

package body Controlador_Disco is

   procedure Lee(Datos: out Datos_T; Dir : Direccion_Disco) is
      Cilindro : Direccion_Cilindro;
   begin
      -- utiliza la dirección del cilindro
      Planificador_Cabezas_Disco.Solicita(Cilindro);
      -- lee datos
      Planificador_Cabezas_Disco.Solicita(Cilindro);
   end Lee;

   -- igualmente para Escribe

end Controlador_Disco;
```

La solución no debiera contener ninguna tarea o hilos/procesos POSIX.

8.24 El siguiente paquete define una abstracción de semáforo en Ada.

```
generic
   Inicial : Natural:= 1; -- valor inicial por defecto del semáforo
package Paquete_Semaforo is
   type Semaforo is limited private;
   procedure Wait (S : Semaforo);
   procedure Signal (S : Semaforo);
private
   type Semaforo is ...; -- no necesario para la cuestión
end Paquete_Semaforo;
```

Utilizando `Paquete_Semaforo`, muestre cómo se pueden implementar los siguientes paradigmas de comunicación (y la especificación de su paquete asociado).

Una multidifusión (multicast) se produce cuando una tarea es capaz de enviar los mismos datos a diferentes tareas que los esperan. A continuación, se proporciona una especificación de un paquete para la abstracción `Multidifusion`.

```
package Multidifusion is

   procedure Envia(I : Integer);
   procedure Recibe(I : out Integer);

end Multidifusion;
```

Las tareas del receptor indican su buena voluntad para recibir datos llamando al procedimiento `Recibe` definido en el paquete dado anteriormente (en este ejemplo, los datos son de tipo `Integer`). Las tareas son bloqueadas por esta llamada. Una tarea emisora indica que desea difundir datos llamado al procedimiento `Envia`. Todas las tareas que están bloqueadas actualmente en la recepción son liberadas cuando el emisor invoca el procedimiento `Envia`. Una vez ha finalizado el procedimiento `Envia`, cualquier nueva llamada a `Recibe` debe esperar al siguiente emisor.

Muestre cómo puede implementarse el cuerpo del paquete `Multidifusion` con semáforos.

8.25 Una difusión total (Broadcast) es similar a una multidifusión, excepto en que todos los receptores deben recibir los datos. A continuación se proporciona la especificación de un paquete para `Difusion_Total` (multidifusión total).

```
package Difusion_Total is

   -- para 10 tareas

   procedure Envia(I : Integer);
   procedure Recibe(I : out Integer);

end Difusion_Total;
```

Considere, por ejemplo, un sistema con 10 tareas receptoras. Estas tareas llaman todas al procedimiento `Recibe` cuando están dispuestas a recibir la difusión. Las tareas son bloqueadas por esta llamada. Una tarea emisora indica que desea difundir datos llamado al procedimiento `Envia`. El procedimiento `Envia` espera hasta que las diez tareas estén dispuestas a recibir la difusión antes de liberar los receptores y pasar los datos. Si hay más de una llamada a `Envia`, entonces son encoladas.

Muestre cómo puede implementarse el cuerpo del paquete `Difusion_Total` utilizando el paquete semáforo dado en el Ejercicio 8.24.

8.26 Se ha sugerido que York podría poner un límite en el número de vehículos a motor que pueden entrar en la ciudad en un momento dado. Una propuesta es establecer controles de monitorización en cada una de las «barras» de la ciudad (entradas a través de las murallas de la ciudad), y cuando la ciudad esté llena, colocar en rojo los semáforos para el tráfico entrante. Para indicar a los conductores que la ciudad está llena, la luz roja estará parpadeando.

Con el fin de conseguir este objetivo, se colocan sensores de presión en la carretera en las barras de entrada y en los puntos de salida. Cada vez que un vehículo entra en la ciudad, se activa una señal para una tarea `ControladorBarra` (como un punto de entrada a una tarea), y lo mismo cuando un vehículo sale:

```
Max_Coches_En_Ciudad_Para_Luz_Roja : constant Positive := N;
Min_Coches_En_Ciudad_Para_Luz_Verde : constant Positive := N - 10;

type Barra is (Walmgate, Goodramgate, Micklegate,
               Bootham, Barbican);

task type Controlador_Barra(G : Bar) is
   entry Coche_Entrado;
   entry Coche_Salido;
end Controlador_Barra;

Walmgate_Controlador_Barra : Controlador_Barra(Walmgate);
Goodramgate_Controlador_Barra : Controlador_Barra(Goodramgate);
Micklegate_Controlador_Barra : Controlador_Barra(Micklegate);
Bootham_Controlador_Barra : Controlador_Barra(Bootham);
Barbican_Controlador_Barra : Controlador_Barra(Barbican);
```

Muestre cómo puede coordinarse el cuerpo de estas tareas para que UNA de ellas llame a `Controlador_Luces_Trafico_Ciudad` (con la siguiente especificación de tarea) para indicar si se permitirán o no más coches.

```
task Controlador_Luces_Trafico_Ciudad is
   entry Ciudad_Esta_Llena;
   entry Ciudad_Tiene_Espacio;
end Controlador_Luces_Trafico_Ciudad;

task body Controlador_Luces_Trafico_Ciudad is separate;
-- el cuerpo no interesa en esta cuestión
```

8.27 Explique por qué el controlador de recursos proporcionado en la Sección 8.4.6 padece de una condición de carrera. Cómo puede modificarse el algoritmo para eliminar esta condición.

8.28 Muestre cómo se puede implementar en Java el problema de los lectores/escritores de modo que se dé prioridad a los lectores y se garantice a los escritores servicio en un orden FIFO.

8.29 Muestre cómo se puede utilizar Java para implementar un controlador de recursos.

8.30 Implemente los semáforos de cantidad utilizando Java.

8.31 Rehaga el ejemplo del búfer limitado en Java utilizando dos instancias de la clase `ConditionVariable`, una para la condición `BuferNoLleno`, y otra para `BuferNoVacio`.

8.32 Aplique la solución anterior al problema del búfer limitado para producir un búfer cuyo acceso pueda ser prohibido. ¿Resuelve esta aproximación la anomalía de la herencia vista en la Sección 8.8.2?

8.33 Considere la siguiente clase Java

```
public class Evento
{

public synchronized void WaitAltaPrioridad();
public synchronized void WaitBajaPrioridad();
public synchronized void signalEvento();
}
```

Muestre cómo se implementa esta clase de forma que `signalEvento` libere un hilo de espera de alta prioridad si está esperando uno. Si no hay hilo de alta prioridad esperando, libera un hilo de espera de baja prioridad. Si no hay ningún hilo esperando, `signalEvento` no tiene efecto.

Considere ahora el caso en el que se puede asociar una identidad a los métodos. Cómo puede modificarse el algoritmo de forma que `signalEvento` despierte el hilo bloqueado apropiado.

Capítulo 9

Sincronización y comunicación basadas en mensajes

La alternativa a la sincronización y comunicación mediante variables compartidas se basa en el paso de mensajes. La aproximación está tipificada por el uso de una única construcción, tanto para la sincronización como para la comunicación. En esta amplia categoría existe, sin embargo, una gran variedad de modelos de lenguaje. Esta variedad en la semántica del paso de mensajes surge, y está dominada, por tres temas:

(1) El modelo de sincronización.

(2) El método de nombrado de los procesos.

(3) La estructura del mensaje.

Inicialmente, se considerarán cada uno de estos tres temas. Después, se discutirán los modelos de paso de mensajes de varios lenguajes, incluyendo Ada y occam2, con respecto al modelo POSIX para tiempo real. Java no soporta explícitamente un modelo de paso de mensajes, aunque se podrían producir clases que implementaran ese modelo. Sin embargo, como no se introducen nuevas características del lenguaje, no se hablará de Java en este capítulo.

9.1 Sincronización de procesos

En todos los sistemas basados en mensajes, existe un tipo de sincronización implícita: un proceso receptor no puede obtener un mensaje antes de que dicho mensaje haya sido enviado. Aunque esto es bastante obvio, habrá que compararlo con el uso de una variable compartida; en este ca-

so, un proceso receptor puede leer una variable y no saber si ha sido escrita por el proceso emisor. Si un proceso ejecuta una recepción de mensaje incondicional cuando no existe ningún mensaje disponible, entonces permanecerá suspendido hasta que llegue el mensaje.

De la semántica de la operación «envía» surgen las variaciones en el modelo de sincronización de procesos, que pueden ser clasificadas como sigue:

- **Asíncrona (o sin espera)**: el emisor continúa inmediatamente, independientemente de si se ha recibido o no el mensaje. El «envía» asíncrono figura en CONIC (Sloman et al., 1984) y en varios sistemas operativos, incluyendo POSIX.

- **Síncrona**: el emisor continua sólo cuando se ha recibido el mensaje. El «envía» síncrono se utiliza en CSP (Hoare, 1985), y en occam2.

- **Invocación remota**: el emisor continúa únicamente cuando se ha devuelto una respuesta desde el receptor. El «envía» de la invocación remota modela el paradigma de comunicación petición-respuesta, y se encuentra en Ada, SR (Andrews y Olsson, 1993), CONIC (Sloman et al., 1984), y en varios sistemas operativos.

Para apreciar la diferencia entre estas aproximaciones, considérese la siguiente analogía. Echar una carta al correo es un envío asíncrono: una vez que la carta ha sido depositada en el buzón, el remitente continúa con su vida, y sólo mediante una contestación a dicha carta el remitente podrá saber si, en realidad, llegó la primera carta. Desde el punto de vista del receptor, una carta sólo puede informar al lector sobre un suceso ya pasado; no dice nada sobre la situaión actual del remitente. (Todo el mundo ha recibido postales de vacaciones de gente que había regresado a su trabajo hacía ya dos semanas.)

El teléfono es una analogía adecuada para la comunicación síncrona. Aquí, el emisor espera que se realice el contacto y se verifique la identidad del receptor antes de enviar el mensaje. Si el receptor puede reponder inmediatamente (es decir, durante la misma llamada), la sincronización es una invocación remota. Dado que el emisor y el receptor «se reunen» en una comunicación sincronizada, ésta suele conocerse como **rendezvous, cita, encuentro** o **reunión**. La forma de invocación remota se conoce como **cita extendida**, ya que se pueden realizar computaciones arbitrarias antes de que se envíe la respuesta (es decir, durante la cita).

Claramente, hay una relación entre estas formas de envío. Dos eventos asíncronos pueden construir una relación síncrona si siempre se envía (y se espera) un mensaje de reconocimiento.

```
        P1                                P2

envía_asinc (mensaje)             espera (mensaje)

espera (reconocimiento)           envía_asinc (reconocimiento)
```

Además, se pueden utilizar dos comunicaciones síncronas para construir una invocación remota:

```
        P1                              P2
envía_sinc (mensaje)            espera (mensaje)
espera (respuesta)                 ...
                                   construye respuesta
                                   ...
                                envía_sinc (respuesta)
```

Como se puede utilizar un envío asíncrono para construir los otros dos, se podría plantear que este modelo proporciona la mayor flexibilidad, y que debiera ser uno de los que adoptaran los lenguajes y los sistemas operativos. Sin embargo, utilizando este modelo existen varias desventajas:

(1) Se necesitan potencialmente infinitos búferes para almacenar los mensajes que no se han leído todavía (quizás porque el receptor haya terminado).

(2) Como la comunicación asíncrona está anticuada, la mayoría de los envíos se programan para esperar un reconocimiento (es decir, una comunicación síncrona).

(3) Se necesitan más comunicaciones con el modelo asíncrono, por lo que los programas son más complejos.

(4) Es más difícil probar la corrección del sistema completo.

Hay que señalar que, si se desea la comunicación asíncrona en un lenguaje de paso de mensajes sincronizado, los procesos búfer se pueden construir fácilmente. Sin embargo, la implementación de un proceso tiene un coste, y, por consiguiente, tener muchos procesos búfer podría tener un efecto perjudicial sobre las prestaciones globales del sistema.

9.2 Nombrado de procesos y estructura de mensajes

El nombrado de procesos implica dos subtemas: dirección frente a indirección, y simetría. En un esquema de nombrado directo, el emisor de un mensaje nombra explícitamente al receptor:

```
envía <mensaje> a <nombre-proceso>
```

Con un esquema de nombrado indirecto, el emisor nombra a alguna entidad intermedia (conocida como canal, buzón, enlace o tubería):

```
envía <mensaje> a <buzón>
```

Hay que señalar que incluso con un buzón, el mensaje que pasa puede ser aún síncrono (es decir, el emisor esperará hasta que el mensaje sea leído). El nombrado directo tiene la ventaja de la simplicidad, mientras que el nombrado indirecto facilita la descomposición del software; un buzón se puede ver como una interfaz entre distintas partes del programa.

Un esquema de nombrado es simétrico si tanto el emisor como el receptor se pueden nombrar entre sí (directa o indirectamente):

```
envía <mensaje> a <nombre-proceso>
espera <mensaje> de <nombre-proceso>
```

```
envía <mensaje> a <buzón>
espera <mensaje> de <buzón>
```

Es asimétrico si el receptor no nombra una fuente específica pero acepta mensajes de cualquier proceso (o buzón):

```
espera <mensaje>
```

El nombrado asimétrico se adapta al paradigma cliente-servidor, donde los procesos «servidor» proporcionan servicios como respuesta a los mensajes de cualquier número de procesos «cliente». Por tanto, una implementación, debe ser capaz de mantener una cola de procesos esperando por el servidor.

Si el nombrado es indirecto, entoces hay que considerar otros temas. El intermediario debería tener:

- Una estructura «muchos a uno» (es decir, cualquier número de procesos podría escribir en él, pero sólo un proceso podría leer de él); esto todavía concuerda con el paradigma cliente-servidor.
- Una estructura «muchos a muchos» (es decir, muchos clientes y muchos servidores).
- Una estructura «uno a uno» (es decir, un cliente y un servidor). Hay que señalar que con esta estructura no es necesario que el sistema de soporte de ejecución mantenga colas.
- Una estructura «uno a muchos»: esta situación es útil cuando un proceso desea enviar una solicitud a un grupo de procesos trabajadores y no le importa qué proceso sirve la solicitud.

9.2.1 Estructura del mensaje

De forma ideal, un lenguaje debiera permitir que cualquier objeto de datos de cualquier tipo definido (predefinido o del usuario) sea transmitido en un mensaje. Alcanzar este ideal es difícil, particularmente si los objetos de datos tienen representaciones diferentes en el emisor y en el receptor, y más difícil aún si la representación incluye punteros (Herlihy y Liskov, 1982). Por estas dificultades, algunos lenguajes (como occam-1) han restringido el contenido de los mensajes a objetos fijos no estructurados de tipo definido por el sistema. Los lenguajes más modernos han eliminado esas restricciones, aunque los sistemas operativos modernos aún requieren que los datos sean convertidos en bytes antes de la transmisión.

9.3 Semántica del paso de mensajes de Ada y occam2

Tanto Ada como occam2 permiten que la comunicación y la sincronización se basen en el paso de mensajes. Con occam2, éste es el único método disponible; con Ada, como ya se ha visto en el capítulo anterior, también es posible la comunicación a través de variables compartidas y tipos protegidos. Existen, sin embargo, diferencias importantes entre los esquemas basados en mensajes que incorporan los dos lenguajes. Resumiendo, Ada utiliza invocación remota asimétrica directa, mientras que occam2 incorpora paso de mensajes síncrono simétrico indirecto. Ambos lenguajes permiten que los mensajes tengan una estructura flexible. Los dos serán descritos a continuación; inicialmente occam2, debido a su semántica más sencilla.

9.3.1 El modelo occam2

Los procesos en occam2 no son nombrados; por tanto, es necesario utilizar durante la comunicación un nombrado indirecto mediante un **canal**. Cada canal sólo puede ser utilizado por un único escritor o un único lector. Ambos procesos nombran el canal; la sintaxis es a veces escueta:

```
ch ! X     -- escribe el valor de la expresión X
           -- en el canal ch

ch ? Y     -- lee del canal ch
           -- en la variable Y
```

En lo anterior, la variable `Y` y la expresión `X` serán del mismo tipo. La comunicación es síncrona; por tanto, cualquiera que sea el primer proceso que acceda al canal será suspendido. Cuando llegue el otro proceso, los datos pasarán desde `X` a `Y` (esto puede verse como una asignación distribuida `Y :=X`). Los dos procesos continuarán entonces sus ejecuciones concurrente e independientemente. Para ilustrar esta comunicación, consideremos dos procesos que se pasan 1.000 enteros entre ellos:

```
CHAN OF INT ch:
PAR
  INT V:
  SEQ i = 0 FOR 1000 --- proceso 1
    SEQ
      -- genera el valor V
      ch ! V
  INT C:
  SEQ i = 0 FOR 1000 --- proceso 2
    SEQ
```

```
ch ? C
-- usa C
```

En cada interacción de los dos bucles, se produce una cita (rendezvous) entre los dos procesos.

Los canales en occam2 pueden ser tipados, y se pueden definir para que pasen objetos de cualquier tipo, incluyendo tipos estructurados. Se pueden definir también arrays de canales.

Es importante darse cuenta de que las operaciones de entrada y salida en los canales se consideran primitivas fundamentales del lenguaje, y constituyen dos de las cinco primitivas de procesos en occam2. Las otras son `SKIP`, `STOP`, y la asignación (véase el Capítulo 3). En comparación, la comunicación y la sincronización no tienen un papel fundamental en Ada.

9.3.2 El modelo Ada

La semántica de la invocación remota tiene muchas semejanzas superficiales con una llamada a procedimientos. Los datos pasan a un receptor, el receptor ejecuta, y entonces los datos son devueltos. Debido a esta semejanza, Ada permite la definición de mensajes de un programa de una forma que es compatible con la definición de procedimientos, subprogramas protegidos y entradas. En particular, los modelos de paso de parámetros son idénticos (es decir, hay un único modelo que se utiliza en todas las situaciones).

Para que una tarea reciba un mensaje, debe definir un `entry` (entrada). Como antes, se permite cualquier número de parámetros, de cualquier modo y de cualquier tipo. Por ejemplo:

```
task type Salida_Pantalla(Id : Identificador_Pantalla) is
  -- una definición de tipo de tarea
  entry Llamada (Valor : Character; X_Coordenada,
                 Y_Coordenada: Integer);
end Salida_Pantalla;

Consola: Salida_Pantalla(Tty1);
-- donde Tty1 es de tipo Identificador_Pantalla

task Servidor_Tiempo is -- una definición sencilla de tarea
  entry Lee_Tiempo (Ahora : out Time);
  entry Establece_Tiempo (Nuevo_Tiempo : Time);
end Servidor_Tiempo;
```

Cualquier `entry` se puede definir como privado; esto significa que sólo podrá ser llamado por tareas locales al cuerpo de la tarea. Por ejemplo, consideremos el siguiente tipo de tarea `Operador_Telefono`. Proporciona tres servicios: una entrada de información que solicita el nombre y dirección de un subscriptor, una entrada de información alternativa que solicita el nombre y el código postal de un subscriptor, y un servicio de información de fallos que solicita el número de la línea defectuosa. La tarea tiene también una entrada privada para ser usada por sus tareas internas:

```
task type Operador_Telefono is
  entry Consulta_Directorio(Persona : in Nombre; Dir : in Direccion;
                            Num : out Numero);

  entry Consulta_Directorio(Persona : in Nombre;
        CP : in Codigo_Postal; Num : out Numero);

  entry Informe_Fallo(Num : Number);
private
  entry Asignar_Trabajador_Reparacion(Num out Numero);
end Operador_Telefono;
```

Ada proporciona también una funcionalidad mediante la que puede ser definido un array de entradas (conocidas como **familias de entradas**). Por ejemplo, considere un multiplexor con siete canales de entrada. En lugar de representar cada uno de ellos como una entrada separada, Ada permite que se defina como una familia.[1]

```
type Número_Canal is new Integer range 1 ..7;
task Multiplexor is
  entry Canales(Número_Canal)(Datos: Datos_Entrada);
end Multiplexor;
```

Lo anterior define siete entradas, todas con la misma especificación de parámetro.

Llamar a una tarea (es decir, enviar un mensaje) simplemente implica nombrar la tarea receptora y su entrada (el nombrado es directo); por ejemplo:

```
Consola.Llamada(Char,10,20); -- donde char es un carácter

Multiplexor.Canales(3)(D);
-- donde 3 indica el índice en la familia de entradas
-- y D es de tipo Datos_Entrada

Servidor_Tiempo.Lee_Tiempo(T); -- donde T es de tipo Time
```

En el último ejemplo, hay que indicar que los únicos datos que se transfieren lo hacen pasando en la dirección opuesta al mensaje mismo (mediante un parámetro «out»). Esto puede conducir a una confusión en la terminología, de modo que el término «paso de mensajes» no se aplica normalmente en Ada. La frase «cita (rendezvous) extendida» es menos ambigua.

Las entradas protegidas y las entradas de tarea de Ada son idénticas desde la perspectiva de la tarea que llama.

Si una llamada de entrada se realiza para una tarea que no continúa activa, se genera la excepción `Tasking_Error` en el punto de llamada. Esto permite que se tome una acción alternativa si, inesperadamente, una tarea ha terminado prematuramente, como se ve a continuación.

[1] Ada también permite familias de entradas protegidas y entradas protegidas privadas.

```
begin
  Consola.Llamada(C,I,J);
exception
  when Tasking_Error => -- informa error y continúa
end;
```

Nótese que esto no es equivalente a comprobar de antemano que la tarea esta disponible:

```
if Consola'Terminated then
  -- informa error y continúa
else
  Consola.Llamada(C,I,J);
end if;
```

Un entrelazado podría hacer que una tarea terminara después de haber sido evaluado el atributo pero antes de haber pasado la llamada.

Recibir un mensaje implica aceptar una llamada a la entrada apropiada:

```
accept Llamada(C: Character; I,J : Integer) do
  Array_Local(I,J) : = C;
end Llamada;

accept Lee_Tiempo(Now : out Time) do
  Now := Clock;
end Lee_Tiempo;

accept Canales(3)(Datos: Datos_Entrada) do
  -- alamacena datos desde el tercer canal en la familia
end Canales;
```

Una sentencia `accept` (de aceptación) debe estar en un cuerpo de tarea (no en un procedimiento llamado), y se puede colocar en el lugar de cualquier otra sentencia. Puede colocarse incluso dentro de otra sentencia `accept` (aunque no en la misma entrada). Todas las entradas (y miembros de la familia) debieran tener un `accept` asociado a ellas. Cada `accept` nombra la entrada implicada, pero no la tarea desde la que se pide la llamada. El nombrado es, por tanto, asimétrico.

Para dar un ejemplo sencillo de dos tareas interaccionando, considere, como en el ejemplo de occam2, dos tareas que realizan un bucle y después se pasan datos entre ellas. En el código Ada, las tareas intercambiarán datos:

```
procedure Test is

  Numero_De_Intercambios : constant Integer := 1000;

  task T1 is
    entry Intercambia (I : Integer; J : out Integer);
```

```
   end T1;

   task T2;

   task body T1 is
      A,B : Integer;
   begin
      for K in 1 .. Numero_De_Intercambios loop
         -- produce A
         accept Intercambio (I : Integer; J : out Integer) do
            J :=A;
            B :=I;
         end Intercambio;
         -- consume B
      end loop;
   end T1;

   task body T2 is
      C,D : Integer;
   begin
      for K in 1 .. Numero_De_Intercambios loop
         -- produce C
         T1.Intercambia(C,D);
         -- consume D
      end loop;
   end T2;

begin
   null;
end Test;
```

Aunque la relación entre las dos tareas (T1 y T2) es esencialmente simétrica, el nombrado asimétrico en Ada requiere que tengan formas completamente diferentes. Esto se debiera comparar con el código occam2, que mantiene la simetría.

9.3.3 El manejo de excepciones y la cita

Como se puede ejecutar cualquier código Ada durante una cita, existe la posibilidad que se pueda activar una excepción con la propia sentencia `accept`. Si esto ocurre, entonces pueden ocurrir dos cosas:

- Hay un manejador de excepción válido con el `accept` (bien como parte de la sentencia `accept`, bien en un bloque anidado con la sentencia `accept`), en cuyo caso el `accept` terminará normalmente.

- La excepción generada no es manejada en el `accept`, y éste termina inmediatamente.

En este último caso, la excepción nominada será rehabilitada tanto en la tarea llamada como en la invocadora. La tarea llamada recibirá la excepción generada inmediatamente después del `accept`; la tarea invocadora recibirá la excepción generada después de invocar la entrada.

Para la ilustrar la interacción entre la cita y los modelos de excepción, consideremos una tarea que actúa como un servidor de archivos. Una de sus entradas permitirá a una tarea cliente abrir un fichero:

```
task Manejador_Archivo is
  entry Abre(F : Tipo_Archivo);
  ...
end Manejador_Archivo;

task body Manejador_Archivo is
  ...
begin
  loop
    begin
      ...
      accept Abre(F : Tipo_Archivo) do
        loop
          begin
            Abre_Dispositivo(F);
            return;
          exception
            when Dispositivo_Apagado =>
              Inicia_Dispositivo;
          end;
        end loop;
      end Abre;
      ...
    exception
      when No_Existe_Archivo =>
        null;
    end;
  end loop;
end Manejador_Archivo;
```

En este código, el `Manejador_Archivo` invoca a un manejador de dispositivo para abrir el archivo especificado. La solicitud puede tener éxito o conducir a una de las dos excepciones habilitadas: `Dispositivo_Apagado` o `No_Existe_Achivo`. La primera excepción es manejada con el `accept`: se intenta iniciar el dispositivo y entonces se repite la solicitud de apertura. Como

el manejador de excepciones está con el `accept`, el cliente no es consciente de esta actividad (aunque si el dispositivo reusa arrancar, será suspendida definitivamente). La segunda excepción se debe a una solicitud fallida por parte de la tarea del usuario. Por tanto, no se genera en el `accept`, y se propagará a la tarea que le invoca, que necesitará protegerse a sí misma contra la excepción:

```
begin
  Manejador_Archivo.Abre(Nuevo_Archivo);
exception
  when Not_Existe_Archivo =>
    Manejador_Archivo.Crea(Nuevo_Archivo);
    Manejador_Archivo.Abre(Nuevo_Archivo);
end;
```

Hay que indicar que la tarea servidor también se protege a sí misma contra esta excepción definiendo un bloque en la construcción del bucle exterior.

9.4 Espera selectiva

En todas las formas de paso de mensajes que se han discutido anteriormente, el receptor de un mensaje debía esperar hasta que el proceso especificado, o canal, liberara la comunicación. Esto es, en general, bastante restrictivo. Un proceso receptor puede en realidad desear esperar para que le llame cualquiera de varios procesos. Los procesos servidor reciben mensajes de solicitud de cierto número de clientes; el orden en el que llaman los clientes es desconocido por los servidores. Para facilitar esta estructura común de programa, se permite a los procesos receptores esperar selectivamente cierto número de mensajes posibles. Pero para comprender la espera totalmente selectiva, se explicarán inicialmente las operaciones con guarda de Dijkstra (Dijkstra, 1975).

Una operación con guarda es aquélla que se ejecuta sólo si su guarda se evalúa a `VERDADERO`. Por ejemplo:

```
x <y ->m:=x
```

Esto significa que si `x` es menor que `y`, entonces asigna el valor de `x` a `m`. Una operación con guarda no es una sentencia en sí misma, sino un componente de un **conjunto de operaciones con guarda** del que hay un cierto número. Aquí, hay preocupación sólo con la construcción elección, o alternativa:

```
if x <=y->m:=x
□ x >= y -> m:= y
fi
```

El símbolo □ significa elección. En el ejemplo anterior, la ejecución del programa asignará `x` a `m` o `y` a `m`. Si ambas alternativas son posibles, es decir, si ambas guardas evalúan verdadero (en este ejemplo, `x = y`), entonces se hace una elección *arbitraria*. El programador no puede de-

terminar qué recorrido se tomará; la construcción es **no determinista**. Un programa bien construido será válido para todas las elecciones posibles. Cuando `x = y` en este ejemplo, ambos recorridos tendrán el mismo efecto.

Es importante indicar que esta estructura no determinista es bastante distinta de la forma determinista que podría haber sido construida utilizando la sentencia normal `if`:

```
if x <= y then m := x;
elsif x >= y then m := y;
end if;
```

Aquí, los valores `x = y` garantizarían que a `m` se le asigna el valor `x`.

La construcción de elección general puede tener cualquier número de componentes con guarda. Si más de una guarda evalúa a VERDADERO, la elección es arbitraria. Pero si ninguna guarda evalúa a VERDADERO, entoces esto se considera un error, y se aborta la sentencia dentro del proceso que la ejecutó.

Las operaciones con guardas son una estructura general de programa. Sin embargo, si la operación que está siendo guardada es un operador de mensaje (normalmente el operador recibe, aunque en algunos lenguajes también puede ser envía), entonces la sentencia se conoce como **espera selectiva**. Esto fue introducido inicialmente en CSP (Hoare, 1978), y está disponible en Ada y en occam2.

9.4.1 El ALT de occam2

Consideremos un proceso que lee enteros en tres canales (`ch1`, `ch2` y `ch3`), y después devuelve los enteros que recibe en otro canal (`chout`). Si los enteros llegaran en secuencia a los tres canales, entonces bastaría una construcción de bucle sencillo.

```
WHILE TRUE
  SEQ
    ch1 ? I              -- para algún entero local I
    chout ! I
    ch2 ? I
    chout ! I
    ch3 ? I
    chout ! I
```

Sin embargo, si se desconoce el orden de llegada, entonces, cada vez que el proceso realiza un bucle, deberá hacer una elección entre tres alternativas:

```
WHILE TRUE
  ALT
    ch1 ? I
      chout ! I
```

```
    ch2 ? I
      chout ! I
    ch3 ? I
      chout ! I
```

Si hay un entero en `ch1`, `ch2` o `ch3`, será leído, y se tomará la acción especificada, que en este caso es siempre pasar el entero adquirido recientemente al canal de salida `chout`. En una situación en la que más de un canal de entrada está dispuesto para comunicarse, se hace una elección arbitraria en el momento en el que se lee cada uno. Antes de considerar el comportamiento de la sentencia `ALT` cuando ninguno de los canales está dispuesto, se esbozará la estructura de una sentencia `ALT`. Consta de un conjunto de procesos con guarda:

```
ALT
  G1
    P1
  G2
    P2
  G3
    P3
    :
  Gn
    Pn
```

Los procesos, en sí mismos, no están restringidos (puede ser cualquier proceso occam2). Las guardas (que también son procesos) pueden tener alguna de estas tres formas (la cuarta posibilidad, que implica la especificación de un retardo de tiempo, se considerará enel Capítulo 12).

```
<expresión_booleana> & operación_del _canal_de entrada

operación_del_canal_de entrada


<expresión_booleana> & SKIP
```

La forma más general es, por tanto, una expresión booleana y un canal de lectura; por ejemplo:

```
NOT BuferLleno & ch ? BUFER[TOPE]
```

Si la expresión booleana es sencillamente `TRUE`, entonces puede omitirse completamente (como en el ejemplo enterior). La forma de guarda `SKIP` se utiliza para especificar alguna acción alternativa para ser tomada cuando se evitan otras alternativas; por ejemplo:

```
ALT
  NOT BuferLleno & ch ? BUFER[TOPE]
    SEQ
```

```
      TOPE := ...
  BuferLleno & SKIP
    SEQ
      -- intercambiar búferes
```

Al evaluar la sentencia ALT, se evalúan las expresiones booleanas. Si ninguna resulta TRUE (y no hay alternativas TRUE por defecto), entonces el proceso ALT no puede continuar, y se convierte en el proceso STOP (error). Suponiendo una ejecución correcta de ALT, se examinan los canales para ver si hay procesos esperando escribir en ellos. A continuación, hay varias posibilidades:

(1) Hay una sola alternativa dispuesta, que es una expresión booleana que evalúa a verdadero (con un proceso esperando para escribir, o una guarda SKIP): se elige esta alternativa, tiene lugar la cita (si no es un SKIP), y se el ejecuta el proceso asociado.

(2) Hay más de una alternativa dispuesta: se elige una arbitrariamente, la cual podría ser el SKIP alternativo, si está presente y dispuesto.

(3) No hay alternativas dispuestas: el ALT se suspende hasta que algún otro proceso escribe en uno de los canales abiertos del ALT.

El ALT se converitirá, por tanto, en un proceso STOP, si todas las expresiones booleanas evalúan a FALSO, pero sólo se suspende si no hay llamadas pendientes. Debido al modelo de variable no compartida de occam2, no es posible para ningún otro proceso cambiar el valor de ningún componente de la expresión booleana.

El ALT, cuando se combina con SEQ, IF, WHILE, CASE y PAR, proporciona el conjunto completo de construcciones de programa de occam2. Puede adjuntarse un replicador a un ALT de la misma forma que ocurre con otras construcciones. Por ejemplo, considere un proceso concentrador que puede leer de 20 procesos (en lugar de 3 como anteriormente); sin embargo, en lugar de usar 20 canales distintos, el proceso servidor utiliza un array de canales como sigue:

```
WHILE TRUE
  ALT j = 0 FOR 20
    ch[j] ? I
      chout ! I
```

Finalmente, se debe indicar que occam2 proporciona una variante de la construcción ALT que no es arbitraria en la selección de las alternativas dispuestas. Si el programador desea dar preferencia a un canal concreto, debiera ser colocado como primer componente de un PRI ALT. La semántica del PRI ALT dicta que se elija la primera alternativa dispuesta textualmente. Los siguiente son ejemplos de sentencias PRI ALT.

```
PRI ALT
  CanalMuyImportante ? mensaje
    -- acción
  CanalImportante ? mensaje
```

```
    -- acción
  CanalMenosImportante ? mensaje
    -- acción

WHILE TRUE
  PRI ALT j = 0 FOR 20 -- se da a ch[0] la mayor prioridad
    ch[j] ? I
      chout ! I
```

En la Sección 11.3.1, se da un ejemplo del uso de `PRI ALT`.

El búfer limitado

Occam2 poporciona primitivas de comunicación de variables no compartidas, y, por tanto, los controladores de recursos, como los búferes, tienen que implementarse como procesos servidor (véase la Sección 7.2.1). Implementar un búfer con un único lector y un único escritor, precisa el uso de dos canales, que enlazan el proceso búfer a los procesos cliente (atender a más lectores y escritores requeriría arrays de canales):

```
CHAN OF Datos Toma, Agrega:
```

Desfortunadamente, la forma natural para este búfer debiera ser:

```
VAL INT Talla IS 32:
INT Tope, Base, NumeroEnBufer:
[Talla]Datos Bufer:
SEQ
  NumeroEnBufer := 0
  Tope := 0
  Base := 0
  WHILE TRUE
  ALT
    NumeroEnBufer < Talla & Agrega ? Bufer[Tope]
      SEQ
        NumeroEnBufer := NumeroEnBufer + 1
        Tope := (Tope + 1) REM Talla
    NumeroEnBufer > 0 & Toma ! Bufer[Base] -- no legal en occam
      SEQ
        NumeroEnBufer := NumeroEnBufer - 1
        Base := (Base + 1) REM Talla
```

En occam2, no se permiten operaciones de salida en este contexto. Las operaciones de entrada sólo pueden formar parte de una guarda `ALT`. La razón de esta restricción es la eficiencia de implementación en un sistema distribuido. La esencia del problema es que la provisión de guardas

simétricas podría conducir a que un canal fuera accedido por un ALT en ambos extremos. La decisión arbitraria de un ALT sería dependiente, por tanto, de la decisión del otro (y viceversa). Si los ALT están en diferentes procesadores, entonces el acuerdo en una decisión colectiva debiera implicar el paso de un número de mensajes de protocolo de bajo nivel.

Para evitar la restricción de guardas, occam2 fuerza a que la operación Toma se programe como una doble interacción. Inicialmente, el proceso cliente debe indicar que desea TOMAR, y entonces debe Tomar: se necesita, por tanto, un tercer canal:

```
CHAN OF DATOS Toma, Agrega, Solicita:
```

El cliente debe hacer las siguientes llamadas:

```
SEQ
  Solicita ! ANY          -- ANY es un token arbitrario
  Toma ? D                -- D es de tipo DATOS
```

El proceso búfer tiene la forma:

```
VAL INT Talla IS 32:
INT Tope, Base, NumeroEnBufer:
[Longitud]Datos Bufer:
SEQ
  NumeroEnBufer := 0
  Tope := 0
  Base := 0
  Datos ANY:
  WHILE TRUE
    ALT
      NumeroEnBufer < Talla & Append ? Bufer[Tope]
        SEQ
          NumeroEnBufer := NumeroEnBufer + 1
          Tope := (Tope + 1) REM Talla
      NumeroEnBufer > 0 & Request ? ANY
        SEQ
          Toma ! Bufer[Base]
          NumeroEnBufer := NumeroEnBufer - 1
          Base := (Base + 1) REM Talla
```

El funcionamiento correcto del búfer depende, por tanto, del uso correcto por parte de los procesos cliente. Esta dependencia es un ejemplo de modularidad pobre. Aunque el select de Ada es también asimétrico (es decir, no se puede seleccionar entre accepts y llamadas de entrada), el hecho de que los datos puedan pasar en dirección opuesta a la llamada elimina la dificultad que manifiesta en occam2 por sí mismo.

9.4.2 La sentencia `select` de Ada

La relación de paso de mensajes de muchos-a-uno de Ada puede tratar fácilmente con varios clientes sobre la misma entrada. Sin embargo, si un servidor debe tratar con las posible llamadas a dos o más entradas diferentes, se requiere de nuevo una estructura de tipo ALT; en Ada, esto se denomina **sentencia `select`**. Consideremos, para ilustrarlo, una tarea servidor que ofrece dos servicios, a través de las entradas `S1` y `S2`. La siguiente estructura suele ser suficiente (es decir, un bucle que contiene una sentencia `select` que ofrece ambos servicios):

```
task Servidor is
  entry S1(...);
  entry S2(...);
end Servidor;

task body Servidor is
  ...
begin
  loop
    -- prepara para el servicio
    select
      accept S1(...) do
        -- código para este servicio
      end S1;
    or
      accept S2(...) do
        -- código para este servicio
      end S2;
    end select;
    -- realiza cualquier gestión
  end loop;
end Servidor;
```

En cada ejecución del bucle, se ejecutará una de las sentencias `accept`.

Como en el primer ejemplo de occam2, este programa Ada no ilustra el uso de expresiones booleanas en guardas. La forma general del `select` Ada es:

```
select
  when <Expresion-Booleana> =>
    accept <entry> do
      ..
    end <entry>;
    -- cualquier secuencia de sentencias
or
```

```
   -- similar
   ...
end select;
```

Puede haber cualquier número de alternativas. Aparte de las alternativas accept (de las que debe haber al menos una), hay otros tres tipos (que no pueden mezclarse en la misma sentencia):

(1) Una alternativa terminate.

(2) Una alternativa else.

(3) Una alternativa delay.

En el Capítulo 12 se tratará de la alternativa delay, junto a la equivalente en occam2. La alternativa else es definida para ser ejecutada cuando (y sólo cuando) ninguna otra alternativa es ejecutable *inmediatamente*. Esto sólo puede ocurrir cuando no haya llamadas pendientes en entradas que tengan expresiones booleanas que evalúen a `True` (o no haya expresiones booleanas).

La alternativa terminate no tiene equivalente en occam2, pero es una primitiva importante. Tiene las siguientes propiedades:

(1) Si es elegida, la tarea que ejecuta el select es finalizada y terminada.

(2) Sólo puede ser elegida si no hay más tareas que puedan llamar a select.

En concreto, una tarea terminará si todas las tareas dependientes del mismo maestro ya han acabado o están igualmente esperando en sentencias select con alternativas de terminación El efecto de esta alternativa es permitir construir las tareas servidor de forma que no deban preocuparse por su terminación, sino que terminarán cuando lo precisen. La carencia de este elemento en occam2 conduce a condiciones de terminación complejas con problemas de interbloqueo asociados (Burns, 1988).

La ejecución del select sigue una secuencia similar a la del `ALT`. Primero se evalúan las expresiones boleanas, y aquéllas que produzcan un valor falso llevarán a una alternativa que será cerrada por la ejecución del select. Siguiendo esta fase, se deriva un conjunto de alternativas posibles. Si este conjunto está vacío, se genera la excepción `Program_Error` justo tras el select. En la ejecución normal, se elige una alternativa. La elección es arbitraria si hay más de una alternativa con una llamada pendiente. Si no hay llamadas pendientes en las alternativas elegibles pueden ocurrir tres cosas:

- Se ejecuta la alternativa, si existe alguna.
- Se suspende la tarea, esperando que se haga una llamada o que expire un límite de tiempo (véase el Capítulo 12).
- Se temina la tarea si hay una opción terminate y no hay otras tareas que puedan llamarla (como se ha descrito anteriormente).

Hay que tener en cuenta que las variables compartidas pueden estar contenidas en guardas, aunque esto no es recomendable, ya que un cambio en ellas no sería advertido hasta que se reeva-

luara la guarda. También, el Anexo de Sistemas de Tiempo Real permite dar prioridad a las ramas del select de acuerdo con la ordenación textual. Por tanto, esto da la misma funcionalidad que PRI ALT de occam2. Como ejemplo final de la sentencia select de Ada, considérese el cuerpo de la tarea Operador_Telefono de la Sección 9.3.2.

```
task body Operador_Telefono is
   Trabajadores : constant Integer := 10;
   Fallado : Numero;
   task type Trabajador_Reparacion;
   Equipo_Trabajo : array (1 .. Trabajadores) of Trabajador_Reparacion;
   task body Trabajador_Reparacion is ...;
   ...
begin
   loop
      -- preparar para aceptar la siguiente solicitud
      select
         accept Consulta_Directorio(Persona : in Nombre;
                Dir : in Direccion; Num : out Numero) do
            -- busca un número de teléfono y
            -- asigna el valor a Num
         end Consulta_Directorio;
      or
         accept Consulta_Directorio(Persona : in Nombre;
                CP : in Codigo_Postal; Num : out Numero) do
            -- busca un número de teléfono y
            -- asigna el valor a Num
         end Consulta_Directorio;
      or
         accept Informe_Fallo(Num : Numero) do
            Fallado := Numero;
         end Informe_Fallo;
         -- almacena el número fallado
      or
         when Fallo_No_Localizado =>
            accept Asigna_Trabajador_Reparacion(Num : out Numero) do
               -- consigue el siguiente número con fallo
               Num := ...;
            end Asigna_Trabajador_Reparacion;
            -- actualiza el informe de números no localizados con fallo
      or
         terminate;
      end select;
```

```
      ...
   end loop;
end Operador_Telefono;
```

Un tipo de tarea local, `Trabajador_Reparacion`, es responsable de la reparación de los fallos en la línea cuando se informa de ellos. Aquél comunica con el `Operador_Telefono` a través de la entrada privada `Asignar_Trabajador_Reparacion`. Para garantizar que las tareas trabajador no comunican continuamente con el `Operador_Telefono`, se guarda la sentencia aceptada. Hay que considerar también que el `Operador_Telefono` realiza tanto trabajo como es posible fuera de la cita para permitir que las tareas cliente avancen tan rápido como sea posible.

En Ada, un cliente también puede utilizar una sentencia select (véase la Sección 12.4.2) y manejar eventos asíncronos (véase la Sección 10.8).

9.4.3 No determinismo, espera selectiva, y primitivas de sincronización

En la discusión anterior, se ha indicado que cuando hay más de una alternatitiva dispuesta en una construcción de espera selectiva, la elección entre ellas es arbitraria. La razón por la que se hace arbitrario el select es que los lenguajes concurrentes hacen normalmente pocas suposiciones sobre el orden en el que se ejecutan los procesos. Se supone que el planificador planifica los procesos de forma no determinista. (Aunque algunos planificadores tienen un comportamiento determinista.)

Para ilustrar esta relación, consideremos un proceso *P* que ejecutará una construcción wait selectiva en la que se podría llamar a los procesos *S* y *T*. Si se supone que el comportamiento del planificador es no determinista, entonces hay un número de posibles entrelazamientos, o «trazas», para este programa:

(1) Primero se ejecuta *P*, y se bloquea en el select. *S* (o *T*) ejecutan después la cita con *P*.

(2) *S* (o *T*) se ejecutan primero y se bloquean en la llamada a *P*. Después, actúa *P* y ejecuta la sentencia select, con el resultado de que se produce una cita con *S* (o *T*).

(3) *S* (o *T*) se ejecutan primero y se bloquean en la llamada a *P*. *T* (o *S*) se ejecutan después, y también se bloquean en P. Finalmente se ejecuta P y se realiza la sentencia select, donde esperan T y S.

Hay tres entrelazados posibles y legales que llevan a *P* a tener ninguna, una, o dos llamadas pendientes en la espera selectiva. El select se define arbitrariamente precisamente porque se supone que el planificador es no determinista. Si *P*, *S* y *T* pueden ejecutarse en cualquier orden, entonces, en el caso (3), *P* debiera ser capaz de elegir la cita con *S* o *T*, lo que no afectará a la corrección del programa.

Un argumento similar se aplica a cualquier cola que define una primitiva de sincronización. La planificación no determinista implica que todas las colas de ese tipo deberían liberar procesos en un orden no determinista. Aunque las colas de semáforos se suelen definir de esta forma, las

colas de entrada y las de monitor suelen ser FIFO. La razón que se da, en este caso, es que las colas FIFO previenen la inanición. Este argumento es, sin embargo, falaz; si el planificador es no determinista, a la postre podrá producirse inanición (puede que a un proceso no se le asigne nunca un procesador). No es apropiado para la primitiva de sincronización intentar prevenir la inanición. Es discutible que también las colas de entrada deban ser no deterministas.

La planificación de procesos con prioridades asignadas se considera con detalle en el Capítulo 13.

9.5 Mensajes POSIX

POSIX soporta paso de mensajes asíncrono indirecto mediante la noción de colas de mensajes. Una cola de mensajes puede tener muchos lectores y muchos escritores. Se puede asociar prioridad a cada mensaje.

Las colas de mensajes están dirigidas, en realidad, a la comunicacón entre procesos (potencialmente distribuidos). Sin embargo, no hay nada que impida su uso entre hilos en el mismo proceso, aunque es más eficiente usar memoria compartida y mutexes para este propósito (véase la Sección 8.6.3).

Todas las colas de mensajes tienen atributos que indican la talla máxima de la cola, la longitud máxima de cada mensaje de la cola, el número de mensajes actualmente encolados, etc. Se emplea un objeto atributo para establecer los atributos de la cola en su creación. Los atributos de una cola se manipulan mediante las funciones `mq_getattr` y `mq_setattr` (estas funciones manipulan los atributos ellas mismas, y no un objeto atributo, lo cual difiere de lo que ocurre con los atributos de hilo y de mutex).

Al crear colas de mensajes se les da un nombre, que es semejante a un nombre de archivo, pero que no necesariamente se refleja en el sistema de archivos. Conseguir acceder a la cola sólo precisa que el proceso usuario efectúe `mq_open` con el nombre asociado. Como en todos los sistemas de ficheros tipo UNIX, `mq_open` se utiliza tanto para crear como para abrir una cola ya existente. También se dispone de las correspondientes rutinas `mq_close` y `mq_unlink`.

El envío y la recepción de mensajes se realiza mediente las rutinas `mq_send` y `mq_receive`. Los datos son leídos/escritos de/hacia un búfer de caracteres. Si el búfer está vacío o lleno, el proceso emisor/receptor se bloquea, a menos que se haya activado el atributo `O_NONBLOCK` para la cola. En éste último caso se devolverá un error. Si al desbloquearse una cola hay emisores y receptores esperando, no se especifica cuál es despertado, a menos que se especifique la opción de planificación por prioridad. Si el proceso tiene múltiples hilos, cada hilo se considera un potencial emisor/receptor en sí mismo.

Un proceso puede indicar también que se le envíe una señal (véase la Sección 10.6) cuando una cola vacía reciba mensajes y no hay receptores esperando. Así, un proceso puede permitirse

Programa 9.1. Una interfaz C para las colas de mensajes POSIX.

```
typedef ... mqd_t;
typedef ... mode_t;
typedef ... size_t;
typedef ... ssize_t;

struct mq_attr {
  ...
  long mq_flags;
  long mq_maxmsg;
  long mq_msgsize;
  long mq_curmsg;
  ...
};
#define O_CREAT ...
#define O_EXCL ...
#define O_RDONLY ...

int mq_getattr(mqd_t mq, struct mq_attr *attrbuf);
  /* obtiene los atributos actuales asociados con mq */
int mq_setattr(mqd_t mq, const struct mq_attr *new_attrs,
               struct mq_attr *old_attrs);
  /* activa los atributos actuales asociados con mq */

mqd_t mq_open(const char *mq_name, int oflags, mode_t mode,
              struct mq_attr *mq_attr);
  /* abre/crea la cola de mensajes nombrada */

int mq_close(mqd_t mq);
  /* cierra la cola de mensajes */

int mq_unlink(const char *mq_name);

ssize_t mq_receive(mqd_t mq, char *msq_buffer,
               size_t buflen, unsigned int *msgprio);
  /* consigue el siguiente mensaje de la cola y lo almacena en el */
  /* area apuntada por msq_buffer; */
  /* se devuelve el tamaño actual del mensaje */
ssize_t mq_receive(mqd_t mq, char *msq_buffer,
               size_t buflen, unsigned int *msgprio,
               const struct timespec *abs_timeout);
  /* como para mq_receive pero con un timeout */
  /* devuelve ETIMEDOUT si expira el timeout */
```

```
int mq_send(mqd_t mq, const char *msq, "
                size_t msglen, unsigned int msgprio);
  /* envía el mensaje apuntado por msq */

int mq_timedsend(mqd_t mq, const char *msq,
                size_t msglen, unsigned int msgprio,
                const struct timespec *abs_timeout);
  /* envía el mensaje apuntado por msq con un timeout */
  /* devuelve ETIMEDOUT si expira el timeout */

int mq_notify(mqd_t mq, const struct sigevent *notification);
  /* solicita que se envíe una señal al proceso llamador */
  /* si llega un mensaje en una mq vacía y no hay */
  /* receptores esperando */

/* Todas las funciones enteras anteriores devuleven 0 si tienen éxito, */
/* si no, -1. */
/* Cuando se devuelve una condición de error en alguna de las */
/* funciones anteriores, la variable compartida errno contiene */
/* la razón del error */
```

no esperar por nuevos mensajes que lleguen a una o más colas de mensajes. Esto permite implementar una espera selectiva equivalente.

El Programa 9.1 resume una interfaz C típica para la funcionalidad de paso de mensajes POSIX.

Para ilustrar el uso de las colas de mensajes, se esboza el sencillo brazo de robot discutido en los Capítulos 7 y 8, con un proceso padre y tres `controladores` hijos. En este caso, el padre comunica con los controladores y les pasa la nueva posición del brazo. Primero se proporciona el código del `controlador`. Aquí, se supone que `MQ_OPEN` y `FORK` son funciones que comprueban el error devuelto por las llamadas al sistema `mq_open` y `fork`, y sólo retornan si éstas tienen éxito.

```
typedef enum {xplano, yplano, zplano} dimension;

void mover_brazo(dimension D, int P);

#define DEFAULT_NBYTES 4
/* se supone que las coordenadas pueden representarse con 4 bytes */
int nbytes = DEFAULT_NBYTES;

#define MQ_XPLANO "/mq_xplano" /* nombre de cola de mensajes */
#define MQ_YPLANO "/mq_yplano" /* nombre de cola de mensajes */
```

```
#define MQ_ZPLANO "/mq_zplano"  /* nombre de cola de mensajes */
#define MODE ...                /* modo de información para mq_open */
/* nombres de las colas de mensajes */

void controlador(dimension dim)
{
  int posicion, indicacion;
  mqd_t mi_cola;
  struct mq_attr ma;
  char buf[DEFAULT_NBYTES];
  ssize_t talla;

  posicion = 0;
  switch(dim) { /* abre la cola de mensajes apropiada */
     case xplano:
        mi_cola = MQ_OPEN(MQ_XPLANO, O_RDONLY, MODE, &ma);
        break;
     case yplano:
        mi_cola = MQ_OPEN(MQ_YPLANO, O_RDONLY, MODE, &ma);
        break;
     case zplano:
        mi_cola = MQ_OPEN(MQ_ZPLANO, O_RDONLY, MODE, &ma);
        break;
     default:
        return;
     };

  while (1) {
     /* lee el mensaje */
     talla = MQ_RECEIVE(mi_cola, &buf[0], nbytes, NULL);
     indicacion = *((int *) (&buf[0]));
     posicion = posicion + indicacion;
     mover_brazo(dim, posicion);
  };
}
```

Ahora, puede darse el programa principal que crea el proceso controlador y les pasa las coordenadas apropiadas:

```
int main(int argc, char **argv) {
  mqd_t mq_xplano, mq_yplano, mq_zplano;
     /* una cola por cada proceso */
  struct mq_attr ma; /* atributos de la cola */
  int xpid, ypid, zpid;
```

```
    char buf[DEFAULT_NBYTES];

    /* establece los atributos de la cola de mensajes requerida */
    ma.mq_flags = 0; /* No hay comportamiento especial */
    ma.mq_maxmsg = 1;
    ma.mq_msgsize = nbytes;

    /* llamadas para habilitar los atributos actuales de las*/
    /* tres colas de mensajes */

    mq_xplano = MQ_OPEN(MQ_XPLANO, O_CREAT|O_EXCL, MODE, &ma);
    mq_yplano = MQ_OPEN(MQ_YPLANO, O_CREAT|O_EXCL, MODE, &ma);
    mq_zplano = MQ_OPEN(MQ_ZPLANO, O_CREAT|O_EXCL, MODE, &ma);

    /* Bifurca el proceso para conseguir los tres controladores */
    switch (xpid = FORK()) {
      case 0:      /* hijo */
             controlador(xplano);
             exit(0);
      default: /* padre */
        switch (ypid = FORK()) {
          case 0:     /* hijo */
            controlador(yplano);
            exit(0);
          default: /* padre */
            switch (zpid = FORK()) {
              case 0: /* hijo */
                controlador(zplano);
                exit(0);
              default: /* padre */
                break;
        }
    }
  }

  while (1) {
    /* Lee una nueva posición y habilita el búfer para */
    /* transmitir cada coordenada a los controladores, */
    /* por ejemplo */
    MQ_SEND(mq_xplano, &buf[0], nbytes, 0);
  }
}
```

9.6 El lenguaje CHILL

En esta sección, se proporciona una breve descripción del modelo de concurrencia en CHILL. Otras características de CHILL, como el modelo de manejo de excepciones, ya han sido discutidas.

El desarrollo de CHILL sigue un recorrido parejo al de Ada. Su dominio de aplicación es, sin embargo, más restringido: los sistemas de conmutación de telecomunicación. A pesar de esta restricción, estos sistemas tienen todas las características de las aplicaciones de tiempo real generales: son grandes y complejos, tienen restricciones de tiempo especificadas y requisitos de alta fiabilidad. A comienzos de 1970, el CCITT (*Comité Consultatif International Télégraphique et Téléphonique*; Comité Consultivo Internacional Telegráfico y Telefónico) reconoció la necesidad de un lenguaje de alto nivel sencillo que hiciera que los sistemas de telecomunicaciones fueran más independientes de los fabricantes de hardware. En 1973, se decidió que ninguno de los lenguajes existentes satisfacía los requisitos, y se estableció un grupo para proporcionar una propuesta preliminar del lenguaje. Se hicieron diversas pruebas, y en otoño de 1979, tras varias iteracciones de diseño, se consensuó una propuesta final. El nombre de CHILL proviene de CCITT High Level Language.

Su modelo de concurrencia es interesante por su pragmatismo. Los procesos sólo pueden ser declarados en el nivel más exterior, y se les puede pasar parámetros cuando se inician. Consideremos el sencillo controlador del brazo robótico visto anteriormente. Un tipo de proceso `control` tiene como un parámetro la dimensión del movimiento (es decir, en el plano *x*, *y* o *z*). La acción del proceso es leer una nueva posición relativa para su plano de acción y producir el movimiento del brazo del robot:

```
newmode dimension = set(xplano, yplano, zplano);
  /* tipo enumeración */

control = process(dim dimension);
  dcl posicion, indicación int;
        /* declara variables enteras: posicion e indicación */

  posicion := 0;
  do for ever;
     nueva_indicacion(dim, indicacion);
     posicion := posicion + indicacion;
     mover_brazo(dim, posicion);
  od;
end control;
```

Para indicar la ejecución de tres procesos de control (uno por cada dimensión), se utiliza la sentencia `start`:

```
start control(xplano);
```

```
start control(yplano);
start control(zplano);
```

Esto producirá el inicio de tres procesos anónimos. Si se desea identificar cada instancia de un *process*, hay que dar nombres únicos como parte de la sentencia `start`. (Un nombre es una variable del tipo *instance*.)

```
dcl xinst, yinst, zinst instance;

start control(xplano) set xinst;
start control(yplano) set yinst;
start control(zplano) set zinst;
```

Una instancia de proceso puede terminarse a sí misma ejecutando una sentencia `stop`.

9.6.1 Comunicación en CHILL

Es en su mecanismo de comunicación y sincronización donde puede verse el pragmatismo de CHILL. Mas que tener un modelo único, soporta tres aproximaciones diferentes:

(1) `regiones`: proporcionan exclusión mutua a las variables compartidas (es decir, hay monitores).

(2) `búferes`: permiten la comunicción asíncrona entre procesos.

(3) `signals`: son una forma de canal, y pueden ser comparadas con el mecanismo de occam.

El motivo de tener tres estructuras separadas está en el reconocimiento de que no hay un único modelo universalmente aceptado como el mejor:

> Cada mecanismo de comunicación puede no ser capaz de funcionar de modo óptimo, tanto en una arquitectura distribuida como en una de memoria común. (Smedema et al., 1983)

Una región garantiza explícitamente el acceso a sus procedimientos; con la región se pueden utilizar eventos (`events`) para retrasar procesos. Sobre los eventos se puede actuar mediante `delay` y `continue`, que tienen una semántica similar a la de las operaciones wait y signal de Modula-1. Lo siguiente implementa un recurso controlador sencillo:

```
recurso : region
  grant asigna, desasigna;
  syn max = 10;                    /* declara constante */
  decl usado int := 0;
  sin_recursos event;

  asigna : proc;
     if usado < max then
        usado := usado + 1;
```

```
    else
      delay(sin_recursos);
    fi;
  end asigna;

  desasigna : proc
    usado := usado - 1;
    continue(sin_recursos)
  end desasigna;

end recurso;
```

Los búferes están predefinidos, con operadores `send` (pon) y `receive` (dame):

```
syn talla = 32;
dcl buf buffer(talla) int;      /* declara el búfer como entero */
...

  send buf(I);                  /* I es un entero */
...

  receive case
    (buf in J) :                /* sentencias */
esac
```

El `receive` se ubica dentro de una sentencia `case`, para que, en general, un proceso pueda intentar leer de más de un búfer. Si todos los búferes están vacíos, el proceso se demorará. También se puede obtener el nombre del proceso que ubicó cierto dato en un búfer, lo cual es especialmente útil en un sistema de comunicaciones, donde los enlaces se establecen entre procesos productores y consumidores.

Las señales se definen en relación con el tipo de dato que comunican y el tipo del proceso destino:

```
signal channel = (int) to consumer;
```

Aquí, `consumer` es un tipo de proceso. Para transmitir datos (`I`) mediante esta señal, la sentencia `send` se emplea de nuevo, así:

```
send channel(I) to con;
```

donde `con` es una instancia de proceso de tipo consumer. La operación receive utiliza la sentencia case, proporcionando así la espera selectiva (sin guardas).

Es interesante ver la convención de nombrado aplicada a las señales en CHILL. La operación send nombra a la señal y la instancia del proceso; receive sólo menciona la señal, aunque puede informarse de la identidad de los procesos asociados.

9.7 Llamadas a métodos remotos

Este capítulo se ha centrado en cómo pueden los procesos comunicar y sincronizar sus actividades. Cualquier discusión sobre las implicaciones que esto tiene cuando los procesos residen en máquinas diferentes conectadas en red, se pospone hasta en Capítulo 14. En este punto, y con el fin de completar estos conceptos, se presenta el concepto de **llamada a método remoto** (RPC; remote procedure call) como mecanismo habitual de tranferencia de control en un entorno distribuido.

En un único procesador, los procesos ejecutan procedimientos para transferir el control de una sección de código a otra, y sólo cuando necesitan comunicarse y sincronizarse con otro proceso precisan de la comunicación entre procesos. Las llamadas a procedimientos remotos son una extensión que permite a un proceso ejecutar código que reside en más de una máquina. Permiten a un proceso que se esté ejecutando concurrentemente en un procesador, ejecutar un procedimiento de otro; que esto deba o no hacerse de modo transparente para el programador de aplicaciones, es un tema discutible que se deja para Capítulo 14. La ejecución de dicho procedimiento puede implicar comunicación con los procesos que residen en la máquina remota; esto se obtiene bien con métodos de variables compartidas (por ejemplo monitores), bien por paso de mensajes (por ejemplo cita).

Merece la pena señalar que el paso de mensajes de invocación remota resulta ser similar, sintácticamente, a una llamada a procedimiento. Aquí el mensaje se codifica en los parámetros de entrada, y la respuesta se codifica en los parámetros de salida (véase, por ejemplo, entry y accept de Ada). Sin embargo, esta convención sintáctica puede ser confusa, puesto que la semántica de paso de mensajes de invocación remota es completamente diferente de la de una llamada a procedimiento. En concreto, la invocación remota confía en una cooperación activa del proceso receptor, ejecutando una operación receive explícita. Una llamada a procedimiento, por el contrario, no es una forma de comunicación entre procesos, sino una transferencia de control a una parte pasiva de código. Cuando el procedimiento es local (en la misma máquina del llamador), su cuerpo puede ser ejecutado por el proceso invocador; en el caso de una llamada a procedimiento remoto, el cuerpo podrá ejecutarse en nombre del llamador a través un proceso anónimo, subordinado, en la máquina remota. La implicación de otro proceso en este caso es una cuestión de implementación, no de semántica. Para que una llamada a procedimiento remoto tenga la misma semántica (reentrante) que una local, la implementación debe permitir mantener concurrentemente un número arbitrario de llamadas. Esto requiere la creación de un proceso/hilo por cada llamada, o la existencia de un conjunto de procesos/hilos lo suficientemente grande como para mantener el número máximo de llamadas concurrentes. El coste de creación o mantenimiento de procesos puede a veces hacer que el grado de concurrencia sea limitada.

Resumen

Ada y POSIX proporcionan soporte para paso de mensajes; también permiten paradigmas de comunicación alternativos. En occam2, sin embargo, el único mecanismo de comunicación entre procesos es el paso de mensajes.

La semántica de la comunicación basada en mensajes se define sobre tres cuestiones:

- El modelo de sincronización.
- El método de nombrado de procesos.
- La estructura del mensaje.

De la semántica de la operación «envía» surgen variaciones en el modelo de sincronización de procesos. Existen tres clases generales:

- *Asíncrono*: el proceso emisor no espera.
- Síncrono: el proceso emisor espera a que sea leido el mensaje.
- Invocación remota: el proceso emisor espera a la lectura del mensaje, la realización de sus tareas, y la generación de una contestación.

Se puede asimilar, sintácticamente, la invocación remota a una llamada a procedimiento, aunque se puede producir confusión cuando se utilizan llamadas a procedimientos remotos (RPC) en un sistema distribuido. RPC es, sin embargo, una estrategia de implementación; la invocación remota define la semántica de modelo de paso de mensajes. Los dos procesos implicados en esta comunicación pueden estar en el mismo procesador, o pueden estar distribuidos; las semánticas son las mismas.

El nombrado de procesos implica dos características diferenciadas: directo (o indirecto) y simetría. Ada utiliza invocación remota con nombrado asimétrico directo. Occam2, sin embargo, tiene un esquema simético síncrono directo (los mensajes pueden tomar la forma de cualquier sistema o tipo definido por el usuario). POSIX soporta un esquema simétrico asíncrono.

Para que dos procesos se comuniquen en occam2, es preciso definir un canal (del protocolo tipo apropiado) y emplear dos operadores de canal: `?` y `!`. La comunicación en Ada requiere que una tarea defina una entrada y, después, dentro de su cuerpo, acepte cualquier llamada entrante. Una cita es el producto de la aceptación de una llamada sobre la entrada de una tarea. En POSIX, la comunicación se realiza mediente colas de mensajes con primitivas envía/recibe. Occam2 soporta un mecanismo de comunicación uno a uno, Ada uno de muchos a uno, y POSIX uno de muchos a muchos.

Con el fin de aumentar la potencia expresiva de los lenguajes de programación concurrente basados en mensajes, es necesario permitir que un proceso elija entre comunicaciones alternativas. A la primitiva del lenguaje que soporta esta posibilidad se la conoce como *espera selectiva*. Aquí, un proceso puede elegir entre diferentes alternativas; en cualquier ejecución concreta, algunas de esas alternativas pueden quedar excluidas utilizando una guarda booleana. La construcción de occam2 `ALT` permite que un proceso elija entre cierto número de operaciones recibe guardadas. Ada soporta una característica del lenguaje similar (la sentencia select). Sin embargo, tiene dos funcionalidades extra:

(1) Una sentencia select puede tener una parte alternativa por defecto, que se ejecutaría de no haber llamadas pendientes sobre las alternativas abiertas.

(2) Una sentencia select puede tener una alternativa «terminate», que producirá la terminación de la tarea si ninguna de las otras tareas que lo pudieran llamar siguen siendo ejecutables.

En POSIX, un proceso o hilo puede indicar que no está preparado para bloquear cuando la cola de mensajes está llena o vacía. Se utiliza un mecanismo de comunicación para permitir que el sistema operativo envíe una señal cuando un proceso puede proceder. Puede usarse este mecanismo para esperar un mensaje en una o más colas de mensajes.

Una característica importante de la cita extendida de Ada es su interacción con el modelo de manejo de excepciones. Si se habilita una excepción pero no se maneja durante una cita, se propaga tanto hacia la tarea que llama como hacia la llamada. Una tarea invocadora debe, por tanto, protegerse a sí misma contra las posibles excepciones anónimas que se generen como resultado de hacer una llamada mediante paso de mensajes.

Con el fin de dar un ejemplo adicional de un diseño de lenguaje, se ha proporcionado una visión general de CHILL. CHILL tiene un diseño inusualmente pragmático, incorporando:

- Inicialización de datos para procesos.
- Nombres de proceso.
- Monitores (regiones con nombre).
- Búferes (que permiten comunicación asíncrona).
- Señales (como mecanismo de canal síncrono).

Lecturas complementarias

Andrews, G. A. (1991), *Concurrent Programming Principles and Practice*, Redwood City, CA: Benjamin/Cummings.

Burns, A. (1988), *Programming in occam2*, Reading, Addison-Wesley.

Burns, A., y Davies, G. (1993), *Concurrent Programming*, Reading: Addison-Wesley.

Burns, A., y Wellings, A. J. (1995), *Concurrency in Ada*, Cambridge: Cambridge University Press.

Hoare, C. A. R. (1985), *Communicating Sequential Processes*, Englewood Cliffs, NJ: Prentice Hall.

Galletly, J. (1990), *Occam2*, London: Pitman.

Silberschatz, A., y Galvin, P. A. (1998), *Operating System Concepts*, New York: John Wiley & Sons.

Ejercicios

9.1 Puesto que Ada soporta objetos protegidos, ¿deben ser eliminadas del lenguaje las funcionalidades de cita?

9.2 Si una tarea Ada tiene dos puntos de entrada, ¿es posible poder aceptar la entrada de la tarea invocadora que haya estado esperando durante más tiempo?

9.3 Muestre cómo implementar un semáforo binario utilizando una tarea Ada y una cita. ¿Qué ocurre si una tarea que ha ejecutado un `Wait` es abortada antes de que pueda ejecutar un `Signal`?

9.4 Discuta las ventajas y desventajas de implementar semáforos mediante una cita en lugar de con un objeto protegido.

9.5 Muestre cómo se pueden utilizar una tarea Ada y citas para implementar los monitores de Hoare.

9.6 La Figura 9.1 representa un proceso en occam2.

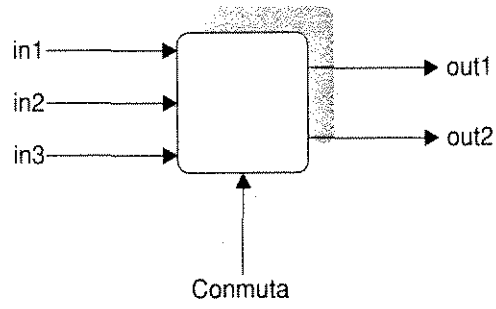


Figura 9.1. Un proceso en occam2.

Se leen enteros en los canales in1, in2, in3. El proceso toma esos enteros como llegan. Inicialmente, todos los enteros de entrada se envían por el canal out1. Si llega cualquier entrada por el canal «conmuta», se cambia la salida hacia el canal out2. Las entradas sucesivas por el canal «conmuta» intercambian el canal de salida. Escriba un PROC occam2 que implemente este proceso

9.7 ¿Hasta qué punto puede considerase que una cita Ada equivale a una llamada a un procedimiento remoto? Discuta el código que viene a continuación, que pretende implementar un procedimiento remoto concreto. Considere sus implicaciones tanto para el procedimiento invocado, como para el invocador.

```
task type Implementacion_Rpc is
   entry Rpc(Param1:Tipo1; Param2:Tipo2);•EJERCICIOS 327
end Implementacion_Rpc;

task body Implementación_Rpc is
```

```
begin
   accept Rpc(Param1:Tipo1; Param2:Tipo2) do
   -- cuerpo del procedimiento
   end Rpc;
end Implementacion_Rpc;

-- declara un array de 1000 tareas de tipo Implementacion_Rpc
Rpc_Concurrentes : array(1 .. 1000) of Implementación_Rpc;
```

9.8 Complete el Ejercicio 7.8 utilizando, para la sincronización de procesos: (a) tareas Ada y la cita, y (b) el canal occam2.

9.9 Utilizando las colas de mensajes POSIX, rehaga la respuesta al Ejercicio 9. 6.

9.10 Considere un sistema Ada de tres tareas «fumador de cigarrillos» y una tarea agente. Cada fumador hace cigarrillos continuamente y después los fuma. Para hacer un cigarrillo se necesitan tres ingredientes: tabaco, papel y cerillas. Una de las tareas tiene un stock infinito de papel, otra tiene un stock de tabaco, y la tercera tiene un stock de cerillas. La tarea agente tiene un stock infinito de los tres ingredientes. Cada fumador debe comunicarse con la tarea agente para obtener los dos ingredientes de los que no dispone.

La tarea agente tiene la siguiente especificación

```
task Agente is
   entry Proporciona_Cerillas(...);
   entry Proporciona_Papel(...);
   entry Proporciona_Tabaco(...);
   entry Cigarrillo_Acabado;
end Agente;
```

El cuerpo de la tarea agente elige dos ingredientes aleatoriamente, y después acepta en las entradas asociadas para pasar los ingredientes a los fumadores. Una vez se han pasado ambos ingredientes, espera la comunicación en la entrada `Cigarrillo_Acabado` antes de repetir el proceso indefinidamente. Cada fumador en posesión de un cigarrillo indica la finalización llamando a la entrada `Cigarrillo_Acabado`, y entonces solicita nuevos ingredientes.

Bosqueje la especificación y el cuerpo de las tareas de tres fumadores y el cuerpo de la tarea agente. Si es necesario, añada parámetros para las especificaciones de entrada de la tarea `Agente`. La solución debiera estar libre de interbloqueo.

9.11 Una tarea servidor tiene la siguiente especificación Ada:

```
task Servidor is
   entry Servicio_A;
   entry Servicio_B;
   entry Servicio_C;
end Servidor;
```

Escriba el cuerpo de la tarea `Servidor` de forma que realice las siguientes operaciones.

- Si las tareas clientes están esperando en todas las entradas, la tarea debiera seguir los siguientes en un orden cíclico; es decir, accepta primero una entrada `Servicio_A`, luego una entrada `Servicio_B`, luego una entrada `Servicio_C`, y luego una entrada `Servicio_A`, y así sucesivamente.
- Si todas las entradas tienen una tarea cliente esperando, el `Servidor` debiera servir las entradas en un orden cíclico. Las tareas `Servidor` no debieran estra bloqueadas si hay clientes esperando todavía un servicio.
- Si la tarea `Servidor` no tiene clientes esperando, entonces *no* debería haber espera ocupada: deberá esperar bloqueada que se realice una solicitud del cliente.
- Si todos los clientes posibles han terminado, el `Servidor` deberá terminar.

Suponga que las tareas cliente no son abortas y plantean sólo llamadas de entrada sencillas.

9.12 Un proceso servidor en occam2 recibe mensajes de sincronización en los siguientes canales.

```
CHAN OF INT servicioA, servicioB, servicioC:
```

Escriba el cuerpo del proceso servidor de forma que realice todas las operaciones siguientes.

- Si los procesos cliente están esperando en todos los canales, el servidor debiera servir a los clientes en un order cíclico: es decir, primero una solicitud `servicioA`, luego una solicitud `servicioB`, luego una solicitud `servicioC`, y luego una solicitud `servicioA`, y así sucesivamente.
- Si no todos los canales tienen un proceso cliente esperando, el servidor debiera servir los canales en orden cíclico. El servidor no debiera estar bloqueado si hay clientes todavía esperando un servicio.
- Si el proceso servidor no tiene clientes esperando, entonces *no* debería hacer espera ocupada; tendría que esperar bloqueado a que se realizara una solicitud del cliente.

9.13 El siguiente paquete Ada proporciona un servicio. Durante la provisión de este servicio podrán generarse las excepciones `A`, `B`, `C`, y `D`.

```
package Servidor is
   A, B, C, D : exception;
   procedure Servicio; -- puede provocar A, B, C o D
end Servidor;
```

En el siguiente procedimiento se han creado dos tareas; la tarea `Uno` realiza una cita con la tarea `Dos`. Durante la cita, la tarea `Dos` llama al `Servicio` proporcionado por el paquete servidor.

```
with Servidor; use Servidor;
with Ada.Text_Io; use Ada.Text_Io;
procedure Main is

   task Uno;

   task Dos is
      entry Sincro;
   end Dos;

   task body Uno is
   begin
      Dos.Sincro;
   exception
      when A =>
         Put_Line("A atrapada en uno ");
         raise;
      when B =>
         Put_Line("B atrapada en uno ");
         raise C;
      when C =>
         Put_Line("C atrapada en uno ");
      when D =>
         Put_Line("D atrapada en uno ");
   end;


   task body Dos is
   begin -- bloque X
      begin -- bloque Y
         begin -- bloque Z
            accept Sincro do
               begin
                  Servicio;
               exception
                  when A =>
                     Put_Line("A atrapada en sincro");
                  when B =>
                     Put_Line("B atrapada en sincro");
                     raise;
                  when C =>
                     Put_Line("C atrapada en sincro");
                     raise D;
               end;
            end Sincro;
```

```
            exception
               when A =>
                  Put_Line("A atrapada en bloque Z");
               when B =>
                  Put_Line("B atrapada en bloque Z");
                  raise C;
               when others =>
                  Put_Line("otras atrapadas en Z");
                  raise C;
            end; -- bloque Z
         exception
            when C =>
               Put_Line("C atrapada en Y");
            when others =>
               Put_Line("otras atrapadas en Y");
               raise C;
         end; -- bloque Y
      exception
         when A =>
            Put_Line("A atrapada en X");
         when others =>
            Put_Line("otras atrapadas en X");
      end; -- bloque X y tarea DOS

   begin -- procedimiento main
      null;

   exception
      when A =>
         Put_Line("A atrapada en main");
      when B =>
         Put_Line("B atrapada en main");
      when C =>
         Put_Line("C atrapada en main");
      when D =>
         Put_Line("D atrapada en main");

   end Main;
```

El procedimiento `Put_Line` viene declarado en el paquete `Text_Io`, y cuando se invoca imprime su argumento en el terminal.

Qué salida debería aparecer si el procedimiento `Servicio`

(1) Ha generado la excepción A.

(2) Ha generado la excepción B.

(3) Ha generado la excepción C.

(4) Ha generado la excepción D.

Suponga que la salida no llega a entremezclarse por las llamadas concurrentes a `Put_Line`.

9.14 Considere el siguiente fragmento de un programa occam2.

```
INT Numero.De.Procesos is 10:
-- otras declaraciones, si es necesario

PROC Controlador
  -- código para el procedimiento Controller
:

PROC Paro.Marcha(VAL INT Id)
  -- código para el procedimiento
:

PAR
  PAR i = 0 FOR Numero.De.Procesos
    SEQ
      -- código de la Parte A para el proceso usuario
      Paro.Marcha(i);
      -- código de la Parte B para el proceso usuario
    Controlador
```

Este procedimiento produce 10 procesos usuario (que ejecutan todos su código de la parte A, después llaman al procedimiento `Paro.Marcha`, y entonces ejecutan su código de la parte B) y un proceso Controlador. El objetivo de los procedimientos `Paro.Marcha` y Controlador es garantizar que *al menos* seis de los procesos usuario ejecutan su parte A antes de ejecutar su parte B.

Muestre cómo se pueden implementar los procedimientos `Paro.Marcha` y Controlador. Si es necesario, incluya declaraciones adicionales.

Capítulo 10

Acciones atómicas, procesos concurrentes y fiabilidad

En el Capítulo 5 se consideró cómo producir software fiable en presencia de una variedad de errores. Se identificaron dos técnicas esenciales para la evaluación y el confinamiento del daño, como son la descomposición modular y las acciones atómicas. Además, se introdujeron las nociones de recuperación de errores hacia adelante y hacia atrás como aproximaciones a la recuperación dinámica de errores. Se mostró que, cuando los procesos se comunican y sincronizan sus actividades, la recuperación de errores hacia atrás puede desencadenar un efecto dominó. En el Capítulo 6 se consideró el manejo de excepciones como un mecanismo que proporciona, en procesos secuenciales, tanto la recuperación de errores hacia adelante como hacia atrás. Los Capítulos 7, 8 y 9 trataron las funcionalidades que proporcionan los sistemas operativos y los lenguajes de tiempo real para la programación concurrente. Este capítulo reune el manejo de errores y la concurrencia para mostrar cómo los procesos pueden interactuar con fiabilidad en presencia de otros procesos y en presencia de fallos. Se explora en detalle la noción de acción atómica, y se presentan los conceptos de evento asíncrono y de transferencia asíncrona de control.

Procesos cooperativos y competitivos

En el Capítulo 7 se describió la interacción de procesos en relación con tres tipos de comportamiento:

- Independencia
- Cooperación
- Competencia

Los procesos independientes no se comunican o sincronizan con los demás. Por ello, si ocurre un error en un proceso, este proceso puede iniciar procedimientos de recuperación de forma aislada

respecto al resto del sistema. Los bloques de recuperación de manejo de excepciones pueden utilizarse como se describe en los Capítulos 5 y 6.

Los procesos cooperantes, por su parte, se comunican y sincronizan sus actividades regularmente para realizar alguna operación común. Si ocurre una condición de error, es preciso que todos los procesos involucrados realicen la recuperación del error. La programación de dicha recuperación de error es el tema de este capítulo.

Los procesos competitivos, aunque esencialmente independientes, se comunican y se sincronizan para obtener recursos. Un error en uno de ellos no debería afectar al resto. Desafortunadamente, no siempre es así, en particular si el error ocurre mientras al proceso se le está asignando un recurso. La asignación fiable de recursos se comentará en el Capítulo 11.

La recuperación puede involucrar al propio recurso cuando los procesos cooperantes se comunican y se sincronizan mediante recursos compartidos. Este aspecto de la asignación de recursos se comentará también en el Capítulo 11.

10.1 Acciones atómicas

Uno de los motivos principales para incluir mecanismos de concurrencia en un lenguaje es hacer posible reflejar el paralelismo del mundo real en los programas de aplicaciones. Esto permite construir programas más expresivos, con lo que los sistemas producidos son más fiables y fáciles de mantener. Sin embargo, de manera decepcionante, los procesos concurrentes crean muchos problemas nuevos que no existían en los programas secuenciales puros. Por ello, se han dedicado los últimos capítulos a comentar algunas de las soluciones a estos problemas: en particular, la comunicación y sincronización entre procesos utilizando (correctamente) variables compartidas y paso de mensajes. Esto se ha hecho de forma claramente aislada, sin hacer consideraciones sobre la forma en que grupos de procesos concurrentes deberían estructurarse para coordinar sus actividades.

Hasta el momento, la interacción entre dos procesos se ha expresado en relación con una única comunicación. En realidad no siempre ocurre así. Por ejemplo, la retirada de fondos de una cuenta bancaria puede suponer un proceso de anotación y un proceso de pago en una secuencia de comunicaciones de autenticación del ordenante, comprobación de saldo y realización del pago. Más aún, puede ser necesario que más de dos procesos interaccionen de esta manera para realizar la acción requerida. En todas estas situaciones es imprescindible que los procesos involucrados vean un estado consistente del sistema. Con procesos concurrentes de por medio es bastante fácil que los grupos de procesos interfieran entre sí.

En este caso, lo que se necesita es que cada grupo de procesos ejecute su actividad conjunta como una **acción indivisible** o **atómica**. Por supuesto, un único proceso también podría querer protegerse de interferencias de otros procesos (por ejemplo, durante la asignación de recursos). Resulta que una acción atómica puede involucrar a uno o más procesos. Las acciones atómicas también han sido denominadas interacciones *multiparte* (Evangelist et al., 1989; Yuh-Jzer y Smolka, 1996).

Existen distintas formas cuasi equivalentes de expresar las propiedades de una acción atómica (Lomet, 1977; Randell et al., 1978).

(1) Una acción es atómica si los procesos que la realizan no saben de la existencia de ningún otro proceso activo, y ningún otro proceso activo tiene constancia de las actividades de los procesos durante el tiempo que en el que están realizando la acción.

(2) Una acción es atómica si los procesos que la realizan no se comunican con otros procesos mientras está siendo realizada la acción.

(3) Una acción es atómica si los procesos que la realizan no pueden detectar ningún cambio de estado salvo aquéllos realizados por ellos mismos, y si no revelan sus cambios de estado hasta que la acción se haya completado.

(4) Las acciones son atómicas si, en lo que respecta a otros procesos, pueden ser consideradas indivisibles e instantáneas, de forma que los efectos sobre el sistema sean como si estuvieran entrelazadas y no en concurrencia.

Estas definiciones no son totalmente equivalentes. Por ejemplo, considérese la segunda: «Una acción es atómica si los procesos que la realizan se comunican sólo entre sí y no con otros procesos del sistema». Al contrario que las otras tres, ésta no define realmente la naturaleza auténtica de una acción atómica. A pesar de que garantiza que la acción es indivisible, es una restricción demasiado fuerte sobre los procesos. Se pueden permitir interacciones entre la acción atómica y el resto del sistema mientras no influyan en la actividad de la acción atómica y no se proporcione al resto del sistema ninguna información sobre el progreso de la acción (Anderson y Lee, 1990). En general, para permitir estas interacciones es necesario conocer detalladamente la función de la acción atómica y su interfaz con el resto del sistema. Como no puede existir un lenguaje que soporte de manera general esta funcionalidad, nos vemos tentados, según Anderson y Lee (1990), a adoptar la definición más restrictiva (la segunda). Sin embargo, esto sólo es posible si la implementación subyacente es capaz de adquirir todos aquellos recursos necesarios para completar la acción atómica, y no mediante instrucciones por programa. Si el programador debe adquirir y liberar los recursos, entonces los procesos de las acciones atómicas tendrán que comunicarse con los gestores de propósito general de los recursos.

Aunque una acción atómica se vea como algo indivisible, puede tener estructura interna. Para permitir la descomposición modular de las acciones atómicas se incorpora la noción de **acciones atómicas anidadas**. Los procesos comprometidos en una acción anidada deben ser un subconjunto de los que están involucrados en el nivel externo de la acción. Si no fuera así, una acción anidada podría pasar información relativa a la acción de nivel externo a un proceso externo. Entonces, la acción de nivel externo ya no sería indivisible.

10.1.1 Acciones atómicas de «dos fases»

Idealmente, todos los procesos involucrados en una acción atómica deberían obtener los recursos necesarios (mientras dure la acción) antes de comenzar. Estos recursos pueden ser liberados después de terminar la acción atómica. Si se siguen estas reglas, no existe la necesidad de que una

acción atómica interactúe con ninguna entidad externa, y se podría adoptar la definición de acción atómica más estricta.

Desafortunadamente, este ideal conlleva a una pobre utilización de recursos, por lo que se necesita una aproximación más pragmática. En primer lugar, es necesario hacer posible que comience la acción atómica sin tener aún todos los recursos. En algún punto, algún proceso de la acción pedirá que se le asigne un recurso, y para ello se comunicará con el gestor del recurso. Dicho gestor podría ser un proceso servidor. Si se sigue un esquema estricto de acción atómica, este servidor formará parte de la acción atómica, y el efecto conjunto deberá ser el de tener una versión serializada de todas las acciones que involucren al servidor. Ésta no es una buena política, por lo que, a la postre, se considera permisible que una acción atómica se comunique hacia el exterior, con los servidores de recursos.

En este contexto se entiende que un servidor de recursos es el que custodia las utilidades de sistema que no se pueden compartir libremente. Así, protegerá estas utilidades contra accesos inapropiados, aunque él mismo no realice ninguna acción sobre ellas.

La asignación de recursos mejora sustancialmente si se permite que la acción atómica libere recursos antes de finalizar. Para que esta liberación prematura tenga sentido, es preciso que el efecto global sobre el estado del recurso sea idéntico al que correspondería si se retuviera el recurso hasta el final. Sin embargo, la pronta liberación mejora la concurrencia de todo el sistema.

Si hay que retardar la obtención de los recursos y adelantar su liberación, puede darse el caso de que un cambio de estado externo esté determinado por un recurso liberado, y observado por la adquisición de un nuevo recurso. Esto estaría en contra de la definición de acción atómica. Resulta, entonces, que la única política segura de uso de recursos pasa por considerar dos fases distintas: en la primera fase «creciente», sólo se hacen peticiones de recursos; en la siguiente fase «decreciente», se van liberando recursos y no se hacen nuevas asignaciones. Con este esquema, se asegura la integridad de la acción atómica. Sin embargo, debe tenerse en cuenta que si se liberan tempranamente los recursos, la recuperación de un fallo en la acción atómica se vuelve más difícil. Lógicamente, esto se debe a que el recurso ha sido actualizado, y posiblemente otro proceso haya obsevado ya el nuevo estado del recurso. Cualquier intento de invocar la recuperación en el otro proceso provocará un efecto dominó (véase la Sección 5.5.3).

En todas las discusiones posteriores se asume que las acciones atómicas son de dos fases; si la acción ha de ser recuperable, los recursos no se liberan hasta que no haya sido completada con éxito.

10.1.2 Transacciones atómicas

La expresión **transacciones atómicas** se ha utilizado frecuentemente en el marco conceptual de los sistemas operativos y las bases de datos. Una transacción atómica tiene todas las propiedades de una acción atómica, más la característica adicional de que su ejecución puede tener éxito o fallar (no éxito). Por fallo se entiende la ocurrencia de un error del que la transacción no puede recuperarse; por ejemplo, un fallo de procesador. Si falla una acción atómica, los componentes del sistema que están siendo manipulados por la acción pueden terminar en un estado inconsistente.

Ante un fallo, una transacción atómica garantiza que los componentes son devueltos a su estado original (esto es, al estado en el que estaban *antes* de que comenzase la transacción). Las transacciones atómicas a veces se conocen como **acciones recuperables**, aunque, desafortunadamente, se tienden a confundir los términos **acción atómica** y **transacción atómica**.

Las dos propiedades distintivas de las transacciones atómicas son:

- Atomicidad de fallo, lo que significa que la transacción debe o bien ser completada con éxito, o (en el caso de fallar) no tener efecto.
- Atomicidad de sincronización (o aislamiento), lo que significa que la transacción es indivisible, en el sentido de que su ejecución parcial no puede ser observada por ninguna transacción que se esté ejecutando concurrentemente.

A pesar de que las transacciones atómicas son útiles para las aplicaciones que implican la manipulación de bases de datos, no son adecuadas para la programación de sistemas tolerantes a fallos *per se*. Esto se debe a que precisan de algún tipo de mecanismo de recuperación proporcionado por el sistema. Este mecanismo viene preestablecido, sin que el programador tenga control sobre su operativa. A pesar de que las transacciones atómicas proporcionan una forma de recuperación de errores hacia atrás, no permiten la escritura de procedimientos de recuperación. Independientemente de lo anterior, las transacciones atómicas tienen su sitio en la protección de la integridad en los sistemas de bases de datos de tiempo real.

Las transacciones atómicas vuelven a considerarse en el Capítulo 14, en el contexto de los sistemas distribuidos y de los fallos de procesador.

10.1.3 Requisitos de las acciones atómicas

Para que un lenguaje de programación de tiempo real pueda soportar acciones atómicas, debe ser posible expresar aquellos requisitos necesarios para su implementación. Estos requisitos son independientes de la noción de proceso y de la forma de comunicación entre procesos que proporcione el lenguaje, y son:

- **Límites bien definidos**

  Cada acción atómica debe tener un comienzo, un fin y una demarcación. El límite de comienzo es el lugar de cada proceso involucrado en la acción atómica donde se considera que comienza la acción. El límite de final es el lugar de cada proceso involucrado en la acción atómica donde se considera que finaliza la acción. La demarcación permite separar los procesos involucrados en la acción atómica del resto del sistema.

- **Indivisibilidad (aislamiento)**

  Una acción atómica no debe permitir el intercambio de información entre los procesos activos de dentro de la acción y los de fuera (salvo los gestores de recursos). Si dos acciones atómicas comparten datos, entonces el valor de los datos después de las acciones atómicas está determinado por la secuenciación estricta de las dos acciones en algún orden.

El comienzo de una acción atómica no implica sincronización. Los procesos pueden incluirse en momentos distintos. Sin embargo, el final de una acción atómica implica sincronización: no se permite que los procesos abandonen la acción atómica hasta que todos los procesos puedan y estén dispuestos a salir.

- **Anidamiento**

  Las acciones atómicas pueden anidadarse en tanto en cuanto no se solapen con otras acciones atómicas. Consecuentemente, y en general, sólo se permite el anidamiento estricto (es decir, si una de ellas está completamente contenida en la otra).

- **Concurrencia**

  Debería ser posible ejecutar concurrentemente distintas acciones atómicas. Una forma de asegurar la indivisibilidad es ejecutar las acciones atómicas secuencialmente. Sin embargo, esto pone en un serio compromiso el rendimiento global del sistema, y por tanto debe evitarse. De cualquier forma, el efecto global de la ejecución concurrente de varias acciones atómicas debe ser el mismo que el que se obtendría de ejecutar sus acciones en serie (serialización).

- Dado que el propósito de las acciones atómicas es que formen la base para el confinamiento de los daños, deben permitir la programación de procedimientos de recuperación.

La Figura 10.1 representa diagramáticamente los límites de una acción atómica anidada en un sistema de 6 procesos. La acción *B* comprende sólo los procesos *P3* y *P4*, mientras que la acción *A* también incluye a *P2* y *P5*. Los otros procesos (*P1* y *P6*) quedan fuera de los límites de ambas acciones atómicas.

Quizás haya que puntualizar en este momento que ciertas definiciones de acción atómica precisan de la sincronización de todos los procesos, tanto en la entrada como en la salida de la acción.

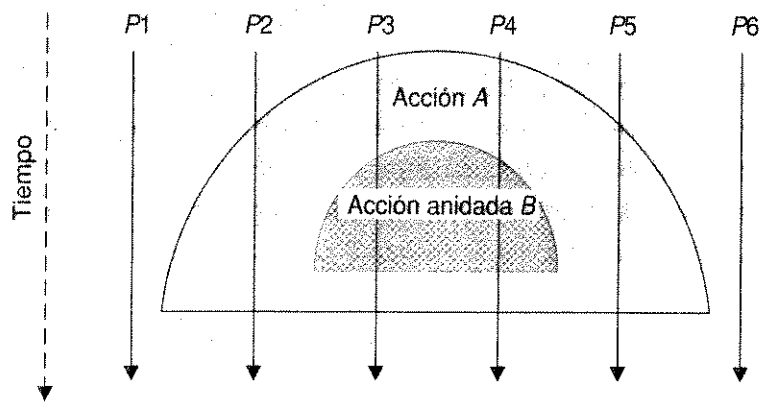


Figura 10.1. Acciones atómicas anidadas.

10.2 Acciones atómicas en lenguajes concurrentes

Las acciones atómicas proporcionan un soporte estructural para el diseño de grandes sistemas embebidos. Para beneficiarse completamente de él es necesario el apoyo del lenguaje de tiempo real. Sin embargo, ninguno de los principales lenguajes de tiempo real lo proporciona directamente. En esta sección se considera la adecuación de las primitivas de comunicación y sincronización tratadas en los Capítulos 8 y 9 para la programación de acciones atómicas. Bajo este enfoque, se da un posible marco de trabajo del lenguaje, que se amplía para tratar la recuperación de errores hacia adelante y hacia atrás.

El problema de la asignación de recursos se pospone hasta el Capítulo 11. De momento, se asume que los recursos disponen de dos modos de uso: compartible y no compartible, admitiendo ciertos recursos con ambos modos. Además, se asume que todas las acciones son de «dos fases», y que el gestor de recursos asegurará que se hace un uso apropiado de éstos. Igualmente, los procesos dentro de una acción sincronizan sus accesos al recurso para evitar cualquier interferencia.

10.2.1 Semáforos

Cuando una acción atómica implica a un solo proceso, puede implementarse por exclusión mutua mediante un semáforo binario.

```
wait(semaforo_exclusion_mutua);
  accion_atomica;
signal(semaforo_exclusion_mutua);
```

Esta solución de «semáforo» se complica, sin embargo, cuando hay más de un proceso involucrado en la acción atómica. Considere, por ejemplo, un recurso no compartible que debe ser manipulado por dos procesos. El siguiente código muestra cómo los procesos *P*1 y *P*2 pueden manipular el recurso a la vez que evitan cualquier interferencia de otros procesos. Los semáforos `inicio_accion_atomica1 y inicio_accion_atomica2` sólo permiten dos procesos en la acción. Cualquier otro proceso es bloqueado. Dos semáforos adicionales, `final_accion_atomica1 y final_accion_atomica2`, asegurarán que ninguno de los procesos abandone la acción hasta que el otro esté listo para abandonar. Hay que tener en cuenta que también se necesitan otros semáforos para controlar el acceso al recurso compartido (y para asignar el recurso en primer lugar).

```
inicio_accion_atomica1, inicio_accion_atomica2 : semaphore := 1;
final_accion_atomica1, final_accion_atomica2 : semaphore := 0;

procedure codigo_para_primer_proceso is
begin
  -- comienzo de acción atómica
  wait(inicio_accion_atomica1);
```

```
      -- obtener recurso en modo no compartible
      -- actualizar recurso

      -- señal al segundo proceso para que
      -- acceda al recurso

      -- procesamiento final

  wait(final_accion_atomica2);
  -- devolver recurso
  signal(final_accion_atomica1);
  signal(inicio_accion_atomica1);
end codigo_para_primer_proceso;

procedure codigo_para_segundo_proceso is
begin
  -- comienzo acción atómica
  wait(inicio_accion_atomica2);
     -- procesamiento inicial

     -- esperar a que el primer proceso señale que
     -- se puede acceder al recurso

     -- acceso al recurso

  signal(final_accion_atomica2);
  wait(final_accion_atomica1);
  signal(inicio_accion_atomica2);
end codigo_para_segundo_proceso;
```

Esta estructura proporciona la protección necesaria, y, de nuevo, muestra que es posible utilizar semáforos para programar la mayoría de los problemas de sincronización; aun así, esta aproximación tiene diversos problemas. En primer lugar, aunque sólo se permiten dos procesos en la acción atómica, no existe garantía de que sean correctos.

En segundo lugar, los semáforos son propensos a errores, lo cual se criticó ya en los capítulos iniciales: un único acceso sin protección (esto es, saliéndose de la estructura wait-signal) rompería la atomicidad. Por último, el extender esta solución a *N* procesos resulta muy complicado y tendente a errores.

10.2.2 Monitores

Es mucho más fácil ocultar las ejecuciones parciales si se encapsula la acción atómica en un monitor. A continuación se presenta el ejemplo anterior, pero ahora implementado como un monitor. La

instrucción `if` al comienzo de cada procedimiento asegura que sólo un proceso tiene acceso, de modo que las variables condicionales procuran la correcta sincronización dentro de la acción.

Sin embargo, esta solución tiene dos problemas. No es posible que dos procesos estén activos simultáneamente dentro del monitor. Esto es, a menudo, más estricto de lo que sería necesario. Además, la implementación de acciones anidadas y la asignación de recursos requerirá llamadas anidadas de monitores. Las dificultades asociadas con las llamadas anidadas de monitores se discutieron en la Sección 8.6.4. El mantenimiento del bloqueo del monitor mientras se ejecuta una llamada anidada, podría retrasar innecesariamente a los procesos que intentan ejecutarse en el contexto de una acción más externa.

```
monitor accion_atomica
  export codigo_para_primer_proceso, codigo_para_segundo_proceso;

  primer_proceso_activo : boolean := false;
  segundo_proceso_activo : boolean := false;
  primer_proceso_finalizado : boolean := false;
  segundo_proceso_finalizado : boolean := false;
  no_primer_proceso, no_segundo_proceso : condition;
  final_accion_atomica1, final_accion_atomica2 : condition;

procedure codigo_para_primer_proceso
begin

  if primer_proceso_activo then
    wait(no_primer_proceso);
  primer_proceso_activo := true;
    -- obtener el recurso en modo no compartible
    -- actualizar recurso

    -- señal al segundo proceso para que
    -- acceda al recurso

    -- procesamiento final
  if not segundo_proceso_finalizado then
    wait(final_accion_atomica2);
  primer_proceso_finalizado := true;
  -- liberar recurso
  signal(final_accion_atomica1);
  primer_proceso_activo := false;
  signal(no_primer_proceso);
end;
```

```
procedure codigo_para_segundo_proceso
begin
  if segundo_proceso_activo then
    wait(no_segundo_proceso);
  segundo_proceso_activo := true;
    -- procesamiento inicial

    -- esperar a que el primer proceso señale que
    -- se puede acceder al recurso

    -- acceso al recurso

  signal(final_accion_atomica2);
  segundo_proceso_finalizado := true;
  if not primer_proceso_finalizado then
    wait(final_accion_atomica1);
  segundo_proceso_activo := false;
  signal(no_segundo_proceso);
end;
```

Pueden eliminarse muchas de las limitaciones de la solución anterior si se utiliza el monitor como un «controlador de acción» y se realizan las propias acciones fuera del monitor. Ésta es la aproximación adoptada en el modelo Ada, la cual se da a continuación.

10.2.3 Acciones atómicas en Ada

En los lenguajes en los que las primitivas de comunicación y sincronización se basan únicamente en el paso de mensajes, todas las acciones de procesos elementales son atómicas si no existen variables compartidas y el propio proceso no se comunica durante la acción. Por ejemplo, el encuentro (cita o *rendezvous*) extendido en Ada está diseñado para permitir una forma común para la programación de acciones atómicas. Aquí, cierta tarea se comunica con otra para pedir algún cómputo; la tarea invocada realiza esta ejecución y responde a través de los parámetros de salida de la cita. La acción atómica toma la forma de una sentencia **accept**, y posee atomicidad de sincronización en tanto en cuanto:

- No actualiza ninguna variable a la que pueda acceder otra tarea.
- No se «encuentra» con ninguna otra tarea.

Una acción atómica entre tres tareas en Ada, podría programarse con una cita anidada; esto, sin embargo, no permitiría paralelismo alguno dentro de la acción.

Un modelo alternativo sería crear un controlador de acción y programar la sincronización requerida. Ésta es la aproximación que se sigue a continuación, utilizando un objeto protegido para el controlador.

```
package Accion_X is
  procedure Codigo_Primera_Tarea(--parametros);
  procedure Codigo_Segunda_Tarea(--parametros);
  procedure Codigo_Tercera_Tarea(--parametros);
end Accion_X;

package body Accion_X is
  protected Controlador_Accion is
    entry Primera;
    entry Segunda;
    entry Tercera;
    entry Terminada;
  private
    Primera_Aqui : Boolean := False;
    Segunda_Aqui : Boolean := False;
    Tercera_Aqui : Boolean := False;
    Liberar : Boolean := False;
  end Controlador_Accion;

  protected body Controlador_Accion is
    entry Primera when not Primera_Aqui is
    begin
      Primera_Aqui := True;
    end Primera;

    entry Segunda when not Segunda_Aqui is
    begin
      Segunda_Aqui := True;
    end Segunda;

    entry Tercera when not Tercera_Aqui is
    begin
      Tercera_Aqui := True;
    end Tercera;

    entry Terminada when Liberar or Terminada'Count = 3 is
    begin
      if Terminada'Count = 0 then
```

```
            Liberar := False;
            Primera_Aqui := False;
            Segunda_Aqui := False;
            Tercera_Aqui := False;
         else
            Liberar := True;
         end if;
      end Terminada;
   end Controlador_Accion;

   procedure Codigo_Primera_Tarea(--parametros) is
   begin
      Controlador_Accion.Primera;
      -- adquirir recursos
      -- la propia acción se comunica con tareas en ejecución
      -- dentro de la acción a través de los recursos
      Controlador_Accion.Terminada;
      -- liberar recursos
   end Codigo_Primera_Tarea;

   -- similar para las tareas segunda y tercera
begin
      -- cualquier inicialización de recursos locales
end Accion_X;
```

En el ejemplo anterior, la acción es sincronizada por el objeto protegido `Controlador_Accion`. Esto asegura que sólo pueda haber tres tareas activas en la acción en un momento dado, y que se sincronicen en la salida. El valor lógico `Liberar` se utiliza para programar las condiciones precisas de liberación sobre `Terminada`. Las dos primeras llamadas sobre `Terminada` se bloquearán, ya que las dos partes de la expresión de guarda son falsas. Cuando llega la tercera llamada, el atributo `Count` pasa a ser tres; la guarda es liberada, y una tarea ejecutará el cuerpo de entrada. La variable `Liberar` asegura que las otras dos tareas son liberadas. La última tarea en salir debe asegurarse de que la guarda es cerrada de nuevo.

En Wellings y Burns (1997), se pueden encontrar más detalles sobre cómo programar acciones atómicas en Ada.

10.2.4 Acciones atómicas en Java

En la sección anterior se mostró la estructura básica para la programación de acciones atómicas. La aproximación Java sigue una estructura similar. Sin embargo, en esta sección se amplía esta aproximación para poder extenderla fácilmente utilizando herencia.

En primer lugar, se define una interfaz para una acción atómica de tres vías, o ternaria:

```
public interface AccionAtomicaTernaria
{
  public void rol1();
  public void rol2();
  public void rol3();
}
```

Extendiendo esta interfaz, se pueden proporcionar diversos controladores de acción que implementan varios modelos. Las aplicaciones podrán elegir entonces el controlador apropiado sin necesidad de cambiar el código.

El siguiente controlador de acción implementa la misma semántica que la dada previamente para semáforos, monitores y Ada, y de hecho el algoritmo es muy similar al de la versión en Ada. La clase sincronizada `Controlador` implementa los protocolos de sincronización precisos de entrada y salida. El método `terminada` es, sin embargo, algo más complejo que el de Ada. Esto se debe a la semántica de `wait` y `notifyAll`. En particular, es necesario contar los procesos cuando salen de la acción (utilizando `aSalir`) para poder saber cuándo inicializar, para la siguiente acción, las estructuras de datos internas. En Ada esto se conseguía con el atributo `Count`.

```
public class ControlAccionAtomica implements AccionAtomicaTernaria
{
  protected Controlador Control;
  public ControlAccionAtomica() // constructor
  {
    Control = new Controlador();
  }

  class Controlador
  {
    protected boolean primeraAqui, segundaAqui, terceraAqui;
    protected int todoHecho;
    protected int aSalir;
    protected int numeroParticipantes;

    Controlador()
    {
      primeraAqui = false;
      segundaAqui = false;
      terceraAqui = false;
      todoHecho = 0;
      numeroParticipantes = 3;
```

```
        aSalir = numeroParticipantes;
    }

    synchronized void primera() throws InterruptedException
    {
        while(primeraAqui) wait();
        primeraAqui = true;
    }

    synchronized void segunda() throws InterruptedException
    {
        while(segundaAqui) wait();
        segundaAqui = true;
    }

    synchronized void tercera() throws InterruptedException
    {
        while(terceraAqui) wait();
        terceraAqui = true;
    }

    synchronized void terminada() throws InterruptedException
    {
        todoHecho++;
        if(todoHecho == numeroParticipantes) {
            notifyAll();
        } else while(todoHecho != numeroParticipantes) {
                wait();
        }
        aSalir--;
        if(aSalir == 0) {
            primeraAqui = false;
            segundaAqui = false;
            terceraAqui = false;
            todoHecho = 0;
            aSalir = numeroParticipantes;
            notifyAll(); // liberar proceso en espera para la siguiente acción
        }
    }
}
public void roll()
```

```
    {
      boolean hecho = false;
      while(!hecho) {
        try {
          Control.primera();
          hecho = true;
        } catch (InterruptedException e) {
          // ignorar
        }
      }
      // .... realizar acción

      hecho = false;
      while(!hecho) {
        try {
          Control.terminada();
          hecho = true;
        } catch (InterruptedException e) {
          // ignorar
        }
      }
    };
    public void rol2()
    {
      // similar a rol1
    }

    public void rol3()
    {
      // similar a rol1
    }
}
```

Dado el entorno anterior, es posible ahora extenderlo para generar, por ejemplo, una acción de cuatro vías, o cuaternaria:

```
public interface AccionAtomicaCuaternaria extends AccionAtomicaTernaria {
  public void rol4();
}
```

y entonces:

```
public class NuevoControlAccionAtomica extends ControlAccionAtomica
```

```
            implements AccionAtomicaCuaternaria
{
  public NuevoControlAccionAtomica()
  {
    Control = new ControladorRevisado();
  }

  class ControladorRevisado extends Controlador
  {
    protected boolean cuartaAqui;

    ControladorRevisado() {
      super();
      cuartaAqui = false;
      numeroParticipantes = 4;
      aSalir = numeroParticipantes;
    }

    synchronized void cuarta() throws InterruptedException
    {
      while(cuartaAqui) wait();
      cuartaAqui = true;
    }

    synchronized void terminada() throws InterruptedException
    {
      super.terminada();
      if(todoHecho == 0) {
        cuartaAqui = false;
        notifyAll();
      }
    }
  }
  public void rol4()
  {
    boolean hecho = false;
    while(!hecho) {
      try {
        // dado que Control es de tipo Controlador, primero debe
        // ser convertido a ControladorRevisado para
        // llamar al método cuarta
```

```
            ((ControladorRevisado)Control).cuarta();
            hecho = true;
         } catch (InterruptedException e) {
            // ignorar
         }
      }

      // .... realizar acción

      hecho = false;
      while(!hecho) {
         try {
            Control.terminada();
            hecho = true;
         } catch (InterruptedException e) {
            // ignorar
         }
      }
   }
}
```

Obsérvese que ha sido necesario sobreescribir el método `terminada`. Aquí hay que poner un cuidado especial debido al distribuidor automático de llamadas a métodos en tiempo de ejecución de Java. Todas las llamadas a `terminada` en el código original serán pasadas al método sobreescrito.

10.2.5 Acciones atómicas en occam2

Cuando un lenguaje permite sólo el paso de mensajes, el propio controlador de la acción es también un proceso. Cada componente de la acción envía un mensaje al controlador al principio y al final de su operación atómica. Dado que la cita de occam2 no engloba varias sentencias, se necesita una doble interacción con el controlador al final de la acción. Además, como los canales en occam2 tienen un único lector y un único escritor, el controlador de la acción necesita tener un array de canales para cada uno de los componentes de la acción. A cada proceso potencial se le asignará un canal de este array.

```
VAL INT max IS 20:  -- número máximo de procesos cliente
                    -- en cualquiera de los tres grupos
[max]CHAN OF INT Primera, Segunda, Tercera:
CHAN OF INT Primera.Terminada, Segunda.Terminada, Tercera.Terminada:
CHAN OF INT Primera.Continuar, Segunda.Continuar, Tercera.Continuar:
```

```
-- todos los canales anteriores se utilizan sólo para sincronización
-- se definen por defecto como protocolo de INT

PROC Accion.Controlador
  BOOL Primera.Aqui, Segunda.Aqui, Tercera.Aqui:
  INT Any:
  WHILE TRUE
    SEQ
      Primera.Aqui := FALSE
      Segunda.Aqui := FALSE
      Tercera.Aqui := FALSE
      WHILE NOT (Primera.Aqui AND Segunda.Aqui AND Tercera.Aqui)
        ALT
          ALT i = 0 FOR max
            NOT Primera.Aqui & Primera[i]?Any
              Primera.Aqui := TRUE
          ALT i = 0 FOR max
            NOT Segunda.Aqui & Segunda[i]?Any
              Segunda.Aqui := TRUE
          ALT i = 0 FOR max
            NOT Tercera.Aqui & Tercera[i]?Any
              Tercera.Aqui := TRUE
        Primera.Terminada?Any
        Segunda.Terminada?Any
        Tercera.Terminada?Any
        PAR
          Primera.Continuar!Any
          Segunda.Continuar!Any
          Tercera.Continuar!Any
:

PROC accion.1(CHAN OF INT primera.cliente)
  INT Any:
  SEQ
    primera.cliente!Any

    -- la propia acción

    Primera.Terminada!Any
    Primera.Continuar?Any
:
```

```
-- similar para accion.2 y accion.3

PAR
  -- todos los procesos del sistema incluyendo
  Accion.Controlador
```

10.2.6 Un entorno de lenguaje para acciones atómicas

Aunque los distintos modelos de lenguaje ya descritos permiten representar acciones atómicas simples, todos se basan en una programación disciplinada para asegurar que no ocurran interacciones con procesos externos (aparte de los asignadores de recursos). Además, se presupone que ningún proceso dentro de una acción atómica podrá ser abortado; si el lenguaje de tiempo real incluye un mecanismo de aborto, entonces un proceso podría ser eliminado asíncronamente de la acción, dejándola en un estado inconsistente.

En general, ninguno de los principales lenguajes o sistemas operativos soporta directamente la recuperación de errores hacia adelante o hacia atrás en el contexto de las acciones atómicas; aun así, Ada, Java y POSIX proporcionan mecanismos de notificación asíncrona que pueden ser usados para ayudar a la recuperación del programa (véanse las Secciones 10.6, 10.8 y 10.9). En varios sistemas de investigación se han propuesto algunos mecanismos para el lenguaje. Para comentar estos mecanismos se presenta un entorno de lenguaje sencillo para acciones atómicas. En este contexto, se discutirán los mecanismos de recuperación.

Para simplificar el entorno sólo se considerarán procesos estáticos. También se asumirá que todos los procesos que intervienen en una acción atómica son conocidos en tiempo de compilación. Cada proceso involucrado en una acción atómica declara una sentencia de acción que especifica: el nombre de la acción, los otros procesos que forman parte de la acción, y el código a ejecutar por el proceso declarante a la entrada de la acción. Por ejemplo, un proceso P_1 que desea entrar en una acción atómica A con los procesos P_2 y P_3, declararía la acción siguiente:

```
action A with (P2, P3)do
  -- adquirir recursos
  -- comunicarse con P2 y P3
  -- liberar recursos
end A;
```

Se presupone que los asignadores de recursos son conocidos y que las comunicaciones dentro de la acción están restringidas a los tres procesos P (junto con las llamadas externas a los asignadores de recursos). Estas restricciones son comprobadas en tiempo de compilación. El resto de procesos declaran acciones similares, y se permiten las acciones anidadas en tanto en cuanto se siga un anidamiento estricto. Téngase en cuenta que si los procesos son conocidos en tiempo de compilación, entonces sólo se permiten comunicaciones si los dos están activos en la misma acción atómica.

La sincronización que se impone sobre la acción es la siguiente. Los procesos que entran en la acción no son bloqueados. Un proceso sólo es bloqueado dentro de la acción si tiene que esperar a que un recurso le sea asignado, o si intenta comunicarse con otro proceso dentro de la acción, y ese proceso o bien está activo pero no en una posición de aceptar la comunicación, o bien todavía no está activo en la acción.

Los procesos sólo pueden abandonar la acción cuando todos los procesos activos en la acción deseen abandonar. Éste no era el caso en los ejemplos anteriores de semáforo, monitores, Ada y occam2, donde se asumía que todos los procesos debían entrar en la acción antes de que cualquiera pudiera abandonar. Aquí, es posible que un subconjunto de los procesos indicados entren en la acción y posteriomente abandonen (sin recurrir a interacciones con los procesos que faltan). Esta posibilidad se considera esencial en los sistemas de tiempo real, donde los plazos de tiempo son importantes. Esto resuelve el problema del **desertor**, en el que todos los procesos se mantienen en una acción porque otro no ha llegado. Este problema será considerado en las próximas dos secciones al tratar sobre recuperación de errores.

10.3 Acciones atómicas y recuperación de errores hacia atrás

En la sección anterior se consideró la noción de acción atómica. Las acciones atómicas son importantes, porque limitan el flujo de información del sistema a entornos bien definidos y, por lo tanto, proporcionan las bases tanto para el confinamiento de problemas como para la recuperación de errores. En esta sección se describen dos aproximaciones de recuperación de errores hacia atrás entre procesos. La discusión de recuperación de errores hacia atrás en el contexto de fallos del procesador se deja hasta el Capítulo 14.

En el Capítulo 5 se mostró que cuando la recuperación de errores hacia atrás se aplica a grupos de procesos que se comunican, puede que haya que retrotraer todos los procesos al comienzo de su ejecución. Esto se conoce como *efecto dominó*. El problema surgía de que no había un conjunto consistente de puntos de recuperación, o línea de recuperación. Las acciones atómicas proporcionan esa línea de recuperación de forma automática. Si ocurre un error dentro de la acción atómica, entonces los procesos involucrados pueden ser retrotraídos al comienzo de la acción y se pueden ejecutar algoritmos alternativos; la atomicidad de la acción asegura que los procesos no han comunicado valores erróneos a otros procesos exteriores a la acción. Cuando las acciones atómicas se utilizan de esta forma, se denominan **conversaciones** (Randell, 1975).

10.3.1 Conversaciones

En las conversaciones, cada una de las sentencias de acción contiene un bloque de recuperación. Por ejemplo:

```
action A with (P2, P3)do
  ensure <test de aceptación>
  by
    -- módulo primario
  else by
    -- módulo alternativo
  else by
    -- módulo alternativo
  else error
end A;
```

Cualquier otro proceso involucrado en la conversación declara su participación en la acción de manera similar. La semántica básica de una conversación puede resumirse de la siguiente forma:

- Al entrar en la conversación se guarda el estado del proceso. El conjunto de puntos de entrada forma la línea de recuperación.
- En el interior de la conversación, sólo se permite la comunicación con otros procesos activos en la conversación y con gestores generales de recursos. Esta propiedad se hereda según se crean conversaciones a partir de acciones atómicas.
- Para poder abandonar la conversación, todos los procesos activos en la conversación deben haber pasado el test de aceptación. En este caso, se finaliza la conversación y se desechan los puntos de recuperación.
- *Si cualquiera de los procesos falla el test de aceptación, entonces todos los procesos recuperarán el estado guardado al comienzo de la conversación y ejecutarán los módulos alternativos*. Se asume, por tanto, que cualquier recuperación de error dentro de una conversación *debe ser* realizada por *todos* los procesos participantes en la conversación.
- Las conversaciones pueden anidarse, pero sólo si el anidamiento es estricto.
- Si fallan todas las alternativas de la conversación, entonces la recuperación deberá realizarse en un nivel superior.

Observe que, según se definen las conversaciones en Randell (1975), todos los procesos que toman parte en una conversación deben haber entrado en ella antes de que cualquiera pueda salir. Esto difiere de la semántica que se describe aquí. Si un proceso no se incorpora a una conversación, bien por un retardo o bien porque haya fallado, la conversación puede ser completada con éxito siempre que el resto de procesos activos en la conversación no quieran comunicarse con aquél. Si un proceso intenta comunicarse con otro que falta, entonces puede o bien bloquearse y esperar a que llegue el proceso, o bien continuar. Esta aproximación tiene dos ventajas (Gregory y Knight, 1985):

- Se pueden especificar conversaciones donde participar no sea obligatorio.
- Permite que los procesos con plazos de tiempo puedan abandonar la conversación, continuar y, si es necesario, realizar alguna acción alternativa.

A pesar de que las conversaciones permiten la recuperación coordinada de un grupo de procesos, también tienen sus problemas. Un punto importante es que cuando falla la conversación, todos los procesos son recuperados y todos realizan sus módulos alternativos. Esto fuerza a que los mismos procesos vuelvan a comunicarse para conseguir el efecto deseado, y a que, en consecuencia, un proceso no pueda desligarse de la conversación, dándose una situación no deseada. Gregory y Knight (1985) destacan que, en la práctica, cuando un proceso falla en la consecución de su objetivo en un módulo primario donde se se comunica con un grupo de procesos, puede que quiera comunicarse con un grupo completamente distinto de procesos en su módulo secundario. Más aún, el test de aceptación para este módulo secundario puede ser distinto. No hay forma de expresar estos requisitos utilizando conversaciones.

10.3.2 Diálogos y coloquios

Para solucionar algunos de los problemas relacionados con las conversaciones, Gregory y Knight (1985) proponen una alternativa a la recuperación de errores hacia atrás entre procesos concurrentes. En su esquema, un grupo de procesos que desean tomar parte en una acción atómica recuperable hacia atrás lo indican ejecutando una instrucción **`dialog`**. La instrucción de diálogo tiene tres funciones: identifica a la acción atómica, declara un test de aceptación global para la acción atómica, y especifica las variables utilizadas en la acción. La instrucción de diálogo tiene la siguiente forma:

```
DIALOG nombre_y_test_aceptacion SHARES (variables)
```

El nombre del diálogo es el mismo que el de la función que define el test de aceptación. Cada proceso que quiere participar en la acción define una instrucción **`discuss`** que designa a la acción. El formato de la instrucción es:

```
DISCUSS nombre_dialogo BY
  -- secuencia de instrucciones
TO ARRANGE expresion_logica;
```

La `expresion_logica` es el test de aceptación local del proceso para la acción.

La instrucción de discusión es un componente de la acción atómica, y por tanto tiene todas la propiedades definidas anteriormente. La línea de recuperación viene dada por el grupo de sentencias de discusión que, conjuntamente, definen la acción completa. Por tanto, se guarda el estado de todo proceso que entre en el diálogo. Ningún proceso puede abandonar el diálogo a no ser que todos los procesos activos hayan pasado con éxito sus test locales de aceptación y se haya pasado también el test global de aceptación. Si falla cualquiera de estos test, se determina que el diálogo ha fallado, y los procesos son recuperados a su estado de entrada a la acción atómica.

Esto finaliza la operativa de la instrucción de discusión, ya que no se ejecutan módulos alternativos; en lugar de esto, estas instrucciones se pueden combinar con otras instrucciones, denominadas **secuencias de diálogo**. Por analogía, si la instrucción de discusión es equivalente al accept de Ada, entonces la secuencia de diálogo es equivalente a la sentencia select de Ada. Sintácticamente, se representa de la siguiente forma:

```
SELECT
  dialogo_1
OR
  dialogo_2
OR
  dialogo_3
ELSE
  -- secuencia de instrucciones
END SELECT;
```

En la ejecución, el proceso prueba primero con `dialogo_1`, y si tiene éxito, se pasa el control a la instrucción que sigue a la selección. Si `dialog_1` falla, entonces se prueba con `dialog_2`, y así sucesivamente. Es importante advertir que `dialog_2` puede involucrar a un conjunto de procesos completamente distinto al que involucraba `dialog_1`. Se denomina **coloquio** a la ejecución combinada de las instrucciones select asociadas.

Si fallan todos los diálogos probados, se ejecuta la instrucción que sigue a `ELSE`. Esto da al programador la última oportunidad de salvar la situación. Si esto falla, entonces falla cualquier coloquio circundante. Obsérvese que un proceso puede hacer fallar un diálogo o coloquio ejecutando la instrucción fail.

Para que el concepto de coloquio pueda ser utilizado en sistemas de tiempo real, se permite asociar un tiempo límite de espera a la selección. Esto se discute en la Sección 12.8.2.

10.4 Acciones atómicas y recuperación de errores hacia adelante

En el Capítulo 5 se destacó que, a pesar de que la recuperación de errores hacia atrás permite el tratamiento de errores no previstos, es difícil deshacer cualquier operación que haya sido realizada en el entorno en el que opera el sistema embebido. Por tanto, es necesario considerar la recuperación de errores hacia adelante y el manejo de excepciones. En esta sección se trata el manejo de excepciones entre procesos concurrentes involucrados en una acción atómica.

Con la recuperación de errores hacia atrás, cuando ocurre un error todos los procesos involucrados en la acción atómica participan en la recuperación. Lo mismo ocurre con el manejo de excepciones y la recuperación hacia adelante. Si ocurre una excepción en uno de los procesos activos de una acción atómica, la excepción se genera para todos los procesos activos de la ac-

ción. Se dice que la excepción es asíncrona, ya que se origina en otro proceso. A continuación se da una posible sintaxis tipo Ada para una acción atómica que soporta manejo de excepciones.

```
action A with ( P 2, P 3)do
  -- la acción
exception
  when excepcion_a =>
       -- secuencia de instrucciones
  when excepcion_b =>
       -- secuencia de instrucciones
  when others =>
     raise atomic_action_failure;
end A;
```

Con el modelo de terminación del manejo de excepciones, si todos los procesos activos en la acción tienen un manejador y todos manejan la excepción sin generar ninguna otra, entonces la acción atómica se completa normalmente. Si se utiliza un modelo de reanudación, cuando la excepción ha sido tratada, el proceso activo retoma su ejecución en el punto en el que la excepción fue generada.

Con cualquiera de los modelos, si no existe un manejador de excepciones *en ninguno de los procesos activos en la acción*, o falla uno de los manejadores, entonces *la acción atómica falla*, con la excepción estándar *atomic_action_failure*. Esta excepción se genera para todos los procesos involucrados.

Cuando se añade el manejo de excepciones a las acciones atómicas, deben considerarse dos asuntos: la resolución de excepciones concurrentes y las excepciones en acciones anidadas (Campbell y Randell, 1986). Ambas se verán a continuación.

10.4.1 Resolución de excepciones concurrentes

Puede ocurrir que haya más de un proceso activo en un acción atómica que genere excepciones diferentes al mismo tiempo. Como se apunta en Campbell y Randell (1986), esto es más que probable si no se pueden identificar unívocamente los errores resultantes de algún fallo por parte de la detección de errores que proporciona cada uno de los componentes de la acción atómica. Si se generan dos excepciones simultáneamente, podría haber dos manejadores de excepciones distintos en cada proceso, y podría ser complicado decidir cuál debe ser elegido. Más aún, la conjunción de las dos excepciones constituye una tercera excepción que indica que han ocurrido las condiciones de excepción de las otras dos.

Para resolver las excepciones concurrentes, Campbell y Randell proponen el uso de un **árbol de excepciones**. Si se han generado varias excepciones concurrentemente, la excepción que se utiliza para identificar el manejador es la que ocupa la raíz del subárbol más pequeño que contiene todas las excepciones (aunque no está claro cómo combinar los parámetros asociados con esta excepción). Cada componente de acción atómica puede declarar su propio árbol de excep-

ciones; los distintos procesos involucrados en una acción atómica pueden perfectamente tener distintos árboles de excepciones.

10.4.2 Excepciones y acciones atómicas internas

Al anidar acciones atómicas, es posible que un proceso activo en una acción genere una excepción cuando en la misma acción hay otros procesos involucrados en una acción anidada. La Figura 10.2 ilustra este problema.

Cuando se genera la excepción, todos los procesos involucrados deben participar en la acción de recuperación. Desafortunadamente, la acción interna, por definición, es indivisible. La aparición de la excepción en dicha acción podría comprometer potencialmente dicha indivisibilidad. Más aún, la acción interna podría no conocer las posibles excepciones que pueden generarse.

Campbell y Randell (1986) han abordado dos posibles soluciones a este problema. La primera solución consiste en retener la generación de la excepción hasta que la acción interna haya terminado. Esto es más que discutible, porque:

- En un sistema de tiempo real, la generación de una excepción puede asociarse con el fallo en un tiempo límite. Retrasar el procedimiento de recuperación podría comprometer seriamente el tiempo de respuesta de la acción.
- La condición de error detectada podría indicar que la acción interna no puede terminar debido a la aparición de alguna condición de interbloqueo.

Por estos motivos, Campbell y Randell permiten que las acciones internas tengan una excepción de aborto predefinida. Esta excepción indica a la acción que se ha generado una excepción en una acción circundante, y que ya no son válidas las precondiciones bajo las que fue invocada la acción. Ante tal excepción, la acción interna debería invocar mecanismos de tolerancia a fallos para abortarse a ella misma. Una vez abortada la acción, la acción circundante será capaz de manejar la excepción original.

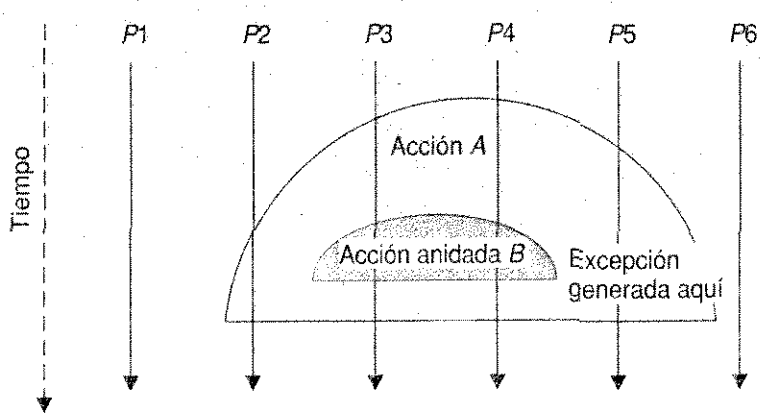


Figura 10.2. Una excepción en una acción atómica anidada.

Si la acción interna no puede ser abortada, entonces debe señalizar una excepción de fallo de acción atómica, que podrá ser combinada con la excepción pendiente, de forma que se pueda hacer una elección sobre la recuperación a realizar desde la acción circundante. Si no se define una excepción de aborto, la acción circundante debe esperar a la finalización de la acción interna. Alternativamente, se podría proporcionar un manejador por defecto, que generaría la excepción de fallo de acción atómica.

10.5 Notificación asíncrona

A pesar de haber abordado por separado la recuperación de errores hacia adelante y hacia atrás, en realidad puede que haya que combinarlas en muchos sistemas de tiempo real. La recuperación de errores hacia atrás es necesaria para recuperarse de errores imprevistos, y la recuperación hacia adelante es necesaria para deshacer o corregir cualquier interacción con el entorno. El manejo de errores hacia adelante puede incluso utilizarse para implementar un esquema de recuperación hacia atrás (véase la Sección 10.8.2).

Como se comentó en la Sección 10.2, ninguno de los lenguajes de tiempo real importantes soportan acciones atómicas, y es necesario utilizar mecanismos más primitivos del lenguaje para conseguir el mismo efecto. Lo mismo ocurre con las acciones recuperables, uno de cuyos principales requisitos es poder llamar la atención de un proceso involucrado en una acción y notificarle que ha ocurrido un error en otro proceso. La mayoría de los lenguajes y sistemas operativos soportan alguna forma de mecanismo de notificación asíncrona. Como ocurría con las excepciones, existen dos modelos básicos: reanudación y terminación.

El modelo de reanudación del manejo asíncrono de notificaciones (a menudo denominado **manejo de eventos**) se comporta como una interrupción software. Cada proceso indica qué eventos está dispuesto a manejar; cuando se señaliza el evento, el proceso es interrumpido (a no ser que temporalmente haya deshabilitado el envío del evento), y se ejecuta el manejador del evento. El manejador responde al evento asíncrono, y el proceso continúa su ejecución en el punto en el que fue interrumpido. Esto, por supuesto, es muy similar al modelo de reanudación del manejo de excepciones dado en la Sección 6.2.4. La diferencia fundamental es que el evento *no* es normalmente señalizado por el proceso afectado (o por una operación que está realizando el proceso afectado), sino que ocurre asíncronamente. Sin embargo, muchos sistemas operativos no proporcionan una funcionalidad especial de manejo de excepciones para el manejo síncrono, sino que utilizan en su lugar los mecanismos de eventos asíncronos. El mecanismo de señales POSIX es un ejemplo de modelo de eventos asíncronos con reanudación.

Observe que con el modelo de reanudación, el flujo de control de un proceso cambia sólo temporalmente; después de que el evento ha sido manejado, el proceso se reanuda. En un proceso multihilo es posible asociar un hilo distinto con el evento y programarlo con la señalización del evento. Java para tiempo real proporciona soporte para este modelo.

En el modelo de terminación de notificaciones asíncronas, cada proceso especifica un dominio de ejecución en el que está preparado para recibir una notificación asíncrona, que hará que el

dominio sea finalizado. A menudo, esta forma se conoce como **transferencia asíncrona de control** (ATC; asynchronous transfer of control). Si una petición ATC ocurre fuera de este dominio, puede ser ignorada o encolada. Una vez manejada la ATC, el control se devuelve al proceso interrumpido en una posición distinta a aquélla en la que la ATC fue enviada. Esto, por supuesto, es muy similar al modelo de terminación del manejo de excepciones. Los lenguajes Ada y Java para tiempo real soportan mecanismos de transferencia asíncrona de control.

La inclusión en un lenguaje (o sistema operativo) de mecanismos asíncronos de notificación es un tema controvertido, ya que complica la semántica del lenguaje e incrementa la complejidad del sistema de soporte de ejecución. Esta sección considera en primer lugar los requisitos de aplicación que justifican la inclusión de esta utilidad. Posteriormente, se abordan los modelos POSIX y Java para tiempo real de manejo asíncrono de eventos, y a continuación los mecanismos Ada y Java para tiempo real de transferencia asíncrona de control.

10.5.1 Las necesidades de usuario para la notificación asíncrona

El requisito fundamental de un mecanismo de notificación asíncrona es permitir que un proceso responda *rápidamente* a una condición que ha sido detectada por otro proceso. El énfasis está en la prontitud de la respuesta; obviamente, un proceso siempre puede responder a un evento simplemente sondeando o esperando ese evento. La notificación del evento puede construirse sobre el mecanismo de comunicación y sincronización de proceso. El proceso, cuando está disponible para recibir el evento, simplemente realiza la petición apropiada.

Desafortunadamente, hay ocasiones en las que no es adecuado esperar o sondear la ocurrencia de eventos, como en las siguientes:

- **Recuperación de errores**

  En este capítulo ya se ha recalcado que cuando hay grupos de procesos que realizan acciones atómicas, la detección de un error en un proceso necesita de la participación del resto de procesos en la recuperación. Por ejemplo, un fallo hardware puede suponer que el proceso nunca termine su ejecución prevista porque las condiciones de comienzo ya no se cumplen; el proceso puede que nunca alcance su punto de sondeo. Adicionalmente, podría ocurrir un fallo de temporización, lo que significaría que el proceso ya no podría cumplir el plazo límite de su servicio. En estas dos situaciones, el proceso debe ser informado de que ha sido detectado un error y de que debe efectuar alguna forma de recuperación de error lo más rápidamente posible.

- **Cambios de modo**

  A menudo, un sistema de tiempo real tiene distintos modos de operación. Por ejemplo, un avión civil *fly-by-wire* (con control software) puede tener un modo de despegue, un modo de vuelo y un modo de aterrizaje. En ocasiones habrá cambios de modo, y éstos deben ser gestionados en puntos bien definidos de la ejecución del sistema, como ocurre en el plan normal de vuelo de un avión. Sin embargo, en algunas áreas de aplicación, los cambios de

modo pueden ser esperados pero no planificados. Por ejemplo, un fallo puede llevar a que un avión cancele su despegue y pase a un modo de operación de emergencia; o un accidente en un proceso de producción puede requerir un cambio inmediato de modo para asegurar la parada ordenada de la planta. En estas situaciones, los procesos deben ser informados de forma rápida y segura de que el modo en el que estaban operando ha cambiado, y deben proceder a un conjunto de acciones diferentes.

- **Planificación utilizando computaciones parciales/imprecisas**

  Existen muchos algoritmos en los que la precisión de los resultados depende del tiempo asignado a los cálculos. Por ejemplo, los cálculos numéricos, computaciones estadísticas y búsquedas heurísticas, generan una estimación inicial del resultado requerido, que después es mejorada con una mayor precisión. En tiempo de ejecución, a cada algoritmo se le asigna una cierta cantidad de tiempo, transcurrido el cual el proceso debe interrumpirse, de forma que se finaliza el mecanismo de refinamiento del resultado.

- **Interrupciones de usuario**

  En un entorno interactivo general, los usuarios a menudo desean finalizar el procesamiento actual porque han detectado una condición de error y desean comenzar de nuevo.

Una aproximación al manejo asíncrono de notificación es abortar el proceso y permitir que otro proceso efectúe alguna forma de recuperación. Todos los sistemas operativos y la mayoría de los lenguajes de programación concurrente proporcionan esta disponibilidad. Sin embargo, el aborto de un proceso puede ser costoso, y a menudo es una respuesta extrema a muchas condiciones de error. Consecuentemente, es necesario algún mecanismo de notificación asíncrona.

10.6 Señales POSIX

POSIX proporciona un mecanismo asíncrono de manejo de eventos denominado **señal**, que también puede ser utilizado para cierto tipo de errores síncronos de entorno (como división por cero, apuntadores ilegales, y similares). Las señales se definieron antes que los hilos, y ha habido no pocas dificultades para extender el modelo a un entorno multihilos. Para procesos monohilo el modelo es bastante sencillo (véase el Programa 10.1 para una especificación de una interfaz C para la interfaz de señales POSIX). Existe un conjunto de señales predefinidas, a cada una de las cuales se le asigna un valor entero. También hay un conjunto de señales que se definen en implementación, y que están disponibles para uso de aplicaciones. Cada señal tine un manejador por defecto, que normalmente finaliza el proceso de recepción. Ejemplos de señales son las siguientes: `SIGABRT` para terminación anormal; `SIGALARM` para la expiración de una alarma de reloj; `SIGILL` para la excepción de una instrucción ilegal; `SIGRTMIN` para el identificador de la primera excepción definible de la aplicación de tiempo real; y `SIGRTMAX` para el identificador de la última excepción definible de la aplicación de tiempo real. Solamente aquellas señales cuya numeración esté entre `SIGRTMIN` y `SIGRTMAX` son consideradas de tiempo real en POSIX. Una

señal de tiempo real es aquélla que tiene información adicional, y que es enviada al manejador desde el proceso que la generó; además, son guardadas en una cola.

Existen tres formas en que un proceso puede tratar una señal:

Programa 10.1. Una interfaz C para señales POSIX.

```
union sigval {
  int sival_int;
  void *sival_ptr;
};

struct sigevent {
  /* utilizada para notificación de mensajes y temporizadores */
  int sigev_notify; /* notificación: SIGEV_SIGNAL, */
                    /* SIGEV_THREAD o SIGEV_NONE */
  int sigev_signo;  /* señal a generar */
  union sigval sigev_value; /* valor a guardar en la cola */
  void (*)sigev_notify_function(union sigval s);
  /* función a ser tratada como un hilo */
  pthread_attr_t *sigev_notify_attributes;
  /* atributos del hilo */
}

typedef struct {
  int si_signo;
  int si_code;
  union sigval si_value;
} siginfo_t;

typedef ... sigset_t;

struct sigaction {
  void (*sa_handler) (int signum); /* manejador no tiempo real */
  void (*sa_sigaction) (int signum, siginfo_t *data,
                        void *extra);   /* manejador tiempo real */
  sigset_t sa_mask; /* señales a enmascarar en el manejador */
  int sa_flags; /* indica si la señal debe ser guardada en la cola */
};

int sigaction(int sig, const struct sigaction *reaccion,
              struct sigaction *antigua_reaccion);
/* establece un manejador de señal, y una reacción */

/* la siguiente función permite que */
```

(Continuación)

```
/* un proceso espere a una señal */
int sigsuspend(const sigset_t *sigmask);
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
int sigtimedwait(const sigset_t *set, siginfo_t *info,
                 const struct timespec *timeout);

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
/* manipula una máscara de señal de acuerdo con el valor de how       */
/* how = SIG_BLOCK -> el conjunto es añadido al conjunto actual       */
/* how = SIG_UNBLOCK -> el conjunto es substraído del conjunto actual */
/* how = SIG_SETMASK -> el conjunto pasa a ser la máscara */

/* la siguiente rutina permite que un conjunto de */
/* señales sea creado y manipulado*/
int sigemptyset(sigset_t *s); /* inicializa el conjunto a vacío */
int sigfillset(sigset_t *s); /* inicializa el conjunto a lleno */
int sigaddset(sigset_t *s, int signum); /* añade una señal */
int sigdelset(sigset_t *s, int signum); /* quita una señal */
int sigismember(const sigset_t *s, int signum);
  /* devuelve 1 si miembro */

int kill (pid_t pid, int sig);
  /* envía la señal sig al proceso pid */
int sigqueue(pid_t pid, int sig, const union sigval value);
  /* envía señal y datos */

/* Todas las funciones anteriores devuelven -1 si ocurren errores */
/* Una variable compartida errno contiene el motivo del error */
```

- Puede **bloquear** la señal y manejarla o aceptarla más tarde.
- Puede **manejar** la señal estableciendo una función que será llamada cuando ocurra.
- Puede **ignorar** la señal en conjunto (en cuyo caso la señal símplemente se pierde).

Una señal que no es bloqueada ni ignorada, es **enviada** tan pronto como es **generada**. Una señal que es bloqueada está **pendiente** de envío, o puede ser **aceptada** llamando a una de las funciones `sigwait()`.

10.6.1 Bloqueo de una señal

POSIX mantiene el conjunto de señales que han sido enmascaradas por un proceso. Para manipular este conjunto se utiliza la función `sigprocmask`. Se puede establecer el valor del pará-

metro how en: SIG_BLOCK, para añadir señales al conjunto; SIG_UNBLOCK, para eliminar señales del conjunto; o SIG_SETMASK, para reemplazar el conjunto.[1] Los otros dos parámetros contienen apuntadores al conjunto de señales a añadir/eliminar/reemplazar (set) y al valor de retorno del conjunto antiguo (oset). La manipulación del conjunto de señales se realiza por medio de varias funciones (sigemptyset, sigfillset, sigaddset, sigdelset, y sigismember).

Cuando una señal es bloqueada, queda como pendiente hasta que es desbloqueada o aceptada. Cuando es desbloqlueada, es enviada. Algunas señales *no pueden* ser bloqueadas.

10.6.2 Manejo de una señal

Se puede establecer un manejador de señal utilizando la función sigaction. El parámetro sig indica qué señal debe ser manejada; reaccion es un apuntador a la estructura que contiene información sobre el manejador; y antigua_reaccion apunta a la información sobre el manejador anterior. En esencia, la información sobre el manejador consta de un apuntador a la función de manejo (sa_handler si no es una señal de tiempo real, o sa_sigaction si la señal es de tiempo real), el conjunto de señales que deben ser enmascaradas durante la ejecución del manejador (sa_mask), y si la señal debe ser guardada en la cola (indicado por la asignación de sa_flags con la constante simbólica SA_SIGINFO; sólo las señales cuyos valores están entre SIGRTMIN y SIGRTMAX pueden ser guardadas). El elemento sa_handler indica la acción asociada con la señal, y puede ser:

- SIG_DFL: acción por defecto (normalmente la terminación del proceso).
- SIG_IGN: ignorar la señal.
- Apuntador a una función: que será llamada cuando la señal sea enviada.

Para las señales que no son de tiempo real, sólo se puede pasar un parámetro entero al manejador cuando se genera la señal. Normalmente, el valor de este parámetro referencia a la propia señal (un mismo manejador puede ser utilizado para más de una señal). Sin embargo, para las señales de tiempo real, se pueden pasar más datos a través del apuntador a la estructura siginfo_t. Esta estructura contiene el número de señal (de nuevo), un código que indica la causa de la señal (por ejemplo, una señal de tiempo), y un entero o valor apuntador.

Si se guarda más de una señal de *tiempo real* en la cola, la que tiene menor valor se envía primero (esto es, SIGRTMIN se envia antes que SIGRTMIN + 1, y así sucesivamente).

También, un proceso puede esperar a que llegue una señal utilizando las funciones sigsuspend, sigwaitinfo o sigtimedwait. La función sigsuspend reemplaza la máscara que se da en el parámetro de la llamada, y suspende el proceso hasta que:

(1) se envía una señal no bloqueada, y

(2) se ejecuta el manejador asociado.

[1] SIG_BLOCK, SIG_UNBLOCK y SIG_SETMASK son constantes en tiempo de compilación.

Si el manejador finaliza el proceso, la función `sigsuspend` nunca termina; de lo contrario, termina de modo que se restablece la máscara de la señal al estado existente *anterior* a la llamada de `sigsuspend`.

La función `sigwaitinfo` también suspende el proceso hasta que llega la señal. Sin embargo, en esta ocasión la señal debe ser bloqueada, y por tanto no se invoca al manejador. En lugar de esto, la función devuelve el número de la señal seleccionada y guarda la información de la señal enviada en el argumento `info`. La función `sigtimedwait` tiene la misma semántica que `sigwaitinfo`, pero permite que se especifique un plazo de tiempo para la suspensión. Si no se envían señales en este plazo, `sigwaitinfo` devulve –1 y `errno` con el valor `EAGAIN`.

Hay que tener cuidado al utilizar señales para la sincronización condicional. Existe la posibilidad de una condición *de carrera* entre la comprobación de que la señal ya ha llegado y la emisión de una petición que cause la suspensión. El protocolo apropiado consiste en bloquear primero la señal, comprobar luego si ha ocurrido y, en caso contrario, suspender y desbloquear la señal utilizando las funciones anteriores.

10.6.3 Ignorar una señal

Una señal puede ser ignorada simplemente estableciendo el valor de `sa_handler` a `SIG_IGN` en una llamada a la función `sigaction`.

10.6.4 Generación de una señal

Un proceso puede generar una señal que es mandada a otro proceso de dos formas distintas: a través de la función `kill`, o por medio de `sigqueue`. Esta última sólo puede enviar señales de tiempo real.

Sin embargo, recuerde que un proceso también puede hacerse la petición de envío de una señal a sí mismo: cuando expira un temporizador –por ejemplo, `SIGALRM` (véase la Sección 12.8.1)–, cuando se completan acciones asíncronas de I/O, en la recepción de un mensaje sobre una cola vacía de mensajes (véase la Sección 9.5), o por la utilización de una instrucción raise en C.

Para los temporizadores, acciones asíncronas de I/O y recepción de mensajes, la interfaz POSIX permite que se pase un valor de tipo `struct sigevent`, que define lo que debe ocurrir cuando se reciba la notificación del evento. Hay tres opciones:

- `SIGEV_NONE`: no se debe enviar notificación alguna.
- `SIGEV_SIGNAL`: generar una señal de la forma habitual.
- `SIGEV_THREAD`: llamar a la función `sigev_notify` como si fuese la rutina de comienzo de un hilo recién creado con `sigev_notify_attributes`. Un atributo nulo indica que el hilo debe ser tratado como si hubiera sido separado.

10.6.5 Un ejemplo sencillo de señales POSIX

El fragmento de programa siguiente muestra un ejemplo de manejo de señales POSIX. Cierto proceso realiza periódicamente algunos cálculos. El cómputo realizado depende de un modo de operación del sistema. Un cambio de modo se propaga a todos los procesos por medio de una señal de tiempo real `CAMBIO_MODO` definida en la aplicación. El controlador `cambiar_modo` simplemente cambia la variable global `modo`. Los procesos acceden a `modo` al principio de cada iteración. La señal `CAMBIO_MODO` es bloqueada para asegurar que el modo no cambie mientras está siendo accedido.

```
#include <signal.h>

#define MODO_A 1
#define MODO_B 2
#define CAMBIO_MODO SIGRTMIN +1

int modo = MODO_A;

void cambiar_modo(int signum, siginfo_t *data, void *extra) {
  /* manejador de señal */
  modo = data -> si_value.sival_int;
}

int main() {

  sigset_t mask, omask;
  struct sigaction s, os;
  int modo_local;

  SIGEMPTYSET(&mask);
  SIGADDSET(&mask, CAMBIO_MODO);

  s.sa_flags = SA_SIGINFO;
  s.sa_mask = mask;
  s.sa_sigaction = &cambiar_modo;
  s.sa_handler = &cambiar_modo;

  SIGACTION(CAMBIO_MODO, &s, &os); /* asignar manejador */

  while(1) {

    SIGPROCMASK(SIG_BLOCK, &mask, &omask);
```

```
        modo_local = modo;
      SIGPROCMASK(SIG_UNBLOCK, &mask, &omask);

      /* operación periódica que utiliza modo */
      switch(modo_local) {
        case MODO_A:

          ...
          break;
        case MODO_B:
          ...
          break;
        default:
          ...
      }
    }
    ...
}
```

10.6.6 Señales e hilos

El modelo original de señales de POSIX proviene de Unix, y fue extendido cuando se especificaron las extensiones de tiempo real de POSIX para hacerlo más apropiado para el tiempo real. El modelo se ha hecho más complejo con las extensiones de hilos de POSIX, representando un compromiso entre un modelo para procesos y un modelo para hilos. Hay que tener en cuenta los siguientes puntos:

- Las señales que son generadas como resultado de un condición de error síncrono (como una infracción de memoria) se envían sólo al hilo causante de la señal.
- Otras señales pueden ser enviadas al proceso en su conjunto, aunque a un único hilo en el proceso.
- La función `sigaction` asigna el manejador para *todos* los hilos del proceso.
- Las funciones `kill` y `sigqueue` son aplicables a procesos. Una función nueva, `pthread_kill`,

  ```
  int pthread_kill(pthread_t hilo, int señal);
  ```

  permite que un proceso envíe una señal a un hilo particular.
- La función `pthread_cancel` también puede ser utilizada para finalizar los hilos:

  ```
  int pthread_cancel(pthread_t hilo);
  ```

El efecto de una petición de cancelación puede ser deshabilitado utilizando la función `pthread_setcancelstate`, o diferido con la función `pthread_setcanceltype`.

```
int pthread_setcancelstate(int estado, int *estadoantiguo);
int pthread_setcanceltype(int tipo, int *tipoantiguo);
```

- Si una señal puede ser enviada a más de un hilo, no está definido qué hilo será elegido.
- Si el manejador de la señal especifica como acción la terminación, entonces *el proceso completo es finalizado*, y no solamente el hilo.
- Las señales pueden ser bloqueadas a nivel de hilo utilizando la función `pthread_sigmask`, que tiene el mismo conjunto de parámetros que `sigprocmask`. No está especificada la función `sigprocmask` para procesos multihilos.
- Las funciones `sigsuspend`, `sigwaitinfo` o `sigtimedwait` operan sobre el hilo que hace la llamada, no sobre el proceso que hace la llamada.
- Una nueva función `sigwait`:

  ```
  int sigwait(const sigset_t *set, int *sig);
  ```

  permite a un hilo esperar a que ocurra una de las diversas señales que están bloqueadas. Su comportamiento es el mismo que el de `sigwaitinfo()`, excepto que no se devuelve la información asociada a la señal. Las señales están especificadas en la ubicación referenciada por `set`. La función devuelve cero si se ha realizado una espera con éxito y la ubicación referenciada por `sig` contiene a la señal recibida.

  La función termina de inmediato si una de las señales ya está pendiente cuando se llama a la función. Si están pendientes más de una, no está definido cuál es elegida, a no ser que las únicas señales pendientes sean de tiempo real. En este caso, se elige aquélla de menor valor.

- Si un hilo establece la acción de una señal a «ignorar», no está especificado si se descarta inmediatamente a la señal generada, o si permanece pendiente.

A pesar de que POSIX permite la gestión de eventos asíncronos por parte de hilos y procesos, se debe tener precaución, ya que algunas de las llamadas al sistema de POSIX **async-signal** y **async-cancel** se dice que son inseguras. Si una señal interrumpe una función insegura llamada desde una función que captura señales, no está definido lo que pasa. Por ejemplo, no es seguro utilizar la función `pthread_cond_signal` en un manejador de señal, dado que se puede establecer una condición «de carrera» con la función `pthread_cond_wait`.

10.6.7 POSIX y las acciones atómicas

Dada la importante interacción que se produce entre las actividades de una acción atómica, es más apropiado considerar la acción que tiene lugar entre hilos POSIX en vez de la que tiene lu-

gar entre procesos POSIX. Existen al menos dos posibilidades para implementar una estructura tipo acción atómica entre hilos:

(1) Utilizar una combinación de señales `setjmp` y `longjmp` para programar la coordinación requerida. Sin embargo, `longjmp` y todas las llamadas al sistema del hilo son async-signal inseguras. Esto significa que no pueden ser llamadas desde un manejador de señal.

(2) Utilizar la creación y cancelación de hilos para programar la recuperación precisada. Debido a que los hilos POSIX están diseñados para ser menos costosos, esta aproximación no tiene la misma penalización de rendimiento que la que tendría una estructura de procesos más «pesados».

El uso del modelo de reanudación hace necesarias estas aproximaciones. Si se adopta el modelo de terminación se obtiene una estructura más sencilla, que será discutida en el contexto de Ada (en la Sección 10.8) y de Java para tiempo real (en la Sección 10.9).

10.7 Manejo de eventos asíncronos en Java para tiempo real

Los eventos asíncronos en Java para tiempo real son el equivalente a las señales POSIX. Incluso existe una clase, `POSIXSignalHandler`, que realiza la correspondencia entre las señales POSIX y los eventos de Java para tiempo real cuando se implementa sobre un sistema operativo compatible POSIX (véase la Sección 15.5.2).

El Programa 10.2 muestra las tres clases principales asociadas con los eventos asíncronos en Java para tiempo real. Cada `AsyncEvent` puede tener uno o más `AsyncEventHandler`. Cuando ocurre el evento (indicado por una llamada al método `fire`), se planifica la ejecución de los manejadores asociados de acuerdo con sus parámetros `SchedulingParameters` (véase la Sección 13.14.3). Nótese que también se puede asociar el disparo de un evento con la ocurrencia de una acción externa dependiente de la implementación, por medio del método `bindTo`.

Cada manejador es planificado una vez para cada disparo de evento pendiente. Sin embargo, el manejador puede modificar el número de eventos pendientes por medio de la clase `AsyncEventHandler`.

El manejador de eventos es una entidad planificable, y sin embargo el objetivo es que no sufra los mismos costes operativos que un hilo de aplicación. Por tanto, no se puede asumir que exista un hilo de implementación separado para cada manejador, dado que cada hilo de implementación concreto puede tener asociado más de un manejador. Si se requiere un hilo dedicado, entonces se debería utilizar `BoundAsyncEventHandler`.

Programa 10.2. Las clases AsyncEvent, AsyncEventHandler y BoundAsyncEventHandler.

```
public class AsyncEvent
{
  public AsyncEvent();

  public synchronized void addHandler(AsyncEventHandler manejador);
  public synchronized void removeHandler(AsyncEventHandler manejador);
  public void setHandler(AsyncEventHandler manejador);
  // Asocia a este evento un nuevo manejador
  // eliminando todos los existentes.

  public void bindTo(java.lang.String ocurrencia);
  // enlaza a evento externo

  public ReleaseParameters createReleaseParameters();
  // crea un objeto ReleaseParameters que representa
  // las características de este evento

  public synchronized void fire();
  // Ejecuta el método run() del conjunto de manejadores para este evento.
  public boolean handledBy(AsyncEventHandler destino);
  // Devuelve true si este evento es manejado por este manejador.
}

public abstract class AsyncEventHandler implements Schedulable
{
  public AsyncEventHandler();
  // los parámetros se heredan del hilo actual

  public AsyncEventHandler(SchedulingParameters planificacion,
         ReleaseParameters liberacion, MemoryParameters memoria,
         MemoryArea area, ProcessingGroupParameters grupo);
  ... // otros constructores disponibles

  // métodos que implementan la interfaz Schedulable,
  // ver Capítulo 13

  protected final synchronized int getAndClearPendingFireCount();
  // Establece de forma atómica el número de ejecuciones pendientes
  // de este manejador a cero, y devuelve el valor de antes de que
```

(Continuación)

```
  // fuera borrado.
  protected synchronized int getAndDecrementPendingFireCount();
  protected synchronized int getAndIncrementPendingFireCount();

  public abstract void handleAsyncEvent();
  // Reemplaza este método para definir la acción a
  // realizar por este manejador

  public final void run();
}

public abstract class BoundAsyncEventHandler extends AsyncEventHandler
{
  public BoundAsyncEventHandler();
  // otros constructores
}
```

10.8 Transferencia asíncrona de control en Ada

En POSIX es posible establecer dominios para manejadores de señales bloqueando y desbloqueando la señal en puntos apropiados del programa. Sin embargo, esto se convierte en algo desestructurado y propenso al error si no existe soporte del lenguaje. Ada proporciona una forma más estructurada de notificación asíncrona, denominada **transferencia asíncrona de control** (ATC). Es más, el mecanismo se construye sobre el mecanismo de comunicación entre tareas, lo que enfatiza la idea de que ATC es una forma de comunicación y sincronización.

En el Capítulo 9 se introdujo la instrucción `select` de Ada, que presenta las siguientes formas:

- Un accept selectivo para soportar el lado del servidor del rendezvous (esto fue discutido en la Sección 9.4.2).
- Una llamada entry temporizada y una condicional, bien a una tarea bien a una entrada protegida (esto se discute en la Sección 12.4.2).
- Un select asíncrono (se discute aquí).

La instrucción select asíncrona proporciona un mecanismo de notificación asíncrono con semántica de terminación.

La ejecución de un select asíncrono comienza con el establecimiento de una llamada entry de disparo o de un retardo de disparo. Si la sentencia de disparo es una llamada entry, los parámetros se evalúan normalmente y se emite la llamada. Si la llamada es puesta en cola, entonces se ejecuta una secuencia de sentencias en una parte abort.

Si la sentencia de disparo se completa antes de que se complete la parte abortable, se aborta la parte abortable. Cuando hayan finalizado estas actividades, se ejecuta la secuencia de sentencias opcionales que siguen a la secuencia de disparo.

Si la parte abortable se completa antes de que se complete la invocación entry, se intenta cancelar la invocación entry y, si esto tiene éxito, se finaliza la ejecución de la sentencia select asíncrona.

```
select
  Trigger.Event;
  -- secuencia opcional de sentencias que son
  -- ejecutadas tras la recepción del evento
then abort
  -- secuencia abortable de sentencias
end select;
```

La sentencia de disparo puede ser un retardo, y por tanto se puede asociar plazo de tiempo con la parte abort (ver Sección 12.4.3).

Si falla la cancelación del evento de disparo porque ha comenzado la acción protegida o cita, la sentencia de select asíncrono espera a que se complete la sentencia de disparo antes de ejecutar la secuencia opcional que sigue a la sentencia de disparo.

Claramente, es posible que el evento de disparo ocurra antes de que haya comenzado la ejecución de la parte abort. En este caso, dicha parte no es ejecutada, y por tanto no es abortada.

Considérese el siguiente ejemplo:

```
task Servidor is
  entry Atc_Evento;
end Servidor;

task A_Interrumpir;
task body Servidor is
begin
  ...
  accept Atc_Evento do
    Seq2;
  end Atc_Evento;
  ...
end Servidor;
```

```
task body A_Interrumpir is
begin
   ...
   select -- sentencia ATC
      Servidor.Atc_Evento;
      Seq3;
   then abort
      Seq1;
   end select;
      Seq4;
      ...
end A_Interrumpir;
```

Cuando se ejecuta la sentencia ATC anterior, la ejecución de las sentencias dependerá del orden en el que ocurran los eventos:

```
si la cita está disponible inmediatamente, entonces
   Servidor.Atc_Evento es emitido
   Seq2 es ejecutada
   Seq3 es ejecutada
   Seq4 es ejecutada ( Seq1 nunca comienza)
si ninguna cita comienza antes de que temine Seq1, entonces
   Servidor.Atc_Evento es emitido
   Seq1 es ejecutada
   Servidor.Atc_Evento es cancelado
   Seq4 es ejecutada
si la cita termina antes de que termine Seq1, entonces
   Servidor.Atc_Evento es emitido
   ejecución parcial de Seq1 ocurre concurrentemente con Seq2
   Seq1 es abortada y finalizada
   Seq3 es ejecutada
   Seq4 es ejecutada
si no (la cita termina después de que termine Seq1)
   Servidor.Atc_Evento es emitido
   Seq1 es ejecutada concurrentemente con la ejecución parcial de Seq2
   Servidor.Atc_Evento se intenta cancelar sin éxito
   la ejecución Seq2 es completada
   Seq3 es ejecutada
   Seq4 es ejecutada
end if
```

Obsérvese que existe una condición «de carrera» entre la finalización de `Seq1` y la finalización de la cita. La situación puede ocurrir cuando `Seq1` finaliza y sin embargo es abortada.

Ada permite que algunas operaciones tengan **cancelación diferida** (abort deferred). Si `Seq1` contiene una operación de cancelación diferida, su cancelación no podrá ocurrir hasta que sea completada. Un ejemplo de tales operaciones es una llamada a un objeto protegido.

Hasta aquí se ha hablado del comportamiento concurrente de `Seq1` y de la cita de disparo. En una implementación multiprocesador podría darse el caso de que `Seq1` y `Seq2` se ejecutaran en paralelo. Sin embargo, en un sistema monoprocesador, el evento de disparo sólo ocurrirá si la acción que lo causa tiene mayor prioridad que `Seq1`. Por tanto, el comportamiento normal consistirá en el «desalojo» de `Seq1` por `Seq2`. Cuando se completa `Seq2` (la cita de disparo), se aborta `Seq1` antes de que pueda volver a ejecutarse. Por tanto, la ATC es «inmediata» (a no ser que se esté realizando una operación de cancelación diferida).

10.8.1 Excepciones y ATC

Con la sententencia de select asíncrono, existe la posibilidad de que dos actividades tengan lugar concurrentemente: la parte abort puede ejecutarse concurrentemente con la acción de disparo (cuando la acción es una llamada entry). En cualquiera de estas actividades se pueden generar excepciones que no son manejadas. Por tanto, la primera impresión es que, potencialmente, pueden propagarse dos excepciones simultáneamente desde una sentencia select. Sin embargo, esto no es así: se estima que una de las excepciones se perderá (la que se genera en la parte abort cuando es abortada), y por tanto sólo se propagará una excepción.

10.8.2 Ada y acciones atómicas

En la Sección 6.5 se mostró que la recuperación de errores hacia atrás en un sistema secuencial se puede implementar por manejo de excepciones. En esta sección se implementa la recuperación de errores hacia atrás y hacia adelante utilizando la ATC y el manejo de excepciones de Ada. Se asume que la implementación y ejecución subyacentes de Ada no fallan, y por tanto que el tipado fuerte de Ada asegura la viabilidad del propio programa.

Recuperación de errores hacia atrás

El siguiente paquete es una versión genérica del que se presentó en la Sección 6.5 para guardar y restaurar el estado de una tarea.

```
generic
   type Dato is private;
package Cache_Recuperacion is
   procedure Guardar(D : in Dato);
   procedure Restaurar(D : out Dato);
end Cache_Recuperacion;
```

Suponga que tres tareas Ada desean entrar en una acción atómica recuperable. Cada una llamará, en el paquete que se da a continuación, a su procedimiento apropiado.

```
package Conversacion is

   procedure T1(Params : Param); -- llamado por la tarea 1
   procedure T2(Params : Param); -- llamado por la tarea 2
   procedure T3(Params : Param); -- llamado por la tarea 3

   Fallo_Accion_Atomica : exception;

end Conversacion;
```

El cuerpo del paquete encapsula la acción y asegura que durante la conversación sólo se permitirá la comunicación entre las tres tareas.[2] El objeto protegido `Controlador` es el responsable de la propagación a todas las tareas de cualquier condición de error que se detecte en una tarea, así como de guardar y restaurar en la caché de recuperación cualquier dato persistente, y asegurar que todas las tareas salen de la acción al mismo tiempo. Contiene tres entradas protegidas y un procedimiento protegido.

- La entrada `Esperar_Abortar` representa el evento asíncrono sobre el que las tareas esperarán mientras realizan su parte de la acción.
- Una tarea invoca `Hecho` si ha terminado sin error su componente de la acción. Solamente cuando las tres tareas hayan realizado `Hecho`, podrán salir.
- De forma similar, una tarea llama a `Limpiar` si ha tenido que realizar una recuperación.
- Si una tarea reconoce una condición de error (bien porque se genera una excepción, bien por el fallo de un test de aceptación), llamará a `Señal_Abortar`. Esto hará que el indicador `Eliminado` pase a true como indicación de que la tarea debe ser recuperada.

Adviértase que, por la forma como se realiza la recuperación de errores hacia atrás, las tareas no están interesadas en la causa real del error. Cuando `Eliminado` pasa a verdadero, todas las tareas de la acción reciben el evento asíncrono. Una vez que el evento ha sido manejado, todas las tareas deben esperar sobre la entrada `Limpiar`, de forma que todas puedan terminar juntas el módulo de conversación.

```
with Cache_Recuperacion;
package body Conversacion is

   Fallo_Primario, Fallo_Secundario,
         Fallo_Terciario: exception;
type Module is (Primario, Secundario, Terciario);
```

[2] En la práctica, esto podría ser difícil de asegurar debido a las reglas de alcance de Ada. Una forma de aumentar la seguridad sería asegurar que el paquete `Conversacion` esté a nivel de biblioteca y su cuerpo sólo referencie paquetes «puros» (sin variables de estado). En la solución que se presenta se asume que las tareas «se comportan bien», y que, para simplificar, `T1`, `T2` y `T3` son invocados por las tareas correctas en los momentos precisos.

```
protected Controlador is
   entry Esperar_Abortar;
   entry Hecho;
   entry Limpiar;
   procedure Señal_Abortar;
private
   Eliminado : Boolean := False;
   Liberar_Hecho : Boolean := False;
   Liberar_Limpiar : Boolean := False;
   Informado : Integer := 0;
end Controlador;

-- cualquier objeto protegido local para comunicación entre acciones

protected body Controlador is
   entry Esperar_Abortar when Eliminado is
   begin
      Informado := Informado + 1;
      if Informado = 3 then
         Eliminado := False;
         Informado := 0;
      end if;
   end Esperar_Abortar;

   procedure Señal_Abortar is
   begin
      Eliminado := True;
   end Señal_Abortar;

   entry Hecho when Hecho'Count = 3 or Liberar_Hecho is
   begin
      if Hecho'Count > 0 then
         Liberar_Hecho := True;
      else
         Liberar_Hecho := False;
      end if;
   end Hecho;

   entry Limpiar when Limpiar'Count = 3 or Liberar_Limpiar is
   begin
      if Limpiar'Count > 0
```

```
            Liberar_Limpiar := True;
         else
            Liberar_Limpiar := False;
         end if;
      end Limpiar;
   end Controlador;

   procedure T1(Params : Param) is separate;
   procedure T2(Params : Param) is separate;
   procedure T3(Params : Param) is separate;
end Conversacion;
```

Cada procedimiento (por ejemplo `T1`), contiene el código de cada tarea. Dentro de cada procedimiento se realizan tres intentos de realizar la acción. Si todos los intentos fallan se genera la excepción `Fallo_Accion_Atomica`. Cada intento se acompaña de una llamada que guarda el estado y lo recupera (si el intento es fallido), encapsulado en un procedimiento local separado (`T1_Primario`, etc) que contiene una única sentencia «select and then abort» que realiza el protocolo establecido con el controlador. Las tareas utilizan la caché de recuperación para guardar sus datos locales.

```
separate(Conversacion)
procedure T1(Params : Param) is
   procedure T1_Primario is
   begin
      select
         Controlador.Esperar_Abortar; -- evento de disparo
         Controlador.Limpiar; -- esperar a que todas terminen
         raise Fallo_Primario;
      then abort
         begin
            -- código de implementación de la acción atómica
            -- el test de aceptación podría generar una excepción
            if Test_Accept = Failed then
               Controlador.Señal_Abortar;
            else
               Controlador.Hecho; -- señal de finalización
            end if;
         exception
            when others =>
               Controlador.Señal_Abortar;
         end;
      end select;
   end T1_Primario;
```

```
  procedure T1_Secundario is ... ;
  procedure T1_Terciario is ... ;

  package Mi_Cache is new Cache_Recuperacion(..); -- para datos locales
begin
  Mi_Cache.Guardar(..);
  for Intento in Modulo loop
    begin
      case Intento is
        when Primario => T1_Primario; return;
        when Secundario => T1_Secundario; return;
        when Terciario => T1_Terciario;
      end case;
    exception
      when Fallo_Primario =>
        Mi_Cache.Restaurar(..);
      when Fallo_Secundario =>
        Mi_Cache.Restaurar(..);
      when Fallo_Terciario =>
        Mi_Cache.Restaurar(..);
      raise Fallo_Accion_Atomica;
      when others =>
        Mi_Cache.Restaurar(..);
        raise Fallo_Accion_Atomica;
    end;
  end loop;
end T1;

-- similar para T2 y T3
```

La Figura 10.3 muestra un diagrama de transición de estados para una tarea que participa en una conversación.

Recuperación de errores hacia adelante

El mecanismo ATC de Ada puede utilizarse con excepciones para implementar acciones atómicas con recuperación de errores hacia adelante entre tareas que se ejecutan concurrentemente. Considérese el siguiente paquete para la implementación de una acción atómica entre tres tareas.

```
package Accion is

  procedure T1(Params : Param); -- llamada por tarea 1
  procedure T2(Params : Param); -- llamada por tarea 2
```

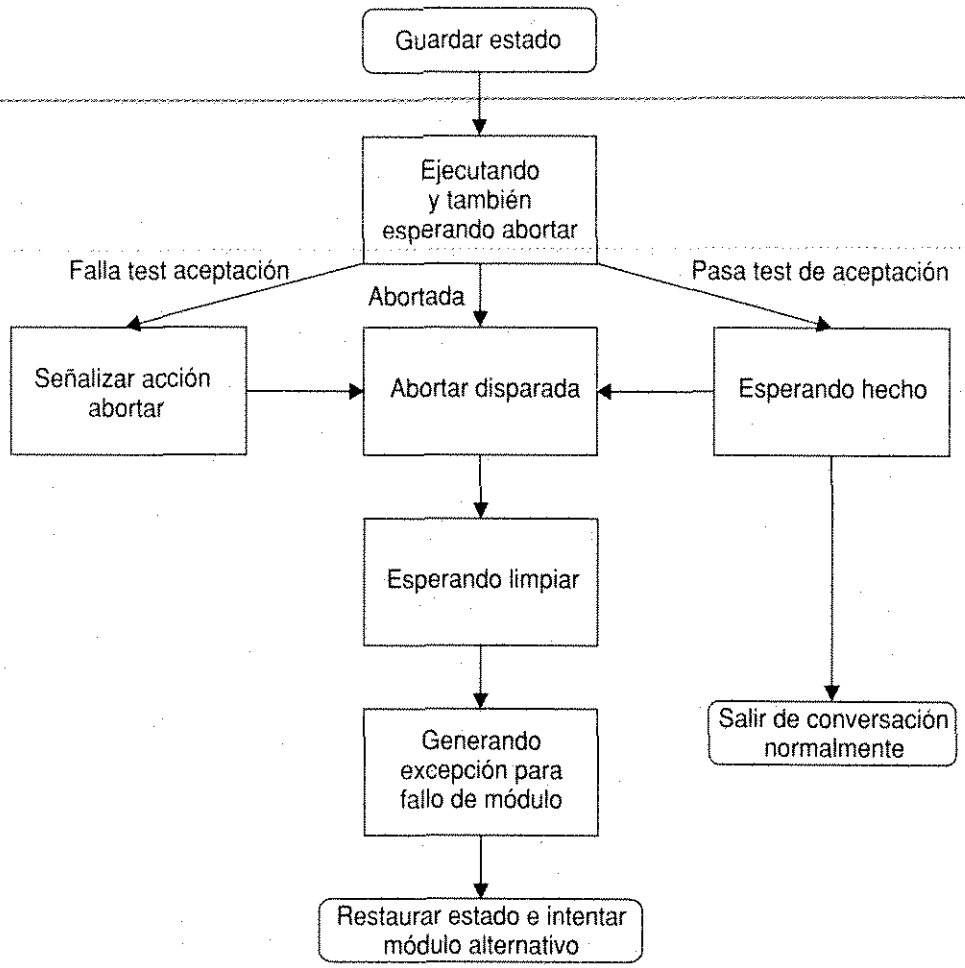


Figura 10.3. Diagrama de transición de estados para una conversación.

```
  procedure T3(Params : Param); -- llamada por tarea 3

  Fallo_Accion_Atomica : exception;

end Accion;
```

De la misma forma que con la recuperación de errores hacia atrás, el cuerpo del paquete encapsula la acción y asegura que sólo se permita la comunicación entre las tres tareas. El objeto protegido `Controlador` es el responsable de propagar a todas las tareas cualquier excepción generada en una tarea, así como de asegurar que todas las tareas salgan de la acción al mismo tiempo.

```
with Ada.Exceptions;
use Ada.Exceptions;
package body Accion is

  type Eleccion_T is (Commit, Abort);
  protected Controlador is
     entry Esperar_Abortar(E: out Exception_Id);
```

```
   entry Hecho;
   procedure Limpiar (Eleccion: Eleccion_T);
   entry Esperar_Limpiar(Resultado : out Eleccion_T);
   procedure Señal_Abortar(E: Exception_Id);
private
   Eliminado : Boolean := False;
   Liberar_Limpiar : Boolean := False;
   Liberar_Hecho : Boolean := False;
   Razon : Exception_Id;
   Resultado_Final : Eleccion_T := Commit;
   Informado : Integer := 0;
end Controlador;

-- cualquier objeto protegido local para comunicación entre acciones

protected body Controlador is
   entry Esperar_Abortar(E: out Exception_Id) when Eliminado is
   begin
      E := Razon;
      Informado := Informado + 1;
      if Informado = 3 then
         Eliminado := False;
         Informado := 0;
      end if;
   end Esperar_Abortar;

   entry Hecho when Hecho'Count = 3 or Liberar_Hecho is
   begin
      if Hecho'Count > 0 then
         Liberar_Hecho := True;
      else
         Liberar_Hecho := False;
      end if;
   end Hecho;

   procedure Limpiar (Eleccion: Eleccion_T) is
   begin
      if Eleccion = Abort then
         Resultado_Final := Abort;
      end if;
   end Limpiar;
```

```
      procedure Señal_Abortar(E: Exception_Id) is
      begin
         Eliminado := True;
         Razon := E;
      end Señal_Abortar;

      entry Esperar_Limpiar (Resultado: out Eleccion_T)
            when Esperar_Limpiar'Count = 3 or Liberar_Limpiar is
      begin
         Resultado := Resultado_Final;
         if Esperar_Limpiar'Count > 0 then
            Liberar_Limpiar := True;
         else
            Liberar_Limpiar := False;
            Resultado_Final := Commit;
         end if;
      end Esperar_Limpiar;
   end Controlador;

   procedure T1(Params: Param) is
      X : Exception_Id;
      Decision : Eleccion_T;
   begin
      select
         Controlador.Esperar_Abortar(X); -- evento de disparo
         Raise_Exception(X); -- generar excepción común
      then abort
         begin
            -- código de implementación de la acción atómica
            Controlador.Hecho; -- señal de finalización
         exception
            when E: others =>
               Controlador.Señal_Abortar(Exception_Identity(E));
         end;
      end select;
   exception
      -- si alguna excepción es generada durante la acción
      -- todas las tareas deben participar en la recuperación
      when E: others =>
```

```
      -- Exception_Identity(E) ha sido generada en todas las tareas

      -- manejar excepción
      if Manejada_Ok then
        Controlador.Limpiar(Commit);
      else
        Controlador.Limpiar(Abort);
      end if;
      Controlador.Esperar_Limpiar(Decision);
      if Decision = Abort then
        raise Fallo_Accion_Atomica;
      end if;
  end T1;

  procedure T2(Params : Param) is ...;

  procedure T3(Params : Param) is ...;

end Accion;
```

Todos los componentes de la acción (`T1`, `T2` y `T3`) tienen idéntica estructura, consistente en la ejecución de una sentencia select con una parte abort. El objeto protegido `Controlador` señaliza el evento de disparo si cualquiera de los componentes indica que se ha generado una excepción que no ha sido manejada *localmente en ninguno* de los componentes. La parte abort contiene el código efectivo del componente. Si este código se ejecuta sin incidentes, se informa al `Controlador` de que este componente está listo para realizar el commit de la acción. Si durante la parte abort se genera alguna excepción, se informa al `Controlador`, pasándole la identidad de la excepción. Vea que, al contrario de lo que ocurría con la recuperación de errores hacia atrás (sección anterior), aquí se debe comunicar la causa del error.

Si el `Controlador` ha recibido notificación de una excepción sin manejar, libera a todas las tareas que están esperando el evento de disparo `Esperar_Abortar` (cualquier tarea que llegue retrasada recibirá el evento inmediatamente en cuanto intente entrar en su sentencia select). La parte abort de las tareas es abortada (de haber comenzado), y se genera la excepción en cada tarea por medio de la sentencia posterior a la llamada entry del controlador. Si la excepción es manejada con éxito por el componente, la tarea indica que está lista para realizar el commit de la acción. Si no, indica que la acción debe ser abortada. Si alguna tarea indica que la acción debe ser abortada, todas las tareas generarán la excepción `Fall_Accion_Atomica`. La Figura 10.4 muestra un sencillo diagrama de transición de estados con esta aproximación.

El ejemplo anterior muestra que en Ada es posible programar acciones atómicas con recuperación de errores hacia adelante. Sin embargo, se deben hacer dos observaciones respecto a este ejemplo:

- Sólo la primera excepción que se pase al `Controlador` será generada en todas las tareas. Dado que cualquier excepción que se eleve en una parte abort se pierde cuando es abortada, no es posible obtener generación concurrente de excepciones.
- Esta solución no trata el problema del desertor. Si uno de los participantes de la acción no se presenta, los demás se quedan en estado de espera en el final de la acción. Para hacer frente a esta situación, es necesario que cada una de las tareas registre su llegada con el controlador de la acción (véase el Ejercicio 10.10).

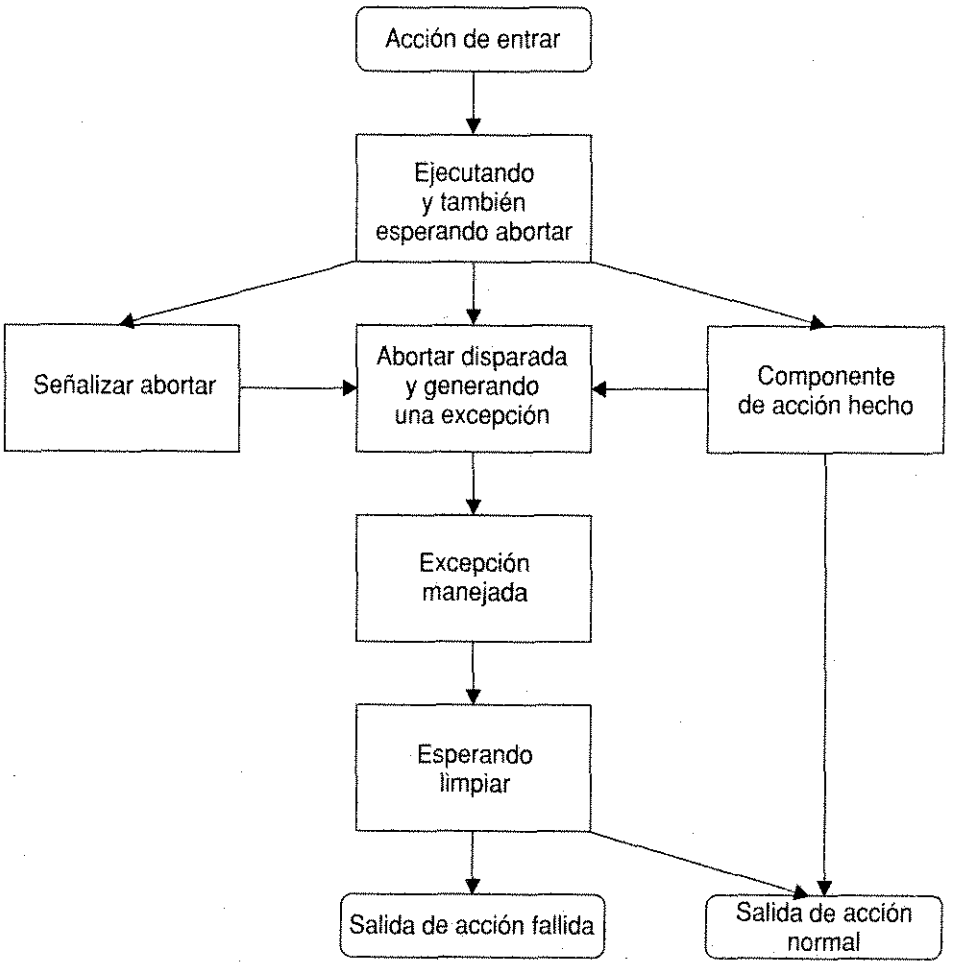


Figura 10.4. Diagrama de transición de estados que muestra la recuperación de errores hacia adelante.

10.9 Transferencia asíncrona de control en Java para tiempo real

Las primeras versiones de Java permitían que un hilo afectase asíncronamente a otro hilo por medio de los siguientes métodos:

```
public final void suspend() throws SecurityException;
```

```
public final void resume() throws SecurityException;

public final void stop() throws SecurityException;
public final void stop(Throwable except) throws SecurityException;
```

En la Sección 7.3.7 se discutieron los métodos `suspend` y `resume`. El método `stop` hace que el hilo pare su actividad y lance una excepción `ThreadDeath`, y, al igual que con el método `stop (Throwable except)`, sólo esta vez se lanza la excepción pasada como parámetro.

En la actualidad, estos métodos se consideran obsoletos, y no deben utilizarse. Java estándar soporta únicamente lo siguiente:

```
public void interrupt() throws SecurityException;
public boolean isInterrupted();
public void destroy();
```

Un hilo pueder señalizar una interrupción a otro hilo llamando al método `interrupt`. El resultado depende del estatus actual del hilo interrumpido.

- Si el método *interrumpido* está bloqueado en el método `wait`, `sleep` o `join`, se pasa a ejecutable y se lanza la excepción `InterruptedException`.
- Si el hilo interrumpido está en ejecución, se establece un indicador que señala que está pendiente una interrupción. *No tiene un efecto inmediato sobre el hilo interrumpido.* Más bien, el hilo invocado debe comprobar periódicamente, con el método `isInterrupted`, si ha sido «interrumpido».

Por sí mismo, este mecanismo no satisface las necesidades señaladas en la Sección 10.5.1.

El método `destroy` es similar a la funcionalidad abort de Ada, y destruye el hilo sin ninguna acción de limpieza.

Java para tiempo real proporciona una aproximación alternativa a la interrupción de hilos basada en la transferencia asíncrona de control (ATC). El modelo ATC de Java para tiempo real es similar al de Ada en cuanto a que es necesario indicar qué regiones del código pueden recibir una petición ATC. Sin embargo, el modelo de Java para tiempo real presenta dos importantes diferencias.

(1) El modelo de Java para tiempo real está integrado con el mecanismo de manejo de excepciones de Java, mientras que el modelo Ada está integrado con los mecanismos de la sentencia select y del manejo de entradas.

(2) El modelo de Java para tiempo real requiere que cada método indique que está preparado para permitir la ocurrencia de un ATC. La ATC es diferida hasta que el hilo esté en ejecución en dicho método. Por contra, la opción por defecto de Ada es permitir la ATC si un subprograma ha sido llamado desde una sentencia select-then-abort; las respuestas diferidas deben controlarse explícitamente.

Ambos lenguajes retrasan la ATC durante la interacción con otros hilos/tareas (por ejemplo en métodos/sentencias sincronizadas de Java, y en acciones protegidas y rendezvous de Ada), o dentro de constructores y cláusulas de finalización.

El modelo ATC de Java para tiempo real reúne el modelo de manejo de excepciones de Java y una extensión de interrupción de hilos. El modelo se explica mejor en dos fases. La primera es el soporte de bajo nivel y la estrategia general; la segunda es el uso del soporte de alto nivel como forma estructurada de gestionar la ATC. El uso de las facilidades básicas ATC necesita de tres actividades:

(1) Declaración de una `AsynchronouslyInterruptedException` (AIE; excepción interrumpida asíncronamente).

(2) Identificación de los métodos que pueden ser interrumpidos.

(3) Señalización de una `AsynchronouslyInterruptedException` al hilo.

El Programa 10.3 muestra la especificación de la clase `AsynchronouslyInterruptedException`. Los métodos serán explicados en su momento, y lo único que es necesario conocer por el momento, es que por cada hilo existe una AIE genérica asociada.

Una AIE puede ser ubicada en la lista de `throws` (lanzar), asociada con un método. Por ejemplo, considérese la siguiente clase, la cual proporciona un servicio interrumpible utilizando un paquete que declara servicios no interrumpibles (esto es, que no contienen ninguna `AsynchronouslyInterruptedException` en la lista `throws`).

```
import serviciosNoInterrumpibles.*;

public class ServicioInterrumpible
{
  public AsynchronouslyInterruptedException pararAhora =
         AsynchronouslyInterruptedException.getGeneric();

  public boolean Servicio() throws AsynchronouslyInterruptedException
  {
     // código intercalado con llamadas a serviciosNoInterrumpibles
  }
}
```

Supongamos ahora que un hilo de tiempo real `t` ha llamado a una instancia de esta clase para proporcionar el `Servicio`:

```
public ServicioInterrumpible IS = new ServicioInterrumpible();

// código del hilo, t
if(IS.Servicio()) { ... }
```

```
else { ... };
```

y que otro hilo interrumpe t:

```
t.interrupt();
```

Programa 10.3. La clase AsynchronouslyInterruptedException de Java para tiempo real.

```
public class AsynchronouslyInterruptedException extends
             java.lang.InterruptedException
{
  public AsynchronouslyInterruptedException();

  public synchronized boolean disable();
  // sólo válido en un doInterruptible, devuelve true si exitoso
  public boolean doInterruptible (Interruptible logica);
  // En un momento dado, sólo un objeto Interruptible específico puede
  // ejecutarse por cada hilo
  // Devuelve True si el Interruptible es ejecutado, false si ya
  // existe otro en progreso para este hilo.

  public synchronized boolean enable();
  public synchronized boolean fire();

  public boolean happened (boolean propaga);

  public static AsynchronouslyInterruptedException getGeneric();
  // devuelve el AsynchronouslyInterruptedException que
  // se genera cuando se invoca RealtimeThread.interrupt()

  public boolean isEnabled();

  public void propagate();
}
```

Las consecuencias de esta llamada dependen del estado actual de t cuando se realiza la llamada.

- Si, en cualquier momento, t está ejecutándose en una sección de ATC diferida, el AsynchronouslyInterruptedException se marca como pendiente. La excepción se lanza tan pronto como t abandone la mencionada región, y ejecuta un método con una declaración de AsynchronouslyInterruptedException en su lista throws.

- Si t está ejecutando un método que *no* tiene una declaración de AsynchronouslyInterruptedException en su lista throws (como ocurre con los del paquete serviciosNoInterrumpibles), entonces la excepción se marca como pendiente. La excepción se lanza tan pronto como t vuelva (o llame) a un método que pueda lanzar AsynchronouslyInterruptedException.

- Si t está ejecutándose en un bloque try de un método que ha declarado una AsynchronouslyInterruptedException en su lista throws, entonces el bloque es terminado y el control transferido a la primera sentencia de la cláusula catch. Si no se encuentra ninguna cláusula catch apropiada, se propaga la excepción AsynchronouslyInterruptedException al método que hizo la llamada. En caso contrario, se ejecuta el manejador correspondiente y se completa el procesamiento de AsynchronouslyInterruptedException (a no ser que la AIE se propague desde dentro del manejador).

- Si t está ejecutándose fuera de un bloque try en un método que ha declarado una AsynchronouslyInterruptedException en su lista throws, se termina el método, y la excepción AsynchronouslyInterruptedException se lanza inmediatamente hacia el método que hizo la llamada.

- Si t está bloqueado dentro de un método wait, sleep o join llamado desde un método que ha declarado una AsynchronouslyInterruptedException en su lista throws, se replanifica t y se lanza AsynchronouslyInterruptedException.

- Si t está bloqueado dentro de un método wait, sleep o join llamado desde un método que no ha declarado una AsynchronouslyInterruptedException en su lista throws, se replanifica t y se marca como pendiente la AsynchronouslyInterruptedException. La excepción se lanza tan pronto como t vuelva a un método con una declaración de AsynchronouslyInterruptedException en su lista throws.

Una vez que se ha lanzado la ATC y que se ha pasado el control a un manejador de excepciones apropiado, es necesario asegurarse de que la ATC atrapada es la esperada por el hilo interrumpido. Si lo es, la excepción puede ser manejada. Si no, la excepción debe ser propagada al método que hizo la llamada. Para este propósito se utiliza el método happened definido en la clase AsynchronouslyInterruptedException.

Considérese lo siguiente:

```
import serviciosNoInterrumpibles.*;
public class ServicioInterrumpible
{
  public AsynchronouslyInterruptedException pararAhora =
            new AsynchronouslyInterruptedException() ;
  public boolean Servicio() throws AsynchronouslyInterruptedException
  {
    try {
```

```
        // código intercalado con llamadas a serviciosNoInterrumpibles
      }
      catch AsynchronouslyInterruptedException AIE {
        if(pararAhora.happened(true)) {
          // manejar la ATC
        }
        // no hay cláusula else, el parámetro true indica que si
        // la excepción actual no es pararAhora,
        // debe ser inmediatamente propagada al
        // método que hizo la llamada
      }
    }
}
```

En este caso, cuando se lanza la AIE, el control se pasa a la cláusula catch del final del bloque try. Se encuentra un manejador para excepciones `AsynchronouslyInterruptedException`. Para poder determinar si la `AsynchronouslyInterruptedException` actual es `pararAhora`, se hace una llamada al método `pararAhora.happened`, que devuelve `true` si la excepción actual es `pararAhora`. Si no es así, se propaga la excepción, ya que el parámetro de `happened` es verdadero. Si el parámetro fuera falso, el control retornaría a la sentencia catch con un valor de `false`. El hilo podría realizar algunas rutinas de limpieza antes de propagar al excepción utilizando el método `propagate`:

```
catch AsynchronouslyInterruptedException AIE {
  if(pararAhora.happened(false)) {
    // manejar la ATC
  } else {
    // limpieza
    AIE.propagate();
  }
}
```

La interfaz `Interruptible`

Las discusiones anteriores muestran el mecanismo básico que proporciona Java para tiempo real para manejar las ATC. Para facilitar un uso estructurado, el lenguaje también proporciona una interfaz denominada `Interruptible` (véase el Programa 10.4).

Un objeto que desee proporcionar un método interrumpible lo hace implementando la interfaz `Interruptible`. El método `run` es el que es interrumpible; si se interrumpe el método `run`, el sistema llama al método `interruptAction`.

Una vez implementada la interfaz, ésta puede puede ser pasada como parámetro al método `doInterruptible` de la clase `AsynchronouslyInterruptedException`. Entonces,

el método puede ser interrumpido llamando al método `fire` de la clase `AsynchronouslyInterruptedException`. Los métodos `disable`, `enable` y `isEnabled` proporcionan un mayor control sobre `AsynchronouslyInterruptedException`. Una `AsynchronouslyInterruptedException` deshabilitada es diferida hasta que sea habilitada. En la Sección 10.9.1 se dará un ejemplo de esto último.

Adviértase que, para una `AsynchronouslyInterruptedException` concreta, sólo un método `doInterruptible` puede estar activo en un momento dado. El método termina inmediatamente con un valor de `false` si existe una llamada pendiente.

Programa 10.4. La interfaz `Interruptible` de Java para tiempo real.

```
public interface Interruptible
{

  public void interruptAction (
              AsynchronouslyInterruptedException excepcion);
  public void run (AsynchronouslyInterruptedException excepcion)
         throws AsynchronouslyInterruptedException;
}
```

Múltiples `AsynchronouslyInterruptedException`

Dado que la `AsynchronouslyInterruptedException` puede ser diferida, también es posible diferir varias ATC. Esto puede ocurrir cuando el método `run` de una clase (que implementa la interfaz `Interruptible`) llama a `doInterruptible` en una AIE. El método `run` asociado podría también llamar a otro `doInterruptible`. Por tanto, es posible para un hilo ejecutar métodos `doInterruptible` anidados. Considérese el siguiente ejemplo:

```
import javax.realtime.*;

public class ATCAnidado
{
  AsynchronouslyInterruptedException AIE1 = new
         AsynchronouslyInterruptedException();
  AsynchronouslyInterruptedException AIE2 = new
         AsynchronouslyInterruptedException();
  AsynchronouslyInterruptedException AIE3 = new
         AsynchronouslyInterruptedException();

  public void metodo1()
  {
```

```
    // región ATC diferida
  }

  public void metodo2() throws AsynchronouslyInterruptedException
  {
    AIE1.doInterruptible
      (new Interruptible()
        {
          public void run(AsynchronouslyInterruptedException e)
                 throws AsynchronouslyInterruptedException
          {
            metodo1();
          }
          public void interruptAction(
                 AsynchronouslyInterruptedException e)
          {
            if(AIE1.happened(false)) {
              // recuperar aquí
            } else {
              // limpiar
              e.propagate();
            }
          }
        }
    );
}

public void metodo3() throws AsynchronouslyInterruptedException
{
  AIE2.doInterruptible
    (new Interruptible()
      {
        public void run(AsynchronouslyInterruptedException e)
          throws AsynchronouslyInterruptedException
        {
          metodo2();
        }
        public void interruptAction(
               AsynchronouslyInterruptedException e)
        {
          if(AIE2.happened(false)) {
```

```
                // recuperar aquí
            } else {
                // limpiar
                e.propagate();
            }
        }
    });
}

public void metodo4() throws AsynchronouslyInterruptedException
{
  AIE3.doInterruptible
     (new Interruptible()
        {
          public void run(AsynchronouslyInterruptedException e)
             throws AsynchronouslyInterruptedException
          {
             metodo3();
          }
          public void interruptAction(
                AsynchronouslyInterruptedException e)
          {
             if(AIE3.happened(false)) {
                // recuperar aquí
             } else {
                // limpiar
                e.propagate();
             }
          }
        }
     );
  }
}
```

Sea ahora un hilo `t` que ha creado una instancia de `ATCAnidado` y ha llamado a `metodo4`, el cual ha llamado a `metodo3`, el cual ha llamado a `metodo2`, el cual ha llamado a `metodo1`, que es una región ATC diferida. Supóngase que el hilo es interrumpido por una llamada a `AIE2.fire()`. Ésta se mantiene pendiente. Si ahora llega `AIE3`, entonces se descarta a `AIE2`, ya que `AIE3` está en un nivel superior de anidamiento. Si llega `AIE1`, es ésta la que es eliminada (porque es de un nivel inferior). La AIE pendiente es lanzada en el momento en que finaliza `metodo1`.

10.9.1 Java para tiempo real y las acciones atómicas

Esta sección muestra cómo implementar en Java para tiempo real acciones atómicas con recuperación de errores hacia adelante utilizando las facilidades de ATC.

En primer lugar, se define `ExcepcionAccionAtomica AsynchronouslyInterruptedException`, junto con una excepción `FalloAccionAtomica`.

```
import javax.realtime.AsynchronouslyInterruptedException;

public class ExcepcionAccionAtomica extends
             AsynchronouslyInterruptedException
{
  public static Exception causa;
  public static boolean fueInterrumpido;
}

public class FalloAccionAtomica extends Exception;
```

Utilizando una `AccionAtomicaRecuperableTernaria` similar a la que fue definida con anterioridad:

```
public interface AccionAtomicaRecuperableTernaria {
  public void rol1() throws FalloAccionAtomica;
  public void rol2() throws FalloAccionAtomica;
  public void rol3() throws FalloAccionAtomica;
}
```

se puede implementar una clase `AccionRecuperable` con una estructura similar a la que se construyó en Ada.

```
import javax.realtime.*;

public class AccionRecuperable
             implements AccionAtomicaRecuperableTernaria
{
  protected ControladorRecuperable Control;
  private final boolean abort = false;
  private final boolean commit = true;

  private ExcepcionAccionAtomica eaa1, eaa2, eaa3;

public AccionRecuperable() // constructor
```

```
{
  Control = new ControladorRecuperable();
     // para recuperación
     eaa1 = new ExcepcionAccionAtomica();
     eaa2 = new ExcepcionAccionAtomica();
     eaa3 = new ExcepcionAccionAtomica();
}

class ControladorRecuperable {
  protected boolean primeraAqui, segundaAqui, terceraAqui;
  protected int todoHecho;
  protected int aSalir, requerido;
  protected int numeroParticipantes;
  private boolean committed = commit;

  ControladorRecuperable()
  {
     // para sincronización
     primeraAqui = false;
     segundaAqui = false;
     terceraAqui = false;
     todoHecho = 0;
     numeroParticipantes = 3;
     aSalir = numeroParticipantes;
     requerido = numeroParticipantes;
  }

  synchronized void primera() throws InterruptedException
  {
     while(primeraAqui) wait();
     primeraAqui = true;
  }

  synchronized void segunda() throws InterruptedException
  {
     while(segundaAqui) wait();
     segundaAqui = true;
  }

  synchronized void tercera() throws InterruptedException
  {
```

```
    while(terceraAqui) wait();
    terceraAqui = true;
}
synchronized void señalAbortar(Exception e)
{
 todoHecho = 0;
 ExcepcionAccionAtomica.causa = e;
 ExcepcionAccionAtomica.fueInterrumpido = true;
 // generar una AsynchronouslyInterruptedException
 // en todos los participantes
 eaa1.fire();
 eaa2.fire();
 eaa3.fire();
}
private void reset()
{
    primeraAqui = false;
    segundaAqui = false;
    terceraAqui = false;
    todoHecho = 0;
    aSalir = numeroParticipantes;
    requerido = numeroParticipantes;
    notifyAll();
}

synchronized void hecho() throws InterruptedException
{
    todoHecho++;
    if(todoHecho == requerido) {
       notifyAll();
    } else while(todoHecho != requerido) {
          wait();
          if(AtomicActionException.wasInterrupted) {
             todoHecho--;
             return;
          }
    }
    aSalir--;
    if(aSalir == 0) {
       reset();
    }
```

```
    }
    synchronized void limpiar(boolean abort)
    {
      if(abort) { committed = false; };
    }

    synchronized boolean esperarLimpiar() throws InterruptedException
    {
      todoHecho++;
      if(todoHecho == requerido) {
        notifyAll();
      } else while(todoHecho != requerido) {
          wait();
      }
      aSalir--;
      if(aSalir == 0) {
        reset();
      }
      return committed;
    };
};
public void roll() throws FalloAccionAtomica,
                          AsynchronouslyInterruptedException
{
boolean Ok;
// protocolo de entrada
// no AIE hasta dentro de la acción atómica
boolean hecho = false;
while(!hecho) {
  try {
    Control.primera();
    hecho = true;
  } catch (InterruptedException e) {
    // ignorar
  }
}
// lo siguiente define una sección de código
// interrumpible, y una rutina que será llamada
// si el código es interrumpido
Ok = eaal.doInterruptible
    (new Interruptible()
```

```
        {
            public void run(AsynchronouslyInterruptedException e)
                    throws AsynchronouslyInterruptedException
            {
                try {

                // realizar acción
                // si necesario llamar a e.disable() y e.enable()
                // para diferir AIE
                Control.hecho();
            }
            catch(Exception x) {
                if(x instanceOf AsynchronouslyInterruptedException)
                    ((AsynchronouslyInterruptedException) x).propagate();
                else
                    Control.señalAbortar(x);
                }
            }

            public void interruptAction(
                    AsynchronouslyInterruptedException e)
            {
                // acción no requerida
            }
        }
    );
if(!Ok) throw new FalloAccionAtomica();
    if(eaa1.fueInterrumpido) {
        try {
            // intentar la recuperación
            Control.limpiar(commit);
            if(Control.esperarLimpiar() != commit) {
                throw new FalloAccionAtomica();
            };
        }
        catch(Exception x) {
            throw new FalloAccionAtomica();
        }
    };
}
```

```
  public void rol2() throws FalloAccionAtomica,
                    AsynchronouslyInterruptedException
  {// similar a rol1 };

  public void rol3() throws FalloAccionAtomica,
                    AsynchronouslyInterruptedException
  {// similar a rol1 };
}
```

Resumen

La ejecución fiable de procesos es esencial en los sistemas embebidos de tiempo real que se utilizan en aplicaciones críticas. Cuando los procesos interactúan es necesario restringir las comunicaciones interproceso, de forma que, si es necesario, se puedan programar procedimientos de recuperación. En este capítulo se han abordado las acciones atómicas como un mecanismo de estructuración de aquellos programas que constan de muchas tareas, de forma que se facilite la recuperación de errores y el confinamiento de los daños.

Las acciones son atómicas si, en relación con otros procesos, pueden ser consideradas indivisibles e instantáneas, de modo que los efectos sobre el sistema sean como si fueran «entrelazadas», en oposición a concurrentes. Una acción atómica muestra unos límites bien definidos, y puede ser anidada. Los recursos utilizados en una acción atómica se asignan en una fase inicial de *crecimiento*, y se liberan en una fase siguiente de *reducción*, o al final de la acción (si la acción debe ser recuperable).

La sintaxis de una acción atómica se expresa en una sentencia de acción. La sentencia siguiente, ejecutada en el proceso P_1, indica que éste desea intervenir en una acción atómica junto con P_2 y P_3 .

```
action A with (P2, P3)do
     -- secuencia de sentencias
end A;
```

P_2 y P_3 deben ejecutar sentencias similares.

Una *conversación* es una acción atómica con un mecanismo de recuperación de errores hacia atrás (en forma de bloques de recuperación).

```
action A with ( P2, P3)do
  ensure <test aceptación>
  by
     -- módulo primario
  else by
```

```
    -- módulo alternativo
  else error
end A;
```

El estado del proceso se guarda en la entrada de la conversación. Mientras esté dentro de la conversación, el proceso sólo puede comunicarse con otros procesos activos de la conversación y con gestores de recursos generales. Para poder salir de la conversación, todos los procesos activos de la conversación deben haber pasado el test de aceptación. Si cualquiera de los procesos falla en su test de aceptación, todos los procesos restaurarán el estado guardado al comienzo de la conversación y ejecutarán sus módulos alternativos. Las conversaciones pueden ser anidadas, y si fallan todas las alternativas en la conversación interior, entonces la recuperación debe ser realizada en un nivel exterior.

La limitación de las conversaciones es que cuando fallan, todos los procesos son restaurados y todos entran en sus módulos alternativos. Para alcanzar el efecto deseado, los mismos procesos están forzados a comunicarse de nuevo; un proceso no puede desligarse de la conversación. A menudo, cuando un proceso no consigue su objetivo en un módulo primario comunicándose con un grupo de procesos, podría querer comunicarse con un nuevo grupo de procesos completamente distintos en su módulo secundario. Los diálogos y coloquios eliminan las limitaciones de las conversaciones.

A las acciones atómicas también se les puede añadir la recuperación de errores hacia adelante con manejadores de excepciones. Si un proceso genera una excepción, todos los procesos activos en la acción deben manejar la excepción.

```
action A with (P2, P3)do
  -- la acción
exception
  when exception_a =>
    -- secuencia de sentencias
  when others =>
    raise fallo_accion_atomica;
end A;
```

La resolución de excepciones generadas concurrentemente y las excepciones en acciones internas, son dos asuntos a resolver en esta aproximación.

Son pocos los lenguajes y sistemas operativos que soportan directamente las nociones de acción atómica y de acción atómica recuperable. Sin embargo, la mayoría de las primitivas de comunicación y sincronización permiten la programación de la propiedad de aislamiento de una acción. Para implementar una acción recuperable es necesario disponer de un mecanismo de notificación asíncrono. Éste puede seguir una semántica de reanudación (en cuyo caso se denomina mecanismo de manejo asíncrono de eventos) o una semántica de terminación (en cuyo caso se denomina transferencia asíncrona de control). POSIX soporta eventos asíncronos utilizando señales y un mecanismo de cancelación de hilos. Una señal puede ser manejada, bloqueada o ignorada. Java para tiempo real también permite eventos asíncronos.

Tanto Ada como Java para tiempo real proporcionan un modelo de terminación de transferencia asíncrona de control. El mecanismo de Ada se construye sobre la sentencia select. Por contra, el ATC de Java para tiempo real está integrado con los mecanismos de excepciones e interrupción de hilos. Estas aproximaciones de terminación, en combinación con las excepciones, permiten una implementación elegante de las acciones recuperables.

Lecturas complementarias

Anderson, T., y Lee, P. A. (1990), *Fault Tolerance Principles and Practice*, 2nd edn. Englewood Cliffs, NJ: Prentice Hall.

Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., y Turnbull, M. (2000), *The Real-Time Specification for Java*, Reading, MA: Addison-Wesley.

Lynch, N. A. (ed.) (1993), *Atomic Transactions*, San Mateo, CA: Morgan Kaufmann.

Lea, D. (1999), *Concurrent Programming in Java: Design Principles and Patterns*, Reading: Addison-Wesley.

Northcutt, J. D. (1987), *Mechanisms for Reliable Distributed Real-time Operating Systems: The Alpha Kernel*, New York: Academic Press.

Shrivastava, S. K., Mancini, L., y Randell, B. (1987), *On the Duality of Fault Tolerant Structures*, Lecture Notes in Computer Science, Volumen 309, pp. 19–37. Berlin: Springer-Verlag.

Ejercicios

10.1 Distinga entre una *acción* atómica y una *transacción* atómica. ¿Cuál es la relación entre una transacción atómica y una conversación?

10.2 Extienda la implementación con semáforo de la acción atómica que se da en la Sección 10.2.1, desde una interacción de dos procesos a una de tres.

10.3 Reescriba la implementación con monitor de la acción atómica que se da en la Sección 10.2.2, de forma que se permita a los dos procesos ser activos en la acción simultáneamente.

10.4 Reescriba el paquete Ada `Accion_X` de la Sección 10.2.3, de manera que sea un paquete de propósito general que controle una conversación de tres tareas (Pista: utilice genéricos).

10.5 ¿Puede extender la solución del Ejercicio 10.4 para tratar un número arbitrario de tareas que participan en la acción atómica?

10.6 ¿Cuáles serían las implicaciones de extender Ada para permitir que una tarea pueda generar una excepción en otra?

10.7 Compare y contraste la notificación asíncrona y el manejo de excepciones.

10.8 El siguiente código muestra una conversación sencilla entre dos procesos. Muéstrese esta construcción como un coloquio.

```
x,y,z : INTEGER;

PROCESS B;

PROCESS A;
BEGIN
   ...
   ACTION conversacion (B) do
      ENSURE A_test_aceptacion
      BY
         -- A_primario
         -- utiliza x,y
      ELSE BY
         -- A_secundario
         -- utiliza y,z
      ELSE
         ERROR
   END conversacion;
   ...
END A;

PROCESS B;
BEGIN
   ...
   ACTION conversacion (A) do
      ENSURE B_test_aceptacion
      BY
         -- B_primario
         -- utiliza x,y
      ELSE BY
         -- B_secundario
         -- utiliza y,z
      ELSE
         ERROR
   END conversacion;
   ...
END B;
```

10.9 En la Sección 10.8.2 se muestra la recuperación de errores hacia atrás y hacia adelante entre tres tareas. Muéstrese como pueden ser combinadas éstas en una única solución para ambas recuperaciones de errores entre las tres mismas tareas.

10.10 Actualice la solución dada en la Sección 10.8 de forma que se trate el problema del «desertor».

10.11 Considere los cuatro fragmentos de código siguientes.

```
-- Fragmento 1
select
   T.Call; -- llamada entry a la tarea T
   Indicador := A;
or
   delay 10.0;
   Indicador := B;
   -- código que tarda 2 segundos en ejecutarse
end select;

-- Fragmento 2
select
   T.Call; -- llamada entry a la tarea T
   Indicador := A;
else
   delay 10.0;
   Indicador := B;
   -- código que tarda 2 segundos en ejecutarse
end select;

-- Fragmento 3
select
   T.Call; -- llamada entry a la tarea T
   Indicador := A;
then abort
   delay 10.0;
   Indicador := B;
   -- código que tarda 2 segundos en ejecutarse
end select;

-- Fragmento 4
select
   delay 10.0;
   Indicador := A;
```

```
then abort
   T.Call; -- llamada entry a la tarea T
   Indicador := B;
   -- código que tarda 2 segundos en ejecutarse
end select;
```

Una cita con `T.Call` tarda 5 segundos en ejecutarse. ¿Cuál es el valor de la variable `Indicador` después de la ejecución de cada uno de los cuatro fragmentos, para los siguientes casos? Puede suponer que una sentencia de asignación de `Indicador` tiene un tiempo cero de ejecución.

(1) `T.Call` está disponible cuando se ejecuta el select.

(2) `T.Call` NO está disponible cuando se ejecuta el select, y tampoco en los siguientes 14 segundos.

(3) `T.Call` NO está disponible cuando se ejecuta el select, pero está disponible después de 2 segundos.

(4) `T.Call` NO está disponible cuando se ejecuta el select, pero está disponible después de 8 segundos.

10.12 Considere la siguiente especificación de paquete que proporciona un procedimiento de búsqueda de una cadena *única* de tamaño fijo en una parte de un array de caracteres. El procedimiento devuelve la posición del comienzo de la cadena, en caso de ser encontrada.

```
package Soporte_Busqueda is
   type Limites_Array is range 1 .. 1_000_000_000;
   type Array_Grande is array(Limites_Array) of Character;
   type Apuntador is access Array_Grande;

   type Cadena_Busqueda is new String(1..10);

   procedure Buscar(Pt: Apuntador;
                    Inf, Sup: Limites_Array;
                    Buscando : Cadena_Busqueda;
                    Encontrado : out Boolean;
                    Localizacion : out Limites_Array);
end Soporte_Busqueda;
```

Tres tareas desean realizar una búsqueda concurrente de la misma cadena en el array; se derivan de un tipo de tarea común:

```
task type Buscador(Array_Busqueda: Apuntador;
                   Inf, Sup: Limites_Array) is
```

```
   entry Encontrar(Buscando : Cadena_Busqueda);
   entry Obtener_Resultado(Localizacion : out Limites_Array);
end Buscador;
```

La cadena buscada es pasada por medio de un rendezvous inicial con las tareas.

Haga un borrador del tipo de tarea (y de cualquier otro objeto que se necesite), de forma que cuando una tarea encuentre la cadena, el resto de tareas sean *inmediatamente* informadas de la ubicación de la cadena para evitar búsquedas posteriores. Suponga que una de las tres tareas encontrará la `Cadena_Busqueda`. Además, todas las tareas deben estar preparadas para devolver el resultado a través de la entrada `Obtener_Resultado`.

10.13 Considere el siguiente fragmento de código Ada:

```
Error_1, Error_2 : exception;
task Vigilar;
task Señalizador;

protected Atc is
   entry Ir;
   procedure Señal;
private
   Indicador : Boolean := False;
end Atc;

protected body Atc is
   entry Ir when Flag is
   begin
      raise Error_1;
   end Ir;

   procedure Señal is
   begin
      Indicador := True;
   end Señal;
end Atc;

task body Vigilar is
begin
   ...
   select
      Atc.Ir;
   then abort
```

```
      -- código que tarda 100 milisegundos
      raise Error_2;
   end select;
   ...
exception
   when Error_1 =>
      Put_Line("Error_1 detectado");
   when Error_2 =>
      Put_Line("Error_2 detectado");
   when others =>
      Put_Line("Otros errores detectados");
end Vigilar;

task body Señalizador is
begin
   ...
   Atc.Señal;
   ...
end Señalizador;
```

Describa cuidadosamente las posibles ejecuciones de este fragmento de código, asumiendo que el cambio de contexto entre tareas puede ocurrir en cualquier momento.

10.14 Una aplicación POSIX particular consta de diversos procesos periódicos y de dos modos de operación: MODO A y MODO B. Uno de los procesos de la operación sólo opera en MODO A. Haga un diseño preliminar de este proceso asumiendo que cuando el sistema desea realizar un cambio de modo, envía una señal que indica el modo actual de operación a todos los procesos. También suponga que existe una rutina denominada `WAIT-NEXTPERIOD`, que suspenderá un proceso hasta que llegue su siguiente periodo de ejecución. Observe que el cambio de modo debería afectar a los procesos sólo al comienzo de cada periodo.

10.15 Muestre cómo puede utilizarse el modelo POO de Ada para generar acciones atómicas extensibles. Compare y contraste los modelos de transferencia asíncrona de control de Ada y Java.

10.16 Compare y contraste los modelos Ada y Java para transferencia asíncrona de control.

10.17 ¿Hasta qué punto se puede utilizar Java estándar para implementar acciones atómicas?

10.18 ¿Por qué están obsoletas las rutinas `resume()`, `stop()` y `suspend()` de Java?

10.19 Rehaga el Ejercicio 10.14 para Java para tiempo real.

10.20 Muestre cómo puede implementarse el modelo de terminación del manejo de excepciones de Ada en respuesta a la recepción de una señal POSIX.

Capítulo 11

Control de recursos

El Capítulo 10 consideraba el problema de conseguir cooperación entre pocesos fiables. Se ha señalado que se precisa también la coordinación entre los procesos si se pretende compartir el acceso a recursos escasos, como dispositivos externos, archivos, campos de datos compartidos, búferes y algoritmos ya codificados. Estos procesos se han denominado procesos **competitivos**. Gran parte del comportamiento lógico (es decir, no temporal) del software de tiempo real está relacionado con la asignación de recursos entre procesos competitivos. Aunque los procesos no se comuniquen directamente entre ellos para pasarse información sobre sus propias actividades, pueden comunicarse para coordinar el acceso a los recursos compartidos. Unos pocos recursos admiten un acceso concurrente ilimitado; sin embargo, la mayoría tienen restringido su uso de alguna manera.

Como se ha apuntado en el Capítulo 7, la implementación de entidades de recursos requiere alguna forma de agente de control. Si el agente de control es pasivo, entonces se dice que el recurso es **protegido** (o **sincronizado**). Alternativamente, si se requiere un agente activo para programar el nivel correcto de control, el controlador de recursos es llamado **servidor**.

En este capítulo se discute el problema del control fiable de los recursos, y se considera la asignación general de recursos entre procesos competitivos. Aunque tales procesos son independientes entre sí, el acto de asignación de recursos tiene implicaciones en la fiabilidad. En particular, el fallo de un proceso puede producir que un recurso asignado no llegue a estar disponible para los demás. Los procesos pueden pasar hambre de recursos si se permite que otros procesos los monopolicen. Además, los procesos pueden llegar a estar interbloqueados por mantener recursos de los que precisan otros procesos mientras, al mismo tiempo, ellos requieren más recursos.

11.1 Control de recursos y acciones atómicas

Aunque los procesos necesitan comunicarse y sincronizarse para realizar la asignación de recursos, no hay por qué hacerlo en forma de una acción atómica. Esto se debe a que la única infor-

mación intercambiada es la necesaria para alcanzar una compartición armoniosa de los recursos; no es posible intercambiar información arbitraria (Shrivastava y Banatre, 1978). Como resultado de esto, el controlador de recursos, en forma de recurso protegido o de servidor, es capaz de cualquier cambio de sus datos locales para asegurar la aceptabilidad global. Si *no* fuera este el caso, cuando un proceso al que se le ha asignado un recurso falla, sería necesario informar de su fallo a todos los procesos que se han comunicado con el controlador del recurso. Sin embargo, el código necesario para que un proceso concreto se comunique con el controlador debiera ser una acción atómica; así, ningún otro proceso en el sistema podría afectar seriamente al proceso cuando se esté asignando, o liberando, un recurso. Además, un gestor de recursos y un cliente podrían utilizar recuperación de errores hacia adelante o hacia atrás con cualquiera de las condiciones de error, anticipadas o no.

Aunque en el capítulo anterior se mostró que, en general, ningún lenguaje de tiempo real soporta directamente acciones atómicas, se puede conseguir el efecto de indivisibilidad mediante un uso cuidadoso de las primitivas de comunicación y sincronización disponibles.

11.2 Gestión de recursos

Las características de la modularidad (en particular la ocultación de información) dictan que los recursos deben encapsularse, y que sólo puede accederse a ellos a través de una interfaz procedimental de alto nivel; en Ada, por ejemplo, donde sea posible, debería usarse un paquete:

```
package Control_de_Recursos is

  type Recurso is limited private;
  function Asigna return Recurso;
  procedure Libera(Este_Recurso : Recurso);

private
  type Recurso is ...

end Control_de_Recursos;
```

Si el gestor de recursos es un servidor, entonces el cuerpo de `Control_de_Recursos` contendrá una tarea (o un objeto de acceso a un tipo tarea). Un recurso protegido utilizará un objeto protegido en el cuerpo del paquete.

En occam2, la única opción para un gestor de recursos es un proceso activo (es decir, todos los controladores de recursos programados como servidores activos). Dicho servidor debe ser instanciado desde un procedimiento (PROC) con parámetros de canal:

```
PROC gestor.recursos ([] CHAN OF cualquier peticion,
                      [] CHAN OF recurso asignado,
                      [] CHAN OF recurso liberado)
```

```
    ...
:
```

Con sincronización basada en monitor, como es el caso de las variables de condición y mutexes POSIX, o de las clases sincronizadas de Java, los recursos protegidos están encapsulados de modo natural dentro de un monitor. Por ejemplo, en Java:

```
public class Gestor_de_Recursos
{
  public synchronized Recurso asigna();
  public synchronized void libera(Recurso r);
}
```

Otras formas de sincronización, como espera ocupada y semáforos, no proporcionan el nivel apropiado de encapsulacion, y por tanto no se considerarán en este capítulo. Las regiones críticas condicionales (CCR) tampoco se evalúan explícitamente; los objetos protegidos son esencialmente una forma moderna de CCR.

La siguiente sección esta relacionada con una discusión sobre la potencia expresiva y la facilidad de uso (usabilidad) de varias aproximaciones a la gestión de recursos. Después de esta discusión, en una sección adicional sobre seguridad, se considerará cómo un controlador de recursos puede protegerse a sí mismo contra el mal uso.

11.3 Potencia expresiva y facilidad de uso

Bloom (1979) ha sugerido criterios para evaluar primitivas de sincronización en el contexto de la gestión de recursos. Este análisis forma la base de esta sección, donde se consideran la potencia expresiva y la facilidad de uso de las primitivas de sincronización para control de recursos. Las primitivas a evaluar son monitores/métodos sincronizados (con su uso de sincronización de condición), servidores (con una interfaz de paso de mensajes), y recursos protegidos (implementados como objetos protegidos). Los dos últimos utilizan guardas para la sincronización, y por ello, un aspecto de este análisis es una comparación entre **sincronización de condición** y **sincronización de evitación**.

Bloom utiliza la expresión «potencia expresiva» para indicar la capacidad de un lenguaje para expresar las restricciones de sincronización requeridas. La facilidad de uso de una primitiva de sincronización abarca:

- La facililidad con la que expresa cada una de esas restricciones de sincronización.
- La facilidad con la que se permite combinar las restricciones para conseguir esquemas de sincronización más complejos.

En el contexto del control de recursos, la información necesaria para expresar estas restriciones se puede clasificar (siguiendo a Bloom) como sigue:

- Tipo de petición de servicio.
- Orden en el que llegan las solicitudes.
- Estado del servidor y de todos los objetos que gestiona.
- Parámetros de una solicitud.

El conjunto original de restricciones de Bloom incluía la «historia del objeto» (esto es, la secuencia de todas las solicitudes previas). Aquí se supone que el estado del objeto se puede extender para incluir la información histórica. Además, se añade a la lista algo que Bloom no incluyó:

- La prioridad del cliente.

Una discusión completa de la prioridad de procesos se proporciona en el Capítulo 13. Para el propósito de este capítulo, se considera la prioridad como medida de la importancia del proceso.

Como se ha indicado anteriormente, hay, en general, dos aproximaciones lingüísticas a la restricción del acceso para un servicio (Liskov et al., 1986). La primera es la **espera condicional**: se aceptan todas las solicitudes, pero cualquier proceso cuya solicitud no se pueda atender actualmente se suspende en una cola. El monitor convencional tipifica esta aproximación: un proceso cuya solicitud no se pueda atender es encolado en una variable de condición, y reanudado cuando se pueda atender la solicitud. La segunda aproximación es **evitación**: las solicitudes no se aceptan a menos que puedan ser satisfechas. Las condiciones bajo las que se puede aceptar una solicitud de forma segura se expresan como una guarda en una acción de aceptación.

A continuación se considerará cada uno de los cinco criterios introducidos anteriormente para valorar las aproximaciones de sincronización.

11.3.1 Tipo de solicitud

La información sobre el tipo de operación solicitada se puede utilizar para dar preferencia a un tipo de solicitud sobre otra (por ejemplo, a las solicitudes de lectura sobre las solicitudes de escritura en una base de datos de tiempo real). Con la sincronización basada en monitor y los métodos sincronizados, las operaciones de lectura y escritura se podrían programar como procedimientos distintos, pero la semántica de un monitor generalmente implica que las llamadas pendientes en los procedimientos de monitor sean manejadas de forma arbitraria, por prioridad, o según el método FIFO. No es posible, por tanto, tratar primero las solicitudes de lectura, ni es factible saber cuántas llamadas pendientes hay para los procedimientos de monitor.

En Ada, se pueden representar diferentes tipos de solicitudes fácilmente con diferentes entradas (entry) en la tarea servidor u objeto protegido. Antes de conseguir el acceso a la entidad (en orden para la cola en una entry), de nuevo no hay forma de proporcionar preferencia sobre otras llamadas. Sin embargo, una forma natural de dar preferencia a solicitudes concretas una vez encoladas es a través de guardas que utilizan el atributo «count» de entradas. A continuación, se muestra un caso en el que las solicitudes `Actualiza` están planteadas para tener prioridad sobre las solicitudes `Modifica`:

```
protected Gestor_de_Recursos is
  entry Actualiza(...);
  entry Modifica(...);
  procedure Bloquea;
  procedure Desbloquea;
private
  Gestor_Bloqueado : Boolean := False;
  ...
end ;

protected body Gestor_de_Recursos is

  entry Actualiza(...) when not Gestor_Bloqueado is
  begin
    ...
  end Actualiza;

  entry Modifica(...) when not Gestor_Bloqueado and
                       Actualiza'Count = 0 is
  begin
    ...
  end Modifica;

  procedure Bloquea is
  begin
    Gestor_bloqueado := True;
  end Bloquea;

  procedure Desbloquea is
  begin
    Gestor_Bloqueado := False;
  end Desbloquea;

end Gestor_de_Recursos;
```

Con objetos protegidos, sólo las entradas pueden tener guardas; una vez que los procedimientos consigan el acceso al objeto, se ejecutarán inmediatamente, y por tanto no se puede dar preferencia a tipos de solicitudes particulares mediante procedimientos.

En el lenguaje occam2, cada tipo de petición está asociada con un grupo de canales diferente. Para elegir entre acciones alternativas, un proceso servidor debe usar una construcción de espera (wait) selectiva; afortunadamente, occam2 proporciona una forma de espera selectiva que da a cada alternativa una prioridad diferente. El servidor actualiza-modifica se estructura fácilmente:

```
WHILE TRUE
  PRI ALT
    ALT i = 0 FOR max
      actualiza[i] ? objeto
        -- actualiza el recurso
    ALT j = 0 FOR max
      modifica[j] ? objeto
        -- modifica el recurso
```

Recuerde que si cualquiera de las operaciones precisa devolver información al invocador, necesitará una doble interacción:

```
WHILE TRUE
  PRI ALT
    ALT i = 0 FOR max
      actualiza[i] ? objeto
    ALT j = 0 FOR max
      lee[j] ? Any
        -- extrae el componente adecuado del recurso
        salida[j] ! objeto
```

con el invocador haciendo las siguientes llamadas:

```
SEQ
  lee[MiCanalParaElServidor] ! Any
  salida[MiCanalParaElServidor] ? Resultado
```

El uso de `PRI ALT` proporciona un select estático determinista. En Ada, se dispone de una forma equivalente de sentencia select si se utiliza uno de los pragmas (`Queuing_Policy`) definidos en el Anexo de Tiempo Real. Este pragma define la política de encolado para las sentencias entry y select. Permite que se programe una selección determinista (de nuevo utilizando orden textual para indicar prioridad), pero requiere que todas las tareas llamadas tengan la misma prioridad (si no es así, entonces la prioridad de la tarea que llama tiene preferencia sobre la ordenación estática de las alternativas de select).

11.3.2 Orden de la solicitud

Ciertas restricciones de sincronización pueden formularse en relación con el orden en el que se han recibido las solicitudes (para asegurar la justicia, o para impedir la inanición de un cliente). Como ya se ha observado, los monitores suelen tratar las solicitudes en orden FIFO, y por tanto satisfacen inmediatamente este requisito. En Ada, las solicitudes pendientes del mismo tipo (llamadas a la misma entry) se pueden servir también de una forma FIFO si se ha elegido la política de encolado apropiada. Desafortunadamente, con esta política las solicitudes pendientes de tipos diferentes (por ejemplo, llamadas a diferentes entradas en una sentencia select) se sirven en un orden arbitrario fuera del control del programador. Por tanto, no hay forma de servir las peti-

ciones de diferentes tipos en el orden de llegada, a menos que se utilice una política FIFO y todos los clientes llamen inicialmente a un entry común de registro:

```
Server.Registra;
Server.Acción(...);
```

Pero esta doble llamada también tiene su dificultad (como se explicará en la Sección 11.3.4).

Con la estructura de nombrado uno a uno de occam2, es imposible que las solicitudes pendientes para una única construcción ALT sean tratadas según el orden de llegada. Un proceso podría tener un número de canales de servicio entre los que elegir, pero no podría detectar qué canal ha tenido un proceso esperando durante más tiempo. También se debiera indicar aquí que un proceso servidor occam2 debe conocer el número de posibles clientes que tiene; cada uno se debe asignar a un canal diferente. El modelo de Ada es mucho más aseguible para el paradigma cliente-servidor, ya que cualquier número de clientes puede invocar un entry, y cada entry puede tratarse en un orden FIFO. Esto es cierto para la tarea del servidor y los recursos protegidos (objetos).

11.3.3 Estado del servidor

Puede que algunas operaciones sólo se permitan cuando el servidor y los objetos que él administra están en un cierto estado. Por ejemplo, sólo se puede asignar un recurso si está libre, y sólo puede colocarse un item en un búfer si hay un *slot* vacío. Con la sincronización de evitación, las restricciones basadas en el estado se expresan como guardas, y las basadas en servidores con el posicionamiento de las sentencias accept (u operadores de recibir mensajes). Análogamente, los monitores son igualmente adecuados (utilizando variables de condición para implementar restricciones).

11.3.4 Parámetros de la petición

El orden de las operaciones en un servidor puede estar restringido por la información contenida en los parámetros de las peticiones. Dicha información se relaciona por lo general con la identidad, o (en el caso de recursos cuantificables como memoria) con el tamaño de la petición. Más adelante se muestra una estructura de monitor directa (en Java) para un controlador general de recursos. Una petición para un conjunto de recursos contiene un parámetro que indica el tamaño del conjunto solicitado. Si no están disponibles suficientes recursos, el invocador es suspendido; cuando se libera cualquier recurso, se despierta a todos los clientes suspendidos para ver si su solicitud puede ser satisfecha entonces.

```
public class GestorDeRecursos
{
  private final int maxRecursos = ...;
  private int recursosLibres;

  public GestorDeRecursos()
```

```
  {
    recursosLibres = maxRecursos;
  }

  public synchronized void asigna(int talla) throws
                        IntegerConstraintError, InterruptedException
  // veáse la Sección 6.3.2 para la definición de IntegerConstraintError
{
    if(talla > maxRecursos) throw new
                         IntegerConstraintError(1, maxRecursos, talla);
    while(talla > recursosLibres) wait();
    recursosLibres = recursosLibres - talla;
  }

  public synchronized void libera(int talla)
  {
    resourcesLibres = resourcesLibres + talla;
    notifyAll();
  }
}
```

Con la sincronización de evitación simple, las guardas sólo tienen acceso a las variables locales para el servidor (o el objeto protegido); los datos enviados en la llamada no pueden ser accedidos hasta que la llamada no ha sido aceptada. Es necesario, por tanto, construir una solicitud como una interacción doble. A continuación se discute cómo puede construirse, en Ada, un asignador de recursos que atienda este problema como un servidor. Las estructuras asociadas para un objeto protegido (en Ada) y un servidor (en occam2), se dejan como ejercicios para el lector (veánse los Ejecicios 11.3 y 11.4). Este ejemplo en Ada es una adaptación de otro dado por Burns et al. (1987).

Asignación de recursos y Ada: un ejemplo

Una forma de abordar el problema en Ada es asociar una familia de entradas con cada tipo de solicitud. Cada valor permisible del parámetro se traslada a un único índice de la familia, de forma que peticiones con parámetros diferentes se dirigen a entries diferentes. Obviamente, esto sólo es apropiado si el parámetro es de un tipo discreto.

La aproximación se ilustra en el siguiente paquete, que muestra cómo la falta de poder expresivo conduce a una estructura de programa compleja (es decir, de pobre facilidad de uso). De nuevo, considérese un ejemplo de asignación de recursos en el que el tamaño de la petición se da como un parámetro de la misma. Como se ha indicado anteriormente, la aproximación estándar es traducir los parámetros de la petición a los índices de una familia de entradas, de forma que solicitudes de diferentes tamaños son enviadas a diferentes entradas. Para intervalos pequeños, se puede utilizar la técnica descrita anteriormente para «tipo de petición», con las sentencias select enumerando las entries individuales de la familia. Sin embargo, para intervalos grandes se necesita una solución más complicada.

```
package Gestor_de_Recursos is
  Max_Recursos : constant Integer := ...;
  type Rango_de_Recursos is new Integer
  range 1..Max_Recursos;
  subtype Instancias_de_Recurso is Rango_de_Recursos range 1.. ...;
  procedure Asigna(Cantidad : Instancias_De_Recurso);
  procedure Libera(Cantidad : Instancias_De_Recurso);
end Gestor_de_Recursos;

package body Gestor_de_Recursos is

  task Gestor is
    entry Iscribe(Cantidad : Instancias_De_Recurso);
    entry Asigna(Instancias_De_Recurso); -- familia de entry
    entry Libera(Cantidad : Instancias_De_Recurso);
  end Gestor;

  procedure Asigna(Cantidad : Instancias_De_Recurso) is
  begin
    Gestor.Inscribe(Cantidad); -- Cantidad es un parámetro
    Gestor.Asigna(Cantidad); -- Cantidad es un índice en la familia
  end Asigna;

  procedure Libera(Cantidad : Instancias_De_Recurso) is
  begin
    Gestor.Libera(Cantidad);
  end Liberar;

  task body Gestor is
    Pendiente : array(Instancias_De_Recurso) of
      Natural := (others => 0);
    Recursos_Libres : Rango_de_Recursos := Rango_de_Recursos'Last;
    Asignado : Boolean;
  begin
    loop
      select
        accept Inscribe(Cantidad : Instancias_De_Recurso) do
          Pendiente(Cantidad) := Pendiente(Cantidad) + 1;
        end Inscribe;
      or
        accept Libera(Cantidad : Instancias_De_Recurso) do
          Recursos_Libres := Recursos_Libres + Cantidad;
```

```
            end Libera;
         end select;
         loop -- bucle principal
            loop -- accepta cualquier inscripción pendiente o libera, no espera
               select
                  accept Inscribe(Cantidad : Instancias_De_Recurso) do
                     Pendiente(Cantidad) := Pendiente(Cantidad) + 1;
                  end Inscribe;
               or
                  accept Libera(Cantidad : Instancias_De_Recurso) do
                     Recursos_Libres := Recursos_Libres + Cantidad;
                  end Libera;
               else
                  exit;
               end select;
            end loop;
            Asignado := False;

            for Pedido in reverse Instancias_De_Recurso loop
               if Pendiente(Pedido) > 0 and Recursos_Libres >= Pedido then
                  accept Asigna(Pedido);
                  Pendiente(Pedido) := Pendiente(Pedido) - 1;
                  Recursos_Libres := Recursos_Libres - Pedido;
                  Asignado := True;
                  exit; --bucle para aceptar nuevas inscripciones
               end if;
            end loop;
            exit when not Asignado;
         end loop;
      end loop;
   end Gestor;
end Gestor_de_Recursos;
```

El gestor da prioridad a las peticiones grandes. Con el fin de adquirir recursos, se requiere una interacción en dos etapas con el gestor: una petición de «inscripción», y una petición de «asignación». Esta doble interacción se oculta del usuario del recurso encapsulando el gestor en un paquete y proporcionando un único procedimiento (`Asigna`) para gestionar la petición.

Cuando no hay llamadas pendientes, el gestor espera una petición inscribe o una libera (para liberar recursos). Cuando llega una de inscripción, se anota su tamaño en el vector de peticiones pendientes. Se introduce, por tanto, un bucle para registrar todas las peticiones de inscripción y de liberación que pueden estar pendientes. Este bucle termina en el momento en el que no hay más peticiones.

Se utiliza después un bucle `for` para rastrear a través del vector de pendientes, y se acepta la petición de mayor tamaño que se puede acomodar. El bucle principal se repite en el caso de que una petición de mayor tamaño haya intentado inscribirse. Si no se puede servir ninguna solicitud, se sale del bucle principal, y el gestor espera a nuevas solicitudes.

La solución es complicada por la necesidad de una transacción de cita doble. También es costosa, ya que se precisa un entry para cada tamaño posible.

Se puede encontrar un sistema mucho más simple en el lenguaje SR (Andrews y Olsson, 1993). Aquí, se permiten las guardas para acceder a (es decir, referirse a) parámetros «in». Una petición de recurso se acepta sólo cuando el servidor o recurso protegido sabe que está en un estado en el que la petición puede ser satisfecha. No se necesita la doble llamada. Por ejemplo (utilizando código de tipo Ada, pero no el Ada válido, para un recurso protegido):

```
protected Control_de_Recursos is -- ADA NO VALIDO
  entry Asigna(Cantidad : Instancias_De_Recurso);
  procedure Libera(Cantidad : Instancias_De_Recurso);
private
  Recursos_Libres : Rango_de_Recursos := Max_Recursos;
end Control_de_Recursos;

protected body Control_de_Recursos is

  entry Asigna(Cantidad : Instancias_Del_Recurso)
    when Recursos_Libres >= Cantidad is     -- ADA NO VALIDO
  begin
    Recursos_Libres := Recursos_Libres - Cantidad;
  end Asigna;

  procedure Libera(Cantidad : Instancias_Del_Recurso) is
  begin
    Recursos_Libres := Recursos_Libres + Cantidad;
  end Liberar;

end Control_de_Recursos;
```

Aunque sintácticamente simple, este tipo de solución tiene la desventaja de que no está claro bajo qué circustancias necesitan ser reevaluadas las guardas o las barreras. Esto puede conducir a implementaciones ineficientes. Una aproximación alternativa es mantener las guardas sencillas pero incrementar la potencia expresiva del mecanismo de comunicación añadiendo una funcionalidad de **reencolado**. Esto se explicará en detalle en la Sección 11.4; sin embargo, en esencia, se permite que una llamada que ha sido aceptada (es decir, que tiene una evaluación guarda) sea reencolada en una entrada nueva (o incluso en la misma), donde debería pasar de nuevo una guarda. Lo siguiente también motiva la necesidad de reencolado.

Interacciones dobles y acciones atómicas

En las discusiones anteriores, se han dado ejemplos, tanto en Ada como occam2, en los que se requiere que el proceso haga una doble llamada en el servidor. Uno de los principales factores que necesita esta doble interacción es la falta de poder expresivo en la sincronización de evitación sencilla. Para programar procedimientos de control de recursos fiables, esta estructura debe ser implementada como una acción atómica. En occam2, la doble llamada

```
SEQ
  lee[MiCanalParaElServidor] ! Objecto
  salida[MiCanalParaElServidor] ? Resultado
```

forma una acción atómica en la que el cliente, habiendo enviado la petición, tiene garantizada la lectura del objeto (de forma similar, el servidor tiene garantizado el envío del dato). Desafortunadamente, con Ada, esta garantía no se puede dar (Wellings et al., 1984). Entre las dos llamadas, puede observarse un estado intermedio en el cliente desde fuera de la «acción atómica», es decir, después de «inscribe» pero antes de «asigna».

```
begin
  Gestor.Inscribe(Cantidad);
  Gestor.Asigna(Cantidad);
end;
```

Este estado es observable, en el sentido de que otra tarea puede abortar al cliente entre las dos llamadas y plantear algún problema al servidor:

- Si el servidor supone que el cliente hará la segunda llamada, el aborto dejará al servidor esperando la llamada (es decir, interbloqueado).
- Si el servidor se protege él mismo contra el aborto de un cliente (no esperando indefinidamente una segunda llamada), podría suponer que el cliente ha sido abortado cuando en realidad lo que ocurre es tan sólo que es lento haciendo la llamada; aquí se bloquearía al cliente erróneamente.

En el contexto del software de tiempo real, se plantean tres aproximaciones para tratar con el problema del aborto:

- Definir que la primitiva abort se aplique a una acción atómica y no a un proceso; la recuperación de errores hacia adelante o hacia atrás se puede utilizar cuando se comunica con el servidor.
- Suponer que abort sólo se utiliza en situaciones extremas, donde la ruptura de la acción atómica no tiene consecuencias.
- Intentar proteger al servidor del efecto del aborto del cliente.

En Ada, la tercera aproximación implica eliminar la necesidad de la doble llamada reencolando la primera llamada (en lugar de hacer que el cliente haga una segunda llamada). Como se ha indicado anteriormente, esto se explicará con más detalle en la Sección 11.4, después de la valoración final de los criterios que se han visto.

11.3.5 Prioridad del solicitante

El criterio final para evaluar las primitivas de sincronización para la gestión de recursos implica el uso de la prioridad del cliente. Si un conjunto de procesos son ejecutables, entonces el distribuidor (dispatcher) puede ordenar su ejecución de acuerdo con la prioridad. El distribudor no puede, sin embargo, tener ningún control sobre los procesos suspendidos que esperan los recursos. Por tanto, es necesario para el orden de las operaciones del gestor de recursos que también estén restringidas por las prioridades de los procesos cliente.

En Ada, Java para tiempo real y POSIX, es posible definir una política de encolado basada en la prioridad, pero, en general, en los lenguajes de programación concurrente los procesos son liberados de primitivas, como semáforos o variables de condición, de una manera arbitraria o FIFO; los monitores suelen seguir el método FIFO en sus entradas; las esperas selectivas suelen ser arbitrarias, o tienen una ordenación de prioridad textual estática. En este último caso, es posible programar clientes de forma que accedan a los recursos mediante una interfaz diferente (entry o canal). Para un rango de prioridad pequeño, esto es equivalente a la restricción *Request Type*. Para rangos de prioridad grandes, es igual que utilizar *Request Parameters*, y aquí se pueden utilizar las aproximaciones anteriores.

Aunque, a menudo, se describen los monitores bajo una disciplina de cola FIFO, esto no es realmente una propiedad esencial. De hecho, las implementaciones de monitores en POSIX y Java para tiempo real no sólo permiten colas con prioridad, sino que también (conceptualmente) mezclan la cola externa (de procesos esperando entrar en el monitor) y la interna (de procesos liberados por la recepción de una señal de una variable de condición), para obtener una única cola ordenada por prioridad. Aquí, los procesos de prioridad mas alta que esperan por conseguir el acceso al monitor tendrán preferencia sobre un proceso que se libere internamente.

11.3.6 Resumen de lo ya visto

En las discusiones anteriores, se han utilizado cinco requisitos para juzgar la adecuación de las estructuras del lenguaje actual para tratar con el control de los recursos generales. Los monitores, con su sincronización de condición, trabajan bien con los parámetros de la petición; las primitivas basadas en evitación necesitan ser aumentadas para tratar con parámetros, pero tienen el límite en el tipo de su solicitud.

Se debe considerar, sin embargo, que los requisitos mismos no son mutuamente consistentes. Podría haber un conflicto entre la prioridad del cliente y el orden de llegada de las solicitudes, o entre la operación solicitada y la prioridad del solicitante. Por ejemplo, en occam2 (con su espera selectiva determina), a una solicitud de actualización se le dió prioridad sobre una modificación por:

```
WHILE TRUE
  PRI ALT
  ALT i = 0 FOR max
    actualiza[i] ? objeto
      -- actualiza recurso
```

```
ALT j = 0 FOR max
   modifica[j] ? objeto
      -- modifica recurso
```

Podría decirse que si un invocador de `modifica` tiene una prioridad más alta que un invocador de `actualiza`, entonces la operación `modifica` debería tener lugar primero. No pueden satisfacerse ambos objetivos, y es preciso desarrollar primitivas de sincronización que permitan al programador tratar este conflicto de una forma estructurada.

Se ha mostrado que es necesario construir la interacción del cliente con el gestor de recursos como una acción atómica. La ausencia de primitivas de lenguaje para soportar acciones atómicas y la existencia de abortos y controles es de transferencia asíncrona lo dificulta: los procesos pueden ser eliminados asíncronamente de los monitores, o terminados entre dos llamadas relacionadas con un proceso servidor. El aborto se utiliza para eliminar procesos erróneos o redundantes. Si se construye un monitor correctamente, entonces se puede suponer que un proceso bribón (malintencionado) no producirá daño alguno mientras está en el monitor (dado que el conocimiento de que es un bribón sólo estaría disponible después de que hubiera entrado). De forma similar, no se podrá herir a un cliente bribón si está esperando para hacer una segunda llamada asíncrona.

Se deduce que hay situaciones bien especificadas en las que un proceso no sería abortado (para ser preciso, el efecto de abortar acabará ocurriendo, pero se aplaza). Esto conduce a la noción de **regiones de aborto aplazado**.

Java posee la noción de región ATC aplazada. Sin embargo, el método `destroy` de Java provoca problemas a los controladores de recursos, y por tanto no debiera ser usado. Argumentos semejantes se pueden aplicar a los métodos obsoletos de la clase `Thread` de Java.

Ada entiende que la ejecución de un objeto protegido (y de una cita), es una región de aborto aplazado. Sin embargo, las interacciones con los controladores de recursos suelen ocasionar potencialmente la suspensión del proceso, y tales llamadas no están permitidas desde dentro de un objeto protegido. Por ejemplo, una doble llamada a una tarea del servidor *no* podría ser encapsulada en un objeto protegido. Sin embargo, como se ha indicado anteriormente, el reencolado resuelve muchos de estos problemas de control de recursos; se describe en la sección siguiente.

11.4 La funcionalidad de reencolado

La discusión sobre los criterios de Bloom ha mostrado que la sincronización de evitación conduce a una forma más estructurada de programación de gestores de recursos, pero carece de poder expresivo cuando se compara con una sincronización de condición de bajo nivel. Un medio para mejorar la facilidad de uso de la sincronización de condición es añadir una funcionalidad de reencolado. El lenguaje Ada tiene dicha funcionalidad, y por tanto la discusión siguiente se centrará en el modelo del lenguaje.

El concepto que está detrás del reencolado es el de mover la tarea (que estaba tras una guarda o barrera) detrás de otra guarda. Considérese una persona (tarea) esperando para entrar en una

habitación (objeto protegido) que tiene una o más puertas (entry guardado) que dan acceso a la habitación. Una vez dentro, la persona puede ser arrojada (reencolada) de la habitación y ser colocada de nuevo detrás de una puerta (potencialmente cerrada).

Ada permite reencolados entre entradas de tarea y entradas de objetos protegidos. Un reencolado puede ser para un mismo entry, para otro entry en la misma unidad, o para otra unidad. Se permiten los reencolados de entradas de tarea (y viceversa). Sin embargo, el principal uso del reencolado es enviar la tarea invocadora a un entry diferente de la misma unidad en la que se ejecutó el reencolado.

En la Sección 10.8.2, se proporciona el código que implementó la recuperación del error hacia adelante, y se utiliza una doble interacción con `Controlador` (es decir, una llamada a `Limpia` seguida de una llamada a `Espera_Limpia`). Esto puede codificarse como una única llamada a `Limpia` reencolando en `Espera_Limpia` desde un entry (el `Espera_Limpia` llegaría entonces a tener un entry privado).

```
entry Limpia (Voto: Voto_T; Resultado : out Voto_T) when True is
begin
  if Voto = Abortado then
    Resultado_Final := Abortado;
  end if;
  requeue Espera_Limpia with abort;
end Limpia;
```

La funcionalidad «with abort» se describirá posteriormente. El efecto de la llamada es mover la tarea invocadora al entry `Espera_Limpia` (igual que si se hiciera una llamada desde el exterior). La ejecución de la sentencia requeue termina la ejecución del entry invocado originalmente.

El problema del control de recursos proporciona otro ejemplo ilustrativo de la aplicación del reencolado. En el algoritmo siguiente, una petición sin éxito se reencola en un entry privado (llamado `Asigna`) del objeto protegido. El invocador de este objeto ahora protegido hace una única llamada a `Solicita`. Cuando los recursos son liberados, se toma nota de cuántas tareas están en el entry `Asigna`. Estas tareas pueden reintentarlo, y obtener sus asignaciones o ser reencoladas en el mismo entry `Asigna`. La última tarea para reintentar abre la barrera:

```
type Rango_Petición is range 1..Max;

protected Controlador_de_Recursosis
  entry Solicita(R : out Recurso; Cantidad : Rango_Peticion);
  procedure Libera(R : Recurso; Cantidad : Rango_Petición);
private
  entry Asigna(R : out Recurso; Cantidad : Rango_Peticion);
  Liberado : Rango_Petición := Rango_Petición'Last;
  Nuevos_Recursos_Liberados : Boolean := False;
  A_Intentar : Natural := 0;
```

```
   ...
end Controlador_de_Recursos;

protected body Controlador_de_Recursosis
   entry Solicita(R : out Recurso; Cantidad : Rango_Peticion)
   when Liberado > 0 is
   begin
      if Cantidad <= Liberado then
         Liberado := Liberado - Cantidad;
         -- asigna
      else
         requeue Asigna;
      end if;
   end Solicita;

   entry Asigna(R : out Recurso; Cantidad : Rango_Petición)
      when Nuevos_Recursos_Liberados is
   begin
      A_Intentar := A_Intentar - 1;
      if To_Try = 0 then
         Nuevos_Recursos_Liberados := False;
      end if;
      if Cantidad <= Liberado then
         Liberado := Liberado - Cantidad;
         -- asigna
      else
         requeue Asigna;
      end if;
   end Asigna;

   procedure Libera(R : Recurso; Cantidad : Rango_Petición) is
   begin
      Liberado := Liberado + Cantidad;
      -- libera recursos
      if Asigna'Count > 0 then
         A_Intentar := Asigna'Count;
         Nuevos_Recursos_Liberados := True;
      end if;
   end Libera;
end Controlador_de_Recursos;
```

Téngase en cuenta que esto sólo funciona si la disciplina de encolado del entry `Asigna` es FIFO. Se deja como un ejercicio para el lector (Ejercicio 11.5) la tarea de desarrollar un algoritmo basado en prioridad.

Debe observarse que se puede derivar un algoritmo más eficiente si el objeto protegido registra la petición pendiente más pequeña. La barrera debería ser abierta entonces sólo en `Libera` (o permanecer abierta en `Asigna`) si `Liberado >= Menor`.

Sin embargo, incluso con este estilo de solución es difícil dar a ciertas peticiones otra prioridad distinta de FIFO o del orden de prioridad de la tarea. Como se ha indicado anteriormente, para programar este nivel de control se precisa una familia de entry. Sin embargo, con reencolado se puede proporcionar una estructura mejorada, que es algo como lo siguiente:

```
procedure Asigna(Cantidad : Instancias_De_Recurso) is
begin
  Gestor.Inscribe(Cantidad); -- tamaño es un parámetro
  Gestor.Asigna(Cantidad);   -- tamaño es un índice de familia
end Asigna;
```

con el servidor de tareas teniendo:

```
accept Inscribe(Cantidad : Instancias_De_Recurso) do
  Pendiente(Cantidad) := Pendiente(Cantidad) + 1;
end Inscribe_In;
```

la doble llamada puede hacerse atómica utilizando:

```
procedure Asigna(Cantidad : Instancias_De_Recurso) is
begin
  Gestor.Inscribe(Cantidad); -- Cantidad es un parámetro
end Asigna;
```

y

```
accept Inscribe(Cantidad : Instancias_De_Recurso) do
  Pendiente(Cantidad) := Pendiente(Cantidad) + 1;
  requeue Asigna(Cantidad);
end Inscribe;
```

La tarea servidor se puede hacer más segura teniendo su entry `Asigna` privada:

```
task Gestor is
  entry Inscribe(Cantidad : Instancias_De_Recurso);
  entry Libera(Cantidad : Instancias_De_Recurso);
private
  entry Asigna(Instancias_De_Recurso) (Cantidad : Instancias_De_Recurso);
end Gestor;
```

Este algoritmo no sólo es más directo que el proporcionado anteriormente, sino que tiene también la ventaja de que es más flexible para permitir que una tarea se elimine ella misma de una cola de entry (después de que el atributo count ha reconocido su presencia). Una vez que se ha

reeencolado una tarea no puede ser abortada o sujetarse a un plazo de tiempo en la llamada al entry (veáse la siguiente discusión).

11.4.1 Semántica del reencolado

Es importante apreciar que el reencolado no es una simple llamada. Si el procedimiento *P* llama al procedimiento *Q*, entonces, una vez que *Q* ha finalizado, el control se pasa de nuevo a *P*. Pero si el entry *X* reencola en el entry *Y*, el control no se pasa de nuevo a *X*. Una vez que se ha completado *Y*, el control pasa de nuevo al objeto que llamó a *X*. Por tanto, cuando un cuerpo entry o accept ejecuta un reencolado, ese cuerpo se completa.

Una consecuencia de esto es que, cuando se realiza un reencolado desde un objeto protegido a otro, entonces la exclusión mutua sobre el objeto original se abandona una vez que se ha encolado la tarea. Otras tareas que esperan para entrar en el primer objeto serán capaces de hacerlo. Sin embargo, un reencolado sobre el mismo objeto protegido mantendrá el bloqueo de exclusión mutua (si el entry objetivo está abierto).

El entry nombrado en una sentencia de reencolado (llamado entry **objetivo**) o no debe tener parámetros, o debe tener un perfil de parámetros equivalente (es decir, conforme con el tipo) al de la sentencia entry (o accept) desde la que se hace el reencolado. Por ejemplo, en el programa de control de recursos, los parámetros de `Asigna` son idénticos a los de `Reserva`. Por esta regla, no es necesario proporcionar los parámetros actuales con la llamada; al contrario, está prohibido hacerlo (en el caso de que el programador intente cambiarlos). Por tanto, si el entry objetivo no tiene parámetros, no se pasa información; si tiene parámetros, los parámetros correspondientes en la entidad que se está ejecutando el reencolado se reaplican (mapped across). Por esta regla, el algoritmo dado anteriormente para conseguir una ordenación FIFO sobre un número de diferentes entradas (es decir utilizando una sola entry de registro):

```
Servidor.Registro;
Servidor.Accion(...);
```

no puede programarse como una simple llamada utilizando reencolado (ya que todos los parámetros para las diferentes acciones pueden no ser idénticos).

Se puede utilizar una claúsula optativa, «with abort», con la sentencia requeue. Normalmente, cuando una tarea está en una cola entry, permanecerá allí hasta que sea servida, a menos que se haga una llamada a un entry temporizado (véase la Sección 12.4.2), o se elimine debido al uso de transferencia de control asíncrono (ATC) o de aborto. Una vez que la tarea se ha aceptado (o comienza a ejecutar el entry de un objeto protegido), se cancela el plazo de tiempo y se aplaza el efecto de cualquier intento de aborto hasta que la tarea sale del entry. Existe, sin embargo, la pregunta de qué debiera suceder con el reencolado. Considerando el tema del aborto, evidentemente se pueden establecer dos visiones:

- En el momento en que la primera llamada ha sido aceptada, el aborto debiera permanecer aplazado, de forma que a la tarea u objeto protegido se les pueda garantizar que la segunda llamada está allí.

- Si el reencolado coloca la tarea invocadora al final de una cola entry, entonces debiera ser posible el aborto.

Se puede argumentar de forma análoga con los plazos de tiempo. La sentencia de reencolado (requeue) permite programar ambas opciones; por defecto, no se permiten más plazos de tiempo o abortos, la adición de la claúsula «with abort» permite eliminar la tarea del segundo entry. Estas semánticas se necesitan, por ejemplo, con el algoritmo de recuperación del error hacia adelante, indicado al principio de esta sección.

El asunto real (en cuanto a decidir si utilizar o no «with abort») es si el objeto protegido (o la tarea servidor) ha reencolado a la tarea cliente que se espera que esté allí cuando se abre la guarda/barrera. Si el comportamiento correcto del objeto necesita la presencia de la tarea, entonces *no* se debería utilizar «with abort».

11.4.2 Reencolado en otras entries

Aunque lo normal suele ser el reencolado para la misma entidad, hay situaciones en las que se utiliza esta característica del lenguaje con total generalidad.

Considérese una situación en la que los recursos se controlan mediante una jerarquía de objetos. Por ejemplo, un router (encaminador) de red podría tener que elegir entre tres líneas de comunicación para pasar mensajes: `Linea_A` es la ruta preferida, pero si ésta se llega a sobrecargar se podría utilizar `Linea_B`; si, a su vez, ésta también se llegará a sobrecargar, se podría utilizar `Linea_C`. Cada línea está controlada por una tarea de servicio; es una entidad activa, ya que tiene que realizar operaciones de administración. Una unidad protegida actúa como interfaz para el router: decide cual de los tres canales se debiera utilizar, y después hace uso del reencolado para pasar la petición al servidor apropiado. La estructura de la solución se presenta en la Figura 11.1, y el programa se da a continuación:

```
type Linea_Id is (Linea_A, Linea_B, Linea_C);
type Linea_Estado is array (Linea_Id) of Boolean;

task type Controlador_Linea(Id : Linea_Id) is
  entry Solicita(...);
end Controlador_Linea;

protected Router is
  entry Envia(...); -- el mismo perfil de parámetros que Solicita
  procedure Sobrecargado(Linea : Linea_Id);
  procedure Limpia(Linea : Linea_Id);
private
  Ok : Linea_Estado := (others => True);
end Router;

La : Controlador_Linea(Linea_A);
```

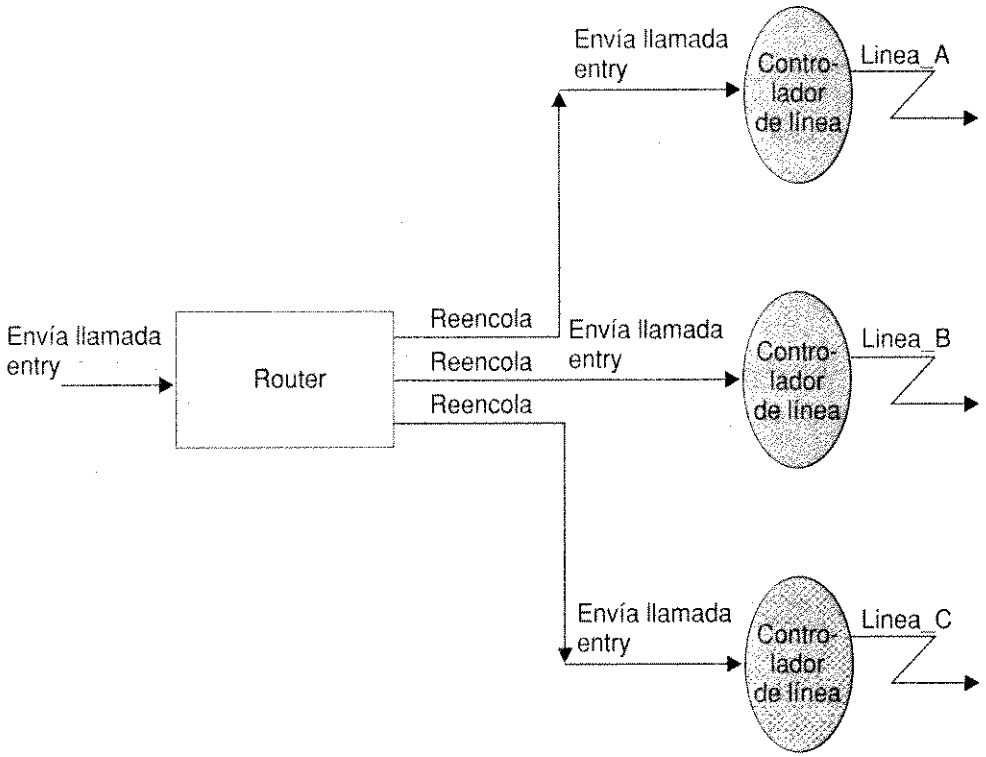


Figura 11.1. Un router de red.

```
Lb : Controlador_Linea(Linea_B);
Lc : Controlador_Linea(Linea_C);

task body Controlador_Linea is
   ...
begin
   loop
      select
         accept Solicita(...) do
            -- petición de servicio
         end Solicita;
      or
         terminate;
      end select;
      -- incluyendo posiblemente gestión
      Router.Sobrecargado(Id);
      -- or
      Router.Limpia(Id);
   end loop;
end Controlador_Linea;

protected body Router is
```

```
   entry Envia(...) when Ok(Linea_A) or
   Ok(Linea_B) or Ok(Linea_C) is
   begin
      if Ok(Linea_A) then
         requeue La.Solicita with abort;
      elsif Ok(Linea_B) then
         requeue Lb.Solicita with abort;
      else
         requeue Lc.Solicita with abort;
      end if;
   end Send;

   procedure Sobrecargado(Linea : Linea_Id) is
   begin
      Ok(Line) := False;
   end Sobrecargado;

   procedure Limpia(Line : Linea_Id) is
   begin
      Ok(Linea) := True;
   end Limpia;

end Router;
```

11.6 Nombrado asimétrico y seguridad

En lenguajes que tienen nombrado simétrico directo, un proceso servidor (o un recurso protegido) conoce siempre la identidad del proceso cliente con el que está tratando. Éste es también el caso con esquemas de nombrado indirecto basados en un intermediario uno a uno (como un canal de occam2). El nombrado asimétrico hace, sin embargo, que el servidor no sea consciente de la identidad del cliente. Ya se ha apuntado antes que esto tiene la ventaja de que se pueden escribir servidores de propósito general, pero puede conducir a una seguridad deficiente en la utilización de los recursos. En particular, un proceso servidor podría desear conocer la identidad del cliente invocador, de modo que:

- Pueda rechazarse una petición en aras de la prevención de interbloqueo (véase la Sección 11.7) o de la ecuanimidad (es decir, conceder la petición sería injusto para otros clientes).
- Pueda garantizarse que los recursos son liberados sólo por el proceso que los había obtenido anteriormente.

En CHILL, los procesos pueden tener una **instancia** o nombre asociado a ellos (veáse la Sección 9.6), que permite a un controlador de recursos (una **region** en CHILL) conocer la identidad de los procesos invocadores. Una funcionalidad semejante se puede programar en Ada utilizando

identificación de tareas (véase la Sección 7.3.6), o en Java utilizando `Thread.currentThread`. Sea un controlador sencillo de recursos, en Ada, que sólo gestiona un recurso.

```
protected Controlador is
  entry Asigna;
  procedure Libera;
private
  Asignado : Boolean := False;
  Propietario_Actual : Task_Id := Null_Task_Id;
end Controlador;


protected body Controlador is

  entry Asigna when not Asignado is
  begin
    Asignado := True;
    Propietario_Actual := Asignado'Caller;
  end Asigna;

  procedure Libera is
  begin
    if Current_Task /= Propietario_Actual then
      raise Invocador_Inválido; -- una excepción apropiada
    end if;
    Asignado := False;
    Propietario_Actual := Null_Task_Id;
  end Libera;
end Controlador;
```

Hay que tener en cuenta que con esta funcionalidad Ada, se identifica al invocador de un entry mediante el atributo `Caller`, mientras que el invocador de un procedimiento se obtiene de la función `Current_Task`. Las razones de esta diferencia no son importantes aquí –veáse Burns y Wellings (1998) para conocerlas–.

11.6 Utilización de los recursos

Cuando varios procesos que compiten o cooperan necesitan recursos, el modo normal de operación para ellos es: `petición` del recurso (esperando si es necesario); `utilización` del recurso; y después `liberación` del mismo. Un recurso se solicita normalmente para su uso en uno o dos modos de acceso, en los cuales puede haber acceso compartido o acceso exclusivo. El acceso compartido implica que el recurso se puede utilizar concurrentemente por más de un pro-

ceso (por ejemplo, un archivo de lectura). El acceso exclusivo implica que el acceso al recurso en cada instante esté permitido a un único proceso (por ejemplo un recurso físico, como una impresora de líneas). Algunos recursos se pueden utilizar en cualquier modo. En este caso, si un proceso pide el acceso al recurso en un modo compartido mientras está siendo accedido en modo exclusivo, entonces dicho proceso debe esperar. Si el recurso se estaba utilizando en modo compartido, entonces el proceso puede continuar. De modo semejante, si se solicita el acceso exclusivo a un recurso, en tonces el solicitante debe esperar hasta que los procesos que están accediendo actualmente a dicho recurso en modo compartido acaben.

Como los procesos pueden ser bloqueados cuando piden recursos, es primordial que no pidan los recursos hasta que no los necesiten. Además, una vez asignados, debieran liberarlos tan pronto como fuera posible. Si esto no se hace, las prestaciones del sistema pueden caer drásticamente, ya que los procesos esperan continuamente para obtener su parte de un recurso escaso. Lamentablemente, si los procesos liberan los recursos muy rápidamente y después fallan, pueden pasar información errónea a través del recurso. Por esta razón, el uso del recurso en dos fases, introducido en la Sección 10.1.1, debe modificarse, de modo que los recursos no sean liberados hasta que la acción haya finalizado. Cualquier procedimiento de recuperación para la acción debe liberar los recursos, si se realiza correctamente la recuperación, o eliminar cualquier efecto en los mismos. Esto último es necesario si no se puede conseguir la recuperación y debe reestablecerse el sistema en un estado seguro. Con la recuperación de errores hacia adelante, estas operaciones se pueden realizar con manejadores de excepciones. Con recuperación de errores hacia atrás, no es posible realizar estas operaciones sin algún soporte adicional del lenguaje (Shrivastava, 1979).

11.7 Interbloqueo

Con muchos procesos compitiendo por un número finito de recursos, se puede producir una situación en la que un proceso, P_1, tiene un único acceso a un recurso, R_1, mientras espera acceder a otro recurso, R_2. Si el proceso P_2 ya tiene un único acceso a R_2 y está esperando para acceder a R_1, entonces se produce **interbloqueo**, puesto que ambos procesos están esperando a que el otro libere un recurso. Con interbloqueo, todos los procesos afectados se suspenden indefinidamente. Una condición grave similar ocurre cuando se inhibe el avance de cierto conjunto de procesos, pero siguen en ejecución. Esta situación se conoce como **bloqueo activo** (livelock). Un ejemplo típico sería un conjunto de procesos interaccionando colocados en bucles de los que no pueden salir pero en los que no están haciendo ningún trabajo.

Otro posible problema se produce cuando varios procesos están intentando continuamente conseguir acceso exclusivo al mismo recurso; si la política de asignación del recurso no es justa, un proceso no podría conseguir nunca el acceso al recurso. Esto se llama **aplazamiento indefinido** o **inanición** (starvation; lockout) (véase el Capítulo 8). En sistemas concurrentes, **vivacidad** (liveness) significa que si se supone que algo puede ocurrir, eventualmente ocurrirá. El incumplimiento de la vivacidad produce interbloqueo, interbloqueo activo, o inanición. Hay que indicar que la vivacidad no es una condición tan fuerte como la ecuanimidad. Sin embargo, es difícil proporcionar una definición precisa y sencilla de ecuanimidad.

El resto de esta sección se centra en el interbloqueo y en cómo se puede prevenir, evitar, detectar y recuperar.

11.7.1 Condiciones necesarias para que ocurra interbloqueo

Hay cuatro condiciones necesarias que se deben dar para que ocurra interbloqueo:

- **Exclusión mutua**: sólo un proceso puede utilizar un recurso al mismo tiempo (es decir, el recurso no se puede compartir o está limitado su acceso concurrente).
- **Mantenimiento y espera**: debe haber procesos que mantengan recursos mientras esperan por otros.
- **No desalojo** (no apropiación): un recurso sólo puede ser liberado por un proceso voluntariamente.
- **Espera circular**: debe existir una cadena circular de procesos, de forma que cada proceso mantenga recursos que son solicitados por el siguiente proceso en la cadena.

11.7.2 Métodos para manejar el interbloqueo

Si un sistema de tiempo real debe ser fiable, tiene que tener en cuenta el problema del interbloqueo. Hay tres aproximaciones posibles:

- Prevención de interbloqueo.
- Evitación del interbloqueo.
- Detección y recuperación de interbloqueo.

Estos enfoques serán discutidos brevemente. Para una discusión exhaustiva se remite al lector a cualquier libro de sistemas operativos.

Prevención de interbloqueo

Se puede prevenir el interbloqueo garantizando que nunca ocurra al menos una de las cuatro condiciones requeridas para el mismo.

- *Exclusión mutua.* Si los recursos son compartibles no pueden estar implicados en un interbloqueo. Desafortunadamente, aunque unos pocos recursos permiten el acceso concurrente, la mayoría tiene restringido su uso de alguna manera.
- *Mantenimiento y espera.* Una forma simple para evitar el interbloqueo es requerir que los procesos o bien pidan los recursos antes de comenzar su ejecución, o bien los pidan en etapas durante su ejecución cuando no haya recursos asignados. Lamentablemente, esto tiende a ser ineficiente en la utilización de los recursos, y puede llevar a la inanición.

- *No desalojo*. Si la restricción de no desalojar a los recursos de los procesos se relaja, entonces puede prevenirse el interbloqueo. Hay varias aproximaciones, incluyendo la liberación de todos los recursos de un proceso, si éste intenta reservar otro recurso y falla, o robar un recurso si un proceso requiere un recurso que está asignado a otro proceso que está bloqueado (esperando por otro recurso). El proceso bloqueado está ahora esperando un recurso extra. La desventaja de esta aproximación es que necesita que el estado del recurso sea almacenado y reestablecido; en muchos casos, esto puede no ser posible.

- *Esperas circulares*. Para evitar esperas circulares, se puede imponer una ordenación lineal de todos los tipos de recursos. Cada tipo de recurso puede tener asignado un número de acuerdo con este orden. Supongamos que R es el conjunto de tipos de recursos; por tanto:

  $R = (R_1, R_2, ...R_n)$

  y la función F tiene un tipo de recurso y devuelve su posición en orden lineal. Si un proceso tiene un recurso R_J y solicita el recurso R_K, se considerará la petición sólo si

  $F(R_J) < F(R_K)$

  Alternativamente, un proceso pidiendo un recurso R_J deberá antes liberar todos los recursos R_I donde

  $F(R_I) < F(R_J)$:

  Téngase en cuenta que la función F debiera derivarse de acuerdo con el uso del recurso.

Otra aproximación para la prevención es analizar formalmente el programa, de manera que se pueda verificar la imposibilidad de esperas circulares. La facilidad para analizar los programas depende mucho del modelo de concurrencia empleado. Las primitivas de sincronización de bajo nivel son extremadamente difíciles de examinar de esta manera; las estructuras basadas en mensajes están, en comparación, mucho más preparadas para la aplicación de reglas de prueba formales. Occam2 está particularmente bien diseñado para este fin. Su semántica se ha especificado formalmente, y gran parte de la teoría sobre los programas libres de interbloqueo que se ha desarrollado para CSP es igualmente aplicable a occam2 (Hoare, 1985).

Evitación de interbloqueo

Si se tiene más información sobre el patrón de utilización de los recursos, entonces es posible construir un algoritmo que permita que ocurran las cuatro condiciones necesarias para el interbloqueo, pero que asegurará también que el sistema nunca entre en un estado de interbloqueo. Un algoritmo para evitar el interbloqueo examinará dinámicamente el estado de asignación del recurso y ejecutará la acción que asegure que el sistema nunca pueda entrar en interbloqueo. Sin embargo, no es suficiente preguntar únicamente si el siguiente estado es interbloqueo, porque si lo es, puede que no sea posible tomar ninguna acción alternativa que evite esta condición. Más bien, es necesario preguntar si el sistema está en un estado seguro. El estado de asignación de recursos está dado por:

```
estado = número de recursos disponibles, número reservado y
         máxima demanda de recursos de cada proceso
```

El sistema es **seguro** si puede asignar recursos a cada proceso (hasta el máximo que precisan) en algún orden, y seguir impidiendo el interbloqueo. Por ejemplo, consideremos un sistema de adquisición de datos que requiere acceder al almacén secundario masivo. Tiene 12 unidades de cinta, y consta de tres procesos: P_0, que precisa un máximo de 10 unidades de cinta; P_1, que precisa 4; y P_2, que precisa 9. Supongamos que en el instante *Ti* P_0 tiene 5, P_1 tiene 2, y P_2 tiene 2 (por tanto, 3 unidades de cinta están disponibles), como se ve en la Tabla 11.1.

Tabla 11.1. Estado seguro.

Proceso	Tiene	Necesita
P_0	5	10
P_1	2	4
P_2	2	9
Total asignado = 9 Unidades de cinta disponibles = 3		

Esto es un estado seguro, porque hay una secuencia (P_1; P_0; P_2) que permitirá que todos los procesos terminen. Si en el instante $T_i + 1$, P_2 pide una unidad de cinta y se le asigna, el estado será: P_0 tiene 5, P_1 tiene 2, P_2 tiene 3, y hay 2 disponibles, como se ve en la Tabla 11.2.

Tabla 11.2. Estado no seguro.

Proceso	Tiene	Necesita
P_0	5	10
P_1	2	4
P_2	3	9
Total asignado = 10 Unidades de cinta disponibles = 2		

Éste no es un estado seguro; con las unidades de cinta disponibles sólo puede terminar P_1, dejando: P_0 con 5 y P_2 con 3, pero sólo 4 disponibles. Ni P_0 ni P_2 pueden, por tanto, ser satisfechos, por lo que el sistema puede entrar en una situación de interbloqueo. La petición de P_2 en $T_i + 1$ debiera ser bloqueada por el sistema.

Obviamente, el interbloqueo es un estado inseguro. Un sistema que está inseguro puede conducir a un interbloqueo o *puede no hacerlo*. Sin embargo, si el sistema permanece seguro *no sufrirá interbloqueo*. Con más de un tipo de recursos, el algoritmo de evitación es más complicado. Uno bastante utilizado es el algoritmo del **banquero**. En la mayoría de los textos de sistemas operativos se puede encontrar una explicación de dicho algoritmo.

Detección y recuperación del interbloqueo

En muchos sistemas concurrentes de propósito general, el uso de la asignación de recursos es desconocido *a priori*. Incluso si es conocido, el coste de la evitación del interbloqueo es, a menudo, prohibitivo. Consecuentemente, muchos de estos sistemas ignorarán los problemas de interbloqueo hasta que entren en un estado de este tipo. Entonces tomarán alguna acción correctiva. Desgraciadamente, si hay muchos procesos, puede ser bastante difícil detectar el interbloqueo. Una aproximación es utilizar **grafos de asignación de recursos** (llamados a veces **grafos de dependencia de recursos**).

Para detectar si un grupo de procesos está interbloqueado, se necesita que el sistema de asignación de recursos sea consciente de qué recursos ya han sido asignados para dichos procesos, y qué procesos están bloqueados esperando recursos que ya han sido asignados. Si se tiene esta información, se puede construir un grafo de asignación de recursos.

Utilizando el grafo, el sistema puede determinar si existe intebloqueo. Esto se hace examinando incialmente aquellos procesos que no están bloqueados. Después se reduce el grafo eliminando dichos procesos, con la suposición de que si los procesos continúan su ejecución, terminarán en algún momento.

En general, si no aparecen ciclos en un grafo de asignación de recursos, no se produce interbloqueo. Sin embargo, si hay un ciclo en el sistema puede *o no* producirse interbloqueo, dependiendo de los otros procesos del grafo. Por tanto, se pueden utilizar los grafos de asignación de recursos para prevenir el interbloqueo. Cada vez que se pide un recurso, se actualiza el grafo. Si existe un ciclo, entonces puede haber interbloqueo, y por tanto la petición debiera ser rechazada.

Si se puede detectar el interbloqueo, entonces puede tener lugar la recuperación eliminando la exclusión mutua de uno o más recursos, abortando uno o más procesos, o desalojando algunos recursos de uno o más procesos interbloqueados. Eliminar la exclusión mutua, aunque es fácil de conseguir, puede dejar el sistema en un estado inconsistente; si un recurso era compartible, se debe hacer compartible. Abortar uno o más procesos es una medida muy drástica y, de nuevo, puede dejar al sistema en un estado inconsistente. Si se adopta cualquiera de estas opciones, se debe tener en cuenta el coste de la acción, que puede incluir consideraciones sobre la prioridad del proceso, sobre cuánto tiempo llevaba ejecutándose, sobre cuántos recursos adicionales se necesitan, etc.

El desalojo necesita que se seleccionen una o más víctimas. La selección debiera tener en cuenta la prioridad de los procesos, si los recursos son fáciles de desalojar, y lo próximos que estén los procesos a su finalización. Una vez seleccionados, el curso de la acción dependerá de la forma en la que se utilice la recuperación de errores. Con recuperación de errores hacia atrás, el proceso es reiniciado en un punto de recuperación anterior al de la asignación del recurso. Con recuperación de errores hacia adelante, se genera una excepción adecuada en la víctima sobre la que se debe tomar alguna acción correctiva.

Evidentemente, se debe tener cuidado para asegurar que no se desaloja permanentemente el mismo proceso, evitando así que padezca hambre de recursos. La solución más común a este

problema es incluir, como parte de un factor de costo, el número de veces que un proceso ha sido desalojado.

En esta sección se han revisado brevemente las posibles aproximaciones que un sistema puede adoptar para evitar o tratar con el interbloqueo. La mayoría de ellas son muy costosas de implementar, y pueden ser prohibitivas en un sistema de tiempo real. Esto es particularmente cierto si el sistema es distribuido. Consecuentemente, algunos sistemas utilizan plazos de tiempo sencillos en las peticiones de asignación de recursos. Si el plazo de tiempo expira, entonces los procesos pueden suponer que existe un interbloqueo potencial y tomar algunas opciones alternativas de acción. En el Capítulo 13, se introdujo una aproximación a la compartición de recursos basada en la planificación con prioridad, lo cual tiene la ventaja de estar libre de interbloqueos (al menos en sistemas con un único procesador).

Resumen

En cada sistema computacional, hay muchos procesos compitiendo por un conjunto limitado de recursos. Se necesitan algoritmos que gestionen los procedimientos de asignación/desasignación (el mecanismo de asignación de recursos), y que garanticen que los recursos se asignan a los procesos de acuerdo con un comportamiento predefinido (la política de asignación de recursos). Estos algoritmos son también los responsables de garantizar que los procesos no puedan interbloquearse mientras están esperando para satisfacer sus solicitudes de asignación de recursos.

Compartir los recursos entre procesos requiere que éstos se comuniquen y sincronicen. Por tanto, es esencial que las funcionalidades de sincronización proporcionadas por un lenguaje de tiempo real tengan potencia expresiva suficiente como para permitir que se especifiquen un amplio rango de restricciones de sincronización. Estas restricciones pueden clasificarse como sigue; la planificación de recursos debe considerar:

- El tipo de servicio solicitado.
- El orden en que llegan las solicitudes.
- El estado del servidor y de los objetos que gestiona.
- Los parámetros de una solicitud.
- La prioridad del cliente.

Los monitores (con las condiciones de sincronización) manejan bien los parámetros de solicitud, y la sincronización de evitación en servidores basados en mensajes u objetos protegidos se las arregla adecuadamente con los tipos de solicitudes.

Donde hay potencia expresiva suficiente, lo procesos son forzados, a menudo, a una doble interacción con un gestor de recursos. Esto se debe realizar como una acción atómica; de otro modo, es posible que el proceso cliente sea abortado después de la primera interacción y antes de la segunda. Si existe esta posibilidad, resulta muy difícil programar gestores de recursos fiables.

Un medio para ampliar la potencia expresiva de las guardas es permitir el reencolado. Esta funcionalidad permite que la sincronización de evitación sea tan efectiva como la de condición.

Un requisito fundamental de la gestión de recursos es la provisión de uso sin interbloqueo. Se deben dar cuatro condiciones necesarias para que pueda ocurrir el interbloqueo:

- Exclusión mutua.
- Mantenimiento y espera.
- No desalojo, y espera circular.

Si un sistema de tiempo real debe ser fiable, tiene que considerar el tema del interbloqueo. Hay tres aproximaciones posibles:

- Prevención de interbloqueo: garantizando que nunca ocurre al menos una de las cuatro condiciones.
- Evitación del interbloqueo: utilizando información del patrón de uso de los recursos para construir un algoritmo que permita que se den las cuatro condiciones, pero que también garantice que el sistema nunca entre en el estado de interbloqueo.
- Detección y recuperación del interbloqueo: permitiendo que ocurra el interbloqueo (y sea detectado) y recuperándose de él mediante: ruptura de la exclusión mutua en uno o más recursos, abortando uno o más procesos, o desalojando algunos recursos de uno o más procesos interbloqueados.

Lecturas complementarias

Bloom, T. (1979), Evaluating Synchronization Mechanisms, *Proceedings of the Seventh Symposium on Operating Systems Principles*, ACM, pp. 24–32.

Silberschatz, A., y Galvin, P. A. (1998), *Operating System Concepts*, New York: John Wiley & Sons.

Ejercicios

11.1 El siguiente controlador de recursos intenta asociar prioridades con solicitudes.

```
type Nivel is (Urgente, Medio, Bajo);

task Controlador is
   entry Solicita(Nivel) (D:Data);
end Controlador;
```

```
task body Controlador is
   ...
begin
   loop
   ...
   select
      accept Solicita(Urgente)(D:Data) do
         ...
      end;
   or
      when Solicita(Urgente)'Count = 0 =>
         accept Solicita(Medio)(D:Data) do
            ...
         end;
   or
      when Solicita(Urgente)'Count = 0 and
           Solicita(Medio)'Count = 0 =>
         accept Solicita(Bajo)(D:Data) do
            ...
         end;
      end select
      ...
   end loop;
end Controlador;
```

Explique esta solución, e indique bajo qué condiciones fallará. ¿Por qué no sería aconsejable extender la solución anterior para arreglárselas con una prioridad numérica que falla en el rango de 0 a 1.000? Esboce una solución alternativa que funcione con rangos de prioridad más grandes. Puede suponer que las tareas invocadoras emiten llamadas de entries sencillas y no son abortadas.

11.2 Muestre cómo se puede programar en Java (que no tiene una funcionalidad de reencolado) el gestor de recurso dado en la Sección 11.3.1.

11.3 Muestre cómo se puede programar el gestor de recurso Ada dado en la Sección 11.3.4 utilizando objetos protegidos.

11.4 Muestre cómo se puede programar en occam2 el gestor de recursos Ada dado en la Sección 11.3.4.

11.5 Muestre cómo el gestor de recursos Ada, proporcionado en la Sección 11.4, puede actualizarse para permitir que clientes con prioridad alta tengan preferencia sobre los de baja prioridad.

11.6 Muestre cómo se pueden utilizar los mutexes y las variables de condición de POSIX para implementar un controlador de recursos en el que se pueden asignar y liberar varios re-

cursos idénticos. Los clientes pueden solicitar y liberar uno o más recursos utilizando la siguiente interfaz:

```
typedef struct {
   ... /* rellenar */
   } recurso;

void asigna(int cantidad, recurso *R);
void desasigna(int cantidad, recurso *R);
void inicializa(recurso *R);
```

11.7 Considere un sistema que tiene cinco procesos (P_1, P_2 ,..., P_5) y 7 tipos de recursos (R_1, R_2, ..., R_7). Hay una instancia de los recursos 2, 5 y 7, y dos instancias de los recursos 1, 3, 4 y 6. Elproceso 1 tiene asignada una instancia de R_1 y requiere una instancia de R_7. El proceso 2 tiene asignada una instancia de R_1, R_2 y R_3 y precisa R_5. El proceso 3 tiene asignada una instancia de R_3 y R_4 y precisa R_1. El proceso 4 tiene asignado R_4 y R_5 y precisa R_2. El proceso 5 tiene asignado R_7.

¿Está este sistema en interbloqueo? Dé sus razones.

11.8 Un sistema que está en un estado descrito en la Tabla 11.3, ¿está en un estado seguro o inseguro? Dé sus razones.

11.9 Un sistema que está en un estado inseguro no ha de llegar necesariamente al interbloqueo. Explique por qué esto es así. Dé un ejemplo de un estado inseguro y muestre cómo los procesos podrían finalizar sin que ocurra interbloqueo.

11.10 Muestre cómo se puede programar en Java el ejemplo de router de red dado en la Sección 11.4.2.

Tabla 11.3. Un estado del sistema para el Ejercicio 11.8.

	Carga actual	**Necesidades máximas**
Proceso 1	2	12
Proceso 2	4	10
Proceso 3	2	5
Proceso 4	0	5
Proceso 5	2	4
Proceso 6	1	2
Proceso 7	5	13
Unidades disponibles = 1		

Capítulo 12

Capacidades de tiempo real

En el Capítulo 1 se comentó que un lenguaje de programación de sistemas embebidos requiere capacidades para gestionar el tiempo real. De hecho, la expresión «tiempo real» se ha utilizado como sinónimo de esta clase de sistemas. Dada la importancia del tiempo en la mayoría de los sistemas embebidos, parece extraño que su estudio haya sido pospuesto hasta el Capítulo 12. Sin embargo, las capacidades para el control en tiempo real suelen construirse sobre el modelo de concurrencia de los lenguajes, y por ello resultaba preciso tratar previamente dicha cuestión.

La introducción de la noción del tiempo en los lenguajes de programación se puede describir en relación con tres elementos independientes:

(1) La interfaz con el «tiempo»; por ejemplo: acceso a relojes de forma que se pueda medir el paso del tiempo, retardo de procesos hasta un cierto instante de tiempo futuro, y programación de tiempos límite de espera (timeouts), de forma que se pueda reconocer y tratar la no ocurrencia de algún evento.

(2) La representación de requisitos temporales; por ejemplo, especificar las tasas de ejecución y los tiempos límite (deadlines).

(3) La satisfacción de los requisitos temporales.

Este capítulo trata básicamente de los dos primeros temas, aunque comenzará con la discusión de la noción de tiempo. El Capítulo 13 considera las formas de implementar sistemas de modo que se pueda predecir su comportamiento en el peor caso, y por lo tanto se puedan ratificar los requisitos temporales.

La noción de tiempo

Nuestra experiencia diaria está tan intrínsecamente relacionada con las nociones de pasado, presente y futuro, que resulta sorprendente que la pregunta «¿qué es el tiempo?» esté aún sin resol-

ver en gran medida. Filósofos, matemáticos, físicos y, más recientemente, ingenieros han estudiado el «tiempo» minuciosamente. Aun así, todavía no existe un consenso definitivo sobre la teoría del tiempo. Como sostiene San Agustín:

> ¿Qué es, entonces, el tiempo? Si nadie me lo pregunta, sé lo que es. Si quisiera explicárselo a quién me lo pregunta, no lo sabría.

Una cuestión clave en la discusión filosófica sobre el tiempo puede enunciarse como sigue: ¿existimos en el tiempo, o es el tiempo parte de nuestra existencia? Las dos escuelas principales del pensamiento son el reduccionismo y el platonismo. Ambas coinciden en que la historia humana (y la biológica) está compuesta de eventos, y que estos eventos están ordenados. Los platónicos creen que el tiempo es una propiedad fundamental de la naturaleza: es continuo, no acaba y no tiene inicio, «y en su ser está el que presente estas propiedades». Nuestra noción del tiempo se deriva de la relación de los eventos históricos dentro de esta referencia temporal externa.

Los reduccionistas, sin embargo, no consideran esta referencia externa. El tiempo histórico, que está formado a partir de los eventos pasados, es la única noción de tiempo que trasciende. Asumiendo que ciertos eventos son «regulares» (por ejemplo, el amanecer, el solsticio de invierno, la vibración de los átomos, etc.), pueden inventar una referencia temporal útil que permita medir el paso del tiempo. Pero esta referencia temporal es una construcción, no algo dado.

Una consecuencia del punto de vista reduccionista es que el tiempo no puede progresar sin el cambio. Si el universo nació en un *Big Bang*, entonces esto representa el primer evento histórico, y por lo tanto el tiempo comenzó junto con el «espacio» en esta primera etapa. Para los platonistas, el *Big Bang* es sólo un evento en una linea ilimitada de tiempo.

Para largas distancias, Einstein demostró que los efectos de la relatividad deforman no sólo el tiempo, sino también la ordenación temporal de los eventos. Según la especial Teoría de la Relatividad, el observador de los eventos impone el marco de referencias. Un observador puede situar el evento A antes del evento B, mientras que otro observador (en un marco de referencias distinto) puede observar un orden inverso. Debido a estas dificultades con la ordenación temporal, Einstein introdujo el concepto de **orden causal**. El evento A puede causar el evento B si todos los posibles observadores ven que el evento A ocurre primero. Otro modo de definir la causalidad es postular la existencia de una señal que viaja, a una velocidad no mayor que la de la luz, desde el evento A al evento B.

Las diferencias de estas concepciones del tiempo se ilustran bien en la noción de eventos simultáneos. Para los platonistas, los eventos son simultáneos si ocurren «al mismo tiempo». Para los reduccionistas, los eventos simultáneos «suceden juntos». Según la Teoría de la Relatividad, los eventos simultáneos son aquéllos en los que no existe una relación causal.

Desde el punto de vista matemático, existen varias topologías diferentes para el tiempo. La más común proviene de la ecuación que considera el paso del tiempo como una recta «real». Consecuentemente, el tiempo es lineal, transitivo, irreflexivo y denso:

- Linealidad: $\forall x, y : x < y \vee y < x \vee x = y$
- Transitividad: $\forall x, y, z : (x < y \wedge y < z) \Rightarrow x < z$

- Irreflexividad: $\forall x : no\ (x < x)$
- Densidad: $\forall x; y : x < y \Rightarrow \exists z : (x < z < y)$

La perspectiva del ingeniero puede obviar las cuestiones filosóficas sobre el tiempo. Un sistema embebido de tiempo real necesita coordinar su ejecución con el «tiempo» de su entorno. El termino «real» se utiliza para marcar una distinción con el tiempo del computador, y es real porque es externo. El hecho de que este marco de referencia externo sea una construcción reduccionista o una aproximación al marco temporal «absoluto» de los platonistas no es importante. Más aún, en la mayoría de las aplicaciones, también se pueden ignorar los efectos relativistas. En relación con la topología matemática de los sistemas de tiempo real, existen opiniones contradictorias sobre si debiera verse el tiempo como discreto o como denso. Debido a que los computadores trabajan en tiempo discreto, existe cierta ventaja en construir modelos computacionales basados en el tiempo discreto. Se pueden encontrar argumentos en el otro sentido, avalados por la experiencia de otras ramas de la ingeniería; en ellas, se emplean modelos de tiempo denso con buenos resultados. Los intentos de integrar ambos enfoques han conducido a una tercera aproximación: los sistemas híbridos. Si, por consenso, se conviene en que ciertos eventos se comportan regularmente, entonces será posible definir una medida estándar del tiempo. En el pasado se establecieron diversos estándares. La Tabla 12.1 muestra una breve descripción de alguno de los más importantes. Esta descripción ha sido tomada de Hoogeboom y Halang (1992).

12.2 Acceso a un reloj

Si un programa va a interactuar de una forma significativa con el marco temporal de su entorno, debería tener acceso a algún método llamado «dime la hora», o al menos, debería tener alguna forma de medir el paso del tiempo. Esto puede hacerse de dos formas distintas:

- Accediendo directamente al marco temporal del entorno.
- Mediante un reloj hardware externo que dé una aproximación adecuada del paso del tiempo en el entorno.

La primera vía no es la más común, aunque se puede lograr de varias formas. La más simple es que el entorno proporcione una interrupción regular sincronizada internamente con el reloj. En el otro extremo, el sistema (o en su lugar cada nodo de un sistema distribuido) puede sintonizar, con un radioreceptor, alguna de las señales de tiempo internacionales. Por ejemplo, en la República Alemana, se dispone de emisores para señales UTC e IAT. El sistema de navegación mundial (GPS) también proporciona un servicio UTC.

Desde la perspectiva del programador, el acceso al tiempo puede darse mediante una primitiva de reloj del lenguaje o a través del controlador de un reloj interno o externo, o de un radioreceptor. La programación de los controladores de dispositivos se trata en el Capítulo 15; las siguientes subsecciones muestran la forma en que occam, Ada, Java y POSIX proporcionan abstracciones del reloj.

Tabla 12.1. Estándares del tiempo.

Nombre	Descripción	Nota
Día solar real	Tiempo entre dos cenits sucesivos (el punto más alto del Sol).	Varía a lo largo del año en 15 minutos (aprox.).
Hora temporal	Un duodécima parte del tiempo entre el amanecer y la puesta del sol.	Varía considerablemente a lo largo del año.
Tiempo universal (UT0)	Tiempo medio solar en el meridiano de Greenwich.	Definido en 1884.
Segundo (1)	1/86.400 de un día solar medio.	
Segundo (2)	1/31.566.925,9747 del año tropical de 1900.	Tiempo de efemérides definido en 1955.
UT1	Corrección al UT0 debida al desplazamiento de los polos.	
UT2	Corrección de UT1 debida a la variación de la velocidad de rotación de la Tierra.	
Segundo (3)	Duración de 9.192.631.770 periodos de la radiación correspondiente a la transición entre dos niveles hiperfinos de un estado del átomo de cesio 133.	La precisión de los relojes atómicos de Cesio actuales parece ser de $1/10^{13}$ (esto es, un error de reloj cada 300.000 años).
Tiempo atómico internacional (IAT)	Basado en el reloj atómico de cesio.	
Tiempo universal coordinado (UTC)	Un reloj IAT sincronizado a UT2 añadiendo ciertos pulsos ocasionales.	La diferencia máxima entre UT2 (basado en medidas astronómicas) y UTC (basado en medidas atómicas) se mantiene por debajo de los 0,5 segundos.

12.2.1 Temporizadores en occam2

Cualquier proceso occam2 puede obtener el valor del reloj «local» leyéndolo de un `TIMER` (no hay posibilidad de acceder al tiempo de calendario). Para ser coherente con el modelo de comunicación occam2 (que es uno a uno), cada proceso debe utilizar un `TIMER` distinto. La lectura de un `TIMER` sigue la misma sintaxis que la lectura de un canal, pero la semántica es diferente, ya que la lectura de un `TIMER` no puede provocar la suspensión; esto es, un reloj siempre estará dispuesto a ofrecer una salida.

```
TIMER reloj:
INT Tiempo:
SEQ
  reloj ? Tiempo          -- leer el tiempo
```

El valor proporcionado por un `TIMER` es de tipo `INT`, pero su significado depende de la implementación: proporciona un valor de reloj relativo, no absoluto. Una única lectura de un `TIMER`

carece, por lo tanto, de utilidad, mientras que la substracción entre dos lecturas consecutivas darán una medida del paso del tiempo entre ellas:

```
TIMER reloj:
INT anterior, actual, intervalo:
SEQ
  reloj ? anterior
  -- otros cálculos
  reloj ? actual
  intervalo := actual MINUS anterior
```

Se utiliza el operador `MINUS` en lugar de «-» para tener en cuenta el sentido modular de los incrementos. Esto se debe a que los enteros dados por un `TIMER` se incrementan en una unidad por cada unidad de tiempo; cuando eventualmente se alcance el valor entero máximo, el siguiente pulso (tick) entero se convertirá en el más negativo posible, y entonces continuará incrementándose. Los usuarios pueden olvidarse de este detalle siempre que utilicen los operadores aritméticos apropiados (definidos en el lenguaje), que son: `MINUS`, `PLUS`, `MULT` y `DIVIDE`. De hecho, cada `TIMER` realiza un `PLUS 1'` para cada pulso de reloj.

Según se puede ver, las posibilidades dadas por occam2 son bastante básicas (aunque posiblemente suficientemente adecuadas). Como sólo se reserva un entero para el reloj, existe una relación de compromiso clara entre la granularidad del reloj (esto es, el intervalo de tiempo que representa cada tic) y el rango de tiempos que se pueden acomodar. En la Tabla 12.2 se muestran los valores típicos, considerando un entero de 32 bits.

Table 12.2. Granularidades de un `TIMER` para un entero de 32 bits.

Granularidad	Rango (aproximado)
1 microsegundo	71.6 minutos
100 microsegundos	119 horas
1 milisegundo	50 días
1 segundo	136 años

12.2.2 Los paquetes reloj en Ada

El acceso a relojes en Ada lo proporciona un paquete de biblioteca (obligatorio) llamado `Calendar`, y otro paquete opcional de tiempo real. El paquete `Calendar` (véase el Programa 12.1) implementa una tipo abstracto de datos `Time`. Proporciona una función `Clock` para leer el tiempo, y varios subprogramas para conversiones entre `Time` y unidades de tiempo humanamente comprensibles, como `Year` (año), `Month` (mes), `Day` (día) y `Seconds` (segundos). Los tres primeros son subtipos enteros, mientras que `Seconds` se define como un subtipo del tipo primitivo `Duration`.

Programa 12.1. El paquete Ada Calendar.

```
package Ada.Calendar is

   type Time is private;

   subtype Year_Number is Integer range 1901..2099;
   subtype Month_Number is Integer range 1..12;
   subtype Day_Number is Integer range 1..31;
   subtype Day_Duration is Duration range 0.0..86400.0;

   function Clock return Time;

   function Year(Fecha:Time) return Year_Number;
   function Month(Fecha:Time) return Month_Number;
   function Day(Fecha:Time) return Day_Number;
   function Seconds(Fecha:Time) return Day_Duration;

   procedure Split(Fecha:in Time; Año:out Year_Number;
                   Mes:out Month_Number; Dia:out Day_Number;
                   Segundos:out Day_Duration);

   function Time_Of(Año:Year_Number; Mes:Month_Number;
                   Dia:Day_Number; Segundos:Day_Duration := 0.0)
                   return Time;

   function "+"(Left:Time;Right:Duration) return Time;
   function "+"(Left:Duration;Right:Time) return Time;
   function "-"(Left:Time;Right:Duration) return Time;
   function "-"(Left:Time;Right:Time) return Duration;

   function "<"(Left,Right:Time) return Boolean;
   function "<="(Left,Right:Time) return Boolean;
   function ">"(Left,Right:Time) return Boolean;
   function ">="(Left,Right:Time) return Boolean;

   Time_Error:exception;
   -- Time_Error es desencadenada por Time_Of, Split,"+" y "-"

private
   -- dependiente de implementación
end Ada.Calendar;
```

El tipo `Duration` es un real de punto fijo que representa un intervalo de tiempo (tiempo relativo). Tanto su precisión como su rango dependen de la implementación, aunque su rango deberá ser por lo menos [–86.400,0 .. 86.400,0] (suficiente para almacenar los segundos que tiene un día). Su granularidad no debe ser mayor de 20 milisegundos. En esencia, un valor de tipo `Duration` debe interpretarse como una cantidad de segundos. Hay que destacar que, además de los subprogramas anteriores, el paquete `Calendar` define operadores aritméticos para combinaciones de parámetros `Duration` y `Time`, y operadores de comparación para valores `Time`.

El código preciso para medir el tiempo empleado en ejecutar un cómputo es bastante sencillo. Hay que destacar el uso del operador "-", que toma dos valores `Time` y devuelve un valor del tipo `Duration`.

```
declare
   Tiempo_Anterior, Tiempo_Actual : Time;
   Intervalo : Duration;
begin
   Tiempo_Anterior := Clock;
   -- otros cálculos
   Tiempo_Actual:= Clock;
   Intervalo := Tiempo_Actual - Tiempo_Anterior;
end;
```

Otro reloj del lenguaje es el dado por el paquete opcional `Real_Time`. Éste tiene una forma similar a `Calendar`, pero está orientado a proporcionar una mayor granularidad. La constante `Time_Unit` es la cantidad más pequeña de tiempo representada por el tipo `Time`. El valor de `Tick` (pulso) no debe ser mayor de un milisegundo, mientras que el rango de `Time` (desde la época que representa el inicio del programa) debe ser de, al menos, cincuenta años.

Además de proporcionar una granularidad más fina, `Clock` (de `Real_Time`) es monotónico. El reloj `Calendar` está diseñado para proporcionar la abstracción de un «reloj de pared», y está, por lo tanto, sujeto a saltos de año, saltos de segundo y otros ajustes. En el Programa 12.2 se describe, a grandes rasgos, el paquete `Real_Time`.

Programa 12.2. *El paquete Ada de reloj en tiempo real.*

```
package Ada.Real_Time is
   type Time is private;
   Time_First: constant Time;
   Time_Last: constant Time;
   Time_Unit: constant := -- numero-real-definido-por-la-implementacion;

   type Time_Span is private;
   Time_Span_First: constant Time_Span;
   Time_Span_Last: constant Time_Span;
   Time_Span_Zero: constant Time_Span;
```

(Continuación)

```
   Time_Span_Unit: constant Time_Span;

   Tick: constant Time_Span;
   function Clock return Time;

   function "+" (Left: Time; Right: Time_Span) return Time;
   ...

   function "<" (Left, Right: Time) return Boolean;
   ...

   function "+" (Left, Right: Time_Span) return Time_Span;
   ...

   function "<" (Left, Right: Time_Span) return Boolean;
   ...

   function "abs"(Right : Time_Span) return Time_Span;

   function To_Duration (Ts : Time_Span) return Duration;
   function To_Time_Span (D : Duration) return Time_Span;

   function Nanoseconds (Ns: Integer) return Time_Span;
   function Microseconds (Us: Integer) return Time_Span;
   function Milliseconds (Ms: Integer) return Time_Span;

   type Seconds_Count is range -- definido-en-la-implementacion

   procedure Split(T : in Time; Sc: out Seconds_Count;
                   Ts : out Time_Span);
   function Time_Of(Sc: Seconds_Count; Ts: Time_Span) return Time;

private
   -- no especificado por el lenguaje
end Ada.Real_Time;
```

12.2.3 Relojes en Java para tiempo real

El estándar Java soporta la noción de reloj de pared, y por lo tanto tiene una función similar a la de Ada. El tiempo actual se puede obtener en Java invocando al método estático `System.currentTimeMillis` del paquete `java.lang`. El sistema devuelve el número de

milisegundos desde la medianoche del 1 de enero de 1970 GMT. La clase `Date`, en `java.util`, utiliza este método por defecto cuando construye objetos fecha.

Java para tiempo real añade a estas posibilidades relojes de tiempo real con tipos de tiempo de alta resolución. El Programa 12.3 muestra un resumen de la definición base de la clase de tiempo de alta resolución. Hay métodos para leer, escribir y comparar valores de tiempo. Esta clase abstracta tiene tres subclases: una para representar tiempo absoluto, otra para representar tiempo relativo (similar al tipo `Duration` de Ada), y otra para representar tiempo racional. Todas ellas se muestran en el Programa 12.4. El tiempo absoluto, en realidad, se considera como un tiempo relativo al 1 de enero de 1970, GMT. El tiempo racional es un tipo de tiempo relativo al cual se le asocia cierta frecuencia. Se utiliza para representar la tasa en la que se dan ciertos eventos (por ejemplo, la ejecución periódica de hilos).

Programa 12.3. Un extracto de la clase Java para tiempo real `HighResolutionTime`.

```
public abstract class HighResolutionTime implements
        java.lang.Comparable
{
   public abstract AbsoluteTime absolute(Clock reloj,
                                         AbsoluteTime destino);

   ...
   public boolean equals(HighResolutionTime tiempo);

   public final long getMilliseconds();
   public final int getNanoseconds();

   public void set(HighResolutionTime tiempo);
   public void set(long milis);
   public void set(long milis, int nanos);
}
```

Las clases `HighResolutionTime`, `AbsoluteTime`, `RelativeTime` y `RationalTime` son todas ellas suficientemente claras. Sin embargo, el método `absolute` necesita cierta aclaración. Su papel es convertir el tiempo encapsulado (ya sea absoluto, relativo o racional) en un tiempo absoluto relativo a algún reloj. Si se pasa como parámetro un objeto `AbsoluteTime`, el objeto se actualiza para reflejar el valor de tiempo encapsulado. Más aún, la función devuelve el mismo objeto. Esto se debe a que si se pasa un objeto nulo como parámetro, se crea y se devuelve un objeto nuevo. En Java, el parámetro se copia por valor, y por lo tanto no puede ser actualizado. Consecuentemente, resulta necesario que la función cree y devuelva un objeto `AbsoluteTime` nuevo.

La clase `Clock` de Java para tiempo real mostrada en el Programa 12.5, define la clase abstracta de la cual se derivan todos los relojes. El lenguaje permite muchos tipos de reloj diferen-

Programa 12.4. Las clases de tiempo absoluto, relativo y racional de Java para tiempo real.

```
public class AbsoluteTime extends HighResolutionTime
{

   // varios métodos constructores incluyendo ...
   public AbsoluteTime(AbsoluteTime T);
   public AbsoluteTime(long milis, int nanos);

   public AbsoluteTime absolute(Clock reloj, AbsoluteTime destino);

   public AbsoluteTime add(long milis, int nanos);
   public final AbsoluteTime add(RelativeTime tiempo);
   ...
   public final RelativeTime subtract(AbsoluteTime tiempo);
   public final AbsoluteTime subtract(RelativeTime tiempo);

}

public class RelativeTime extends HighResolutionTime
{
   // varios métodos constructores incluyendo ...
   public RelativeTime(long milis, int nanos);
   public RelativeTime(RelativeTime tiempo);

   public AbsoluteTime absolute(Clock reloj, AbsoluteTime destino);

   public RelativeTime add(long milis, int nanos);
   public final RelativeTime add(RelativeTime tiempo);

   public void addInterarrivalTo(AbsoluteTime destino);
   public final RelativeTime subtract(RelativeTime tiempo);

   ...
}

public class RationalTime extends RelativeTime
{
   // varios métodos constructores incluyendo ...
public RationalTime(int frecuencia, RelativeTime intervalo)
       throws IllegalArgumentException;

public AbsoluteTime absolute(Clock reloj, AbsoluteTime destino);
```

(Continuación)

```
public void addInterarrivalTo(AbsoluteTime destino);
public int getFrequency();
public RelativeTime getInterarrivalTime(RelativeTime destino);
public void set(long milis, int nanos)
       throws IllegalArgumentException;
public void setFrequency(int frecuencia)
       throws ArithmeticException;
}
```

tes; por ejemplo, podría haber un reloj de tiempo de ejecución (que midiera el tiempo de ejecución empleado). Siempre hay un reloj de tiempo real que avanza sincronizado con el mundo exterior. El método `getRealtimeClock` permite obtener el valor de este reloj.[1] También tiene otros métodos para obtener la resolución de un reloj y, si el hardware lo permite, fijar la resolución de un reloj.

Programa 12.5. La clase Java para tiempo real `Clock`.

```
public abstract class Clock
{
   public Clock();

   public static Clock getRealtimeClock();

   public abstract RelativeTime getResolution();

   public AbsoluteTime getTime();
   public abstract void getTime(AbsoluteTime tiempo);

   public abstract void setResolution(RelativeTime resolucion);
}
```

El código para medir el tiempo empleado en ejecutar un cómputo es:

```
{
  AbsoluteTime tiempoAnterior, tiempoActual;
  RelativeTime intervalo;
  Clock reloj = Clock.getRealtimeClock();

  tiempoAnterior = reloj.getTime();
```

[1] Hay que destacar que al ser un método estático, puede invocarse directamente sin tener que emplear ninguna subclase.

```
    // otros cálculos
    tiempoActual = reloj.getTime();

    intervalo = tiempoActual.subtract(tiempoAnterior);

}
```

12.2.4 Relojes en C y POSIX

ANSI C tiene una biblioteca estándar de interfaz con el tiempo de «calendario». Ésta define un tipo de tiempo básico `time_t` y varias rutinas para la manipulación de objetos de este tipo (el Programa 12.6 define algunas de estas funciones). POSIX permite que una implementación soporte varios relojes. Cada reloj tiene su propio identificador (de tipo `clockid_t`), y el estándar IEEE requiere que se soporte al menos un reloj (`CLOCK_REALTIME`). El Programa 12.7 ilustra una interfaz típica en C para relojes POSIX.

El valor devuelto por un reloj (vía `clock_gettime`) está definido por la estructura `timespec`, donde `tv_sec` representa el numero de segundos transcurridos desde el 1 de enero de 1970, y `tv_nsec` la cantidad adicional de nanosegundos (aunque se muestra como un entero lar-

Programa 12.6. Una interfaz ANSI C para fecha y hora.

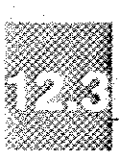
```
typedef ... time_t;

struct tm {
   int tm_sec;        /* segundos pasados del minuto - [0, 61] */
                      /* 61 permitido para saltos de 2 segundos */
   int tm_min;        /* minutos pasados de la hora - [0, 59] */
   int tm_hour;       /* horas desde medianoche - [0, 23] */
   int tm_mday;       /* día del mes - [1, 31] */
   int tm_mon;        /* meses desde Enero - [0, 11] */
   int tm_year;       /* años desde 1900 */
   int tm_wday;       /* días desde el Domingo - [0, 6] */
   int tm_yday;       /* días desde el primero de Enero - [0, 365] */
   int tm_isdst;      /* indicador de ajuste de horario de verano */
};
double difftime(time_t tiempo1, time_t tiempo2);
   /* resta dos valores de tiempo */
time_t mktime(struct tm *ptiempo); /* compone un valor de tiempo */
time_t time(time_t *reloj);
   /* devuelve el tiempo actual y si el reloj no es nulo */
   /* también deposita el tiempo en esa zona de memoria */
```

Programa 12.7. *Una interfaz C para los relojes POSIX.*

```
#define CLOCK_REALTIME ...;
#define CLOCK_PROCESS_CPUTIME_ID ...;
#define CLOCK_THREAD_CPUTIME_ID ...;

struct timespec {
   time_t tv_sec;  /* numero de segundos */
   long tv_nsec;   /* numero de nanosegundos */
};
typedef ... clockid_t;

int clock_gettime(clockid_t reloj_id, struct timespec *tmpo);
int clock_settime(clockid_t reloj_id, const struct timespec *tmpo);
int clock_getres(clockid_t reloj_id, struct timespec *res);

int clock_getcpuclockid(pid_t pid, clockid_t *reloj_id);
int clock_getcpuclockid(pthread_t_t thread_id, clockid_t *reloj_id);

int nanosleep(const struct timespec *tmpodormir, struct timespec *tmporest);
/* Aviso, nanosleep devuelve -1 si es interrumpido por una señal. */
/* En este caso, tmporest contendrá el tiempo de sueño restante */
```

go, el valor que tome `tv_nsec` debe ser siempre menor que 1.000.000.000 y no negativo: C no proporciona un mecanismo de subtipado). POSIX exige que la resolución mínima de `CLOCK_REALTIME` sea de 50Hz (20 milisegundos); la función `clock_getres` permite establecer la resolución del reloj.

Los relojes de tiempo de ejecución se discutirán en la Sección 12.8.1.



12.3 Retraso de un proceso

Además de poder acceder a un reloj, los procesos también deben ser capaces de retrasar su ejecución, ya sea por un periodo fijo de tiempo o hasta cierto instante absoluto de tiempo futuro.

12.3.1 Retardos relativos

Un retardo relativo posibilita que un proceso sea encolado hasta cierto evento futuro en lugar de efectuar una espera ocupada basada en llamadas al reloj. El siguiente código ADA muestra la forma en que las tareas pueden esperar durante 10 segundos mediante un bucle:

```
Comienzo := Clock; -- de Calendar
loop
  exit when (Clock - Comienzo) > 10.0;
end loop;
```

Para eliminar la necesidad de estas esperas ocupadas, la mayoría de los lenguajes y sistemas operativos proporcionan alguna forma de primitiva de retardo. En Ada, se trata de una sentencia de retardo.

```
delay 10.0;
```

El valor tras **delay** (de tipo `Duration`) es relativo (esto es, la sentencia anterior significa «espera 10 segundos a partir de ahora»); un valor negativo se considerará como cero.

En POSIX, se pueden obtener retardos invocando la llamada del sistema «sleep», si fuera suficiente con una granularidad gruesa (esto es, un retardo de «segundos»), o «nanosleep» si se necesitara una granularidad más fina (véase el Programa 12.7). En este caso, basándose en `CLOCK_REALTIME`, Java proporciona funcionalidades similares a las de POSIX. El método `sleep` de la clase `Thread` permite que un hilo se retrase a sí mismo con una granularidad de milisegundos. La clase `RealtimeThread` permite precisión de alta resolución con respecto a un reloj.

Es importante recalcar que, un «retardo» o un «sleep» sólo garantizan que el proceso será ejecutable después de que el periodo indicado haya finalizado.[2] El retardo real antes de que la tarea comience su ejecución depende, por supuesto, de los demás procesos con los cuales está compitiendo por el procesador. También debería tenerse en cuenta que la granularidad del retardo y la del reloj no tienen por qué ser necesariamente la misma. Por ejemplo, POSIX y Java para tiempo real permiten granularidades hasta de nanosegundos, aunque pocos sistemas actuales soporten esto.

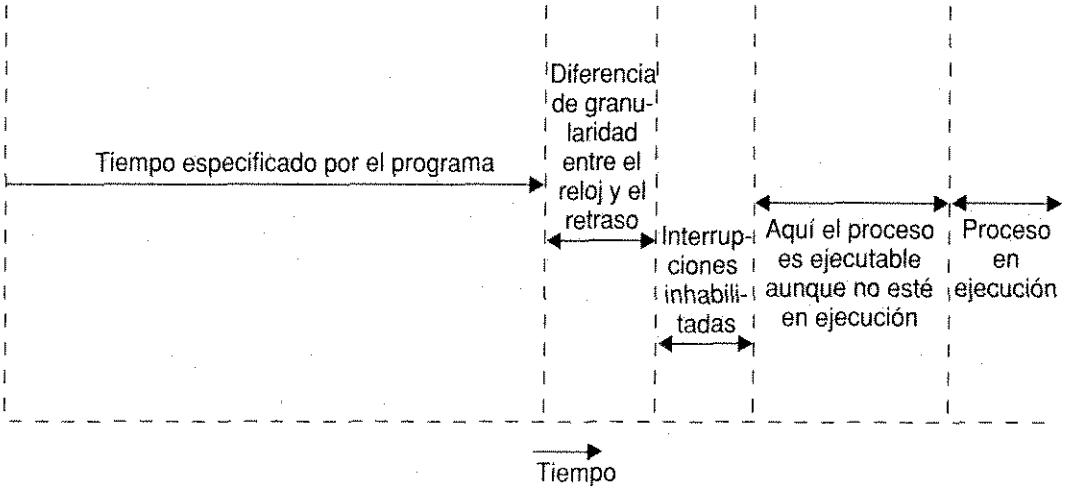


Figura 12.1. Tiempos del retardo.

[2] En POSIX, el proceso puede, de hecho, ser despertado antes si recibe una señal. En este caso se devolverá una indicación de error, y el parámetro `tmporest` devolverá el tiempo restante. La situación en Java es similar. Si un hilo es interrumpido, entonces será despertado, y se generará el objeto error `InterruptedException`. En Ada, una tarea retrasada sólo será despertada temprano si está dentro de una sentencia `select-then-abort`.

Más aún, el reloj interno se puede implementar utilizando una interrupción que puede ser inhibida durante cortos periodos de tiempo. La Figura 12.1 ilustra los factores que afectan a los retardos.

12.3.2 Retardos absolutos

La utilización de **delay** en Ada (y de `sleep` en POSIX o en Java para tiempo real) está asociada a un periodo de tiempo relativo (por ejemplo, 10 segundos desde ahora). Si fuera necesario retrasar hasta un instante de tiempo absoluto, el programador necesitaría una primitiva adicional, o tendría que calcular el periodo a retrasar. Si una acción debe tener lugar, por ejemplo, 10 segundos después del comienzo de alguna otra acción, entonces se podrá utilizar el código Ada siguiente (con `Calendar`):

```
Comienzo := Clock;
Primera_Accion;
delay 10.0 - (Clock - Comienzo);
Segunda_Accion;
```

Lamentablemente, esto bien pudiera no lograr el resultado deseado. Para que esta formulación tenga el resultado buscado, entonces

```
delay 10.0 - (Clock - Comienzo);
```

debería ser una acción no interrumpible (atómica), cosa que no es. Por ejemplo, si `Primera_Accion` dura dos segundos, entonces

```
10.0 - (Clock - Comienzo);
```

debería ser igual a ocho segundos. Pero, si después de haber calculado ese valor se desaloja la tarea del procesador, podrían pasar varios segundos (pongamos 3) antes de que se ejecutara otra vez. En ese instante, se habrá producido un retardo de ocho segundos, en lugar de los cinco previstos. Para resolver este problema, Ada introduce la sentencia **delay until**:

```
Comienzo := Clock;
Primera_Accion;
delay until Comienzo + 10.0;
Segunda_Accion;
```

Como ocurre con `delay`, **`delay until`** especifica únicamente un límite inferior. Las tareas implicadas no serán liberadas antes de que se haya alcanzado el tiempo especificado en la sentencia, pero podrían ser liberadas después.

El tiempo sobrepasado, asociado tanto con los retardos relativos como con los absolutos, se denomina **deriva local**, y no puede ser eliminado. Es posible, sin embargo, eliminar la **deriva acumulada**, que puede producirse si se permite que las derivas locales se superpongan. El código siguiente, escrito en Ada, muestra cómo ejecutar el cálculo de `Accion` cada 7 segundos de media. Este código compensará cualquier deriva local. Por ejemplo, si entre dos llamadas conse-

cutivas a Accion mediaran 7,4 segundos, entonces el siguiente retardo sería sólo de 6,6 segundos (aproximadamente).

```
declare
  Siguiente : Time;
  Intervalo : constant Duration := 7.0;
begin
  Siguiente := Clock + Intervalo;
  loop
    Accion;
    delay until Siguiente;
    Siguiente := Siguiente + Intervalo;
  end loop;
end;
```

En Java para tiempo real se puede usar el método sleep (definido en la clase RealtimeThread) tanto para retardos relativos como absolutos.

Occam2 sólo soporta retardos absolutos. Para reforzar su semántica de final abierto, se utiliza la palabra clave AFTER. Si un proceso deseara esperar 10 segundos, primero debería leer el reloj TIMER, después añadir 10 segundos, y luego esperar ese tiempo. El valor de 10 segundos se obtiene a través de la constante G, que se introduce (aquí) para dar una medida de la granularidad de la implementación (G es el número de actualizaciones por segundo de TIMER):

```
SEQ
  reloj ? ahora
  reloj ? AFTER ahora PLUS (10 * G)
```

En occam2, el código para evitar las derivas acumuladas es:

```
INT siguiente, ahora:
VAL intervalo IS 7*G:
SEQ
  reloj ? ahora
  siguiente := ahora PLUS intervalo
  WHILE TRUE
    SEQ
      ACTION
      reloj ? AFTER siguiente
      siguiente := siguiente PLUS intervalo
```

También se pueden construir retardos absolutos en POSIX utilizando un temporizador absoluto y esperando a la señal que se genera cuando expira (véase la Sección 12.8.1).

12.4 Programación de tiempos límite de espera

Quizás la restricción temporal más simple que puede tener un sistema embebido es reconocer y actuar sobre la no ocurrencia de cierto suceso externo. Por ejemplo, se precisa que un sensor de temperatura realice una lectura cada segundo, y que se considere como fallo si no se devuelve medición alguna en al menos 10 segundos. En general, un **tiempo límite de espera** (timeout) es una restricción sobre el tiempo que un proceso está dispuesto a esperar por una comunicación. En los Capítulos 8 y 9 se han discutido en detalle las posibilidades de comunicación entre procesos. Cada una de ellas requerirá un tiempo límite de espera si va a ser utilizada en un entorno de tiempo real.

Como ocurre con la espera en las comunicaciones, también se necesitan tiempos límite de espera en la ejecución de acciones. Por ejemplo, un programador puede necesitar que cierta sección de código se ejecute en un cierto tiempo. Si esto no sucediera, en tiempo de ejecución, entonces se precisará de algún mecanismo de recuperación de errores.

12.4.1 Comunicación por variables compartidas y tiempos límite de espera

En el Capítulo 8, se discutieron varios mecanismos de comunicación y sincronización basados en variables compartidas. Tanto el acceso mutuamente excluyente a una sección crítica como la sincronización de condiciones fueron vistos como requisitos importantes. Cuando un proceso intenta acceder a una sección crítica, será bloqueado si cualquier otro proceso se encuentra ya activo en la sección. Este bloqueo, sin embargo, está limitado por el tiempo necesario para ejecutar el código dentro de la sección y el número de procesos que también desean ejecutar dicha sección crítica. Por esta razón, normalmente no se considera necesario disponer de un tiempo límite de espera asociado al intento de entrada. (Sin embargo, POSIX proporciona estos tiempos límite de espera.) En la Sección 13.10, se analiza en detalle este tiempo de bloqueo.

Por contra, el bloqueo asociado con la sincronización de condición no es fácilmente limitable, y depende de la aplicación. Por ejemplo, un proceso productor que intenta colocar datos en un búfer lleno debe esperar a que un proceso consumidor retire datos del mismo. Los procesos consumidores podrían no estar preparados para ello durante un periodo importante de tiempo. Por esta razón, es importante permitir que los procesos tengan la opción de limitar el tiempo de espera por una sincronización de condición.

Las posibilidades de sincronización consideradas en el Capítulo 8 eran:

- Semáforos.
- Regiones críticas condicionales.
- Variables de condición en monitores, métodos mutuamente excluyentes o sincronizados.
- Entradas en objetos protegidos.

El sistema Euclid de tiempo real (Kligerman y Stoyenko, 1986) utiliza semáforos, y extiende la semántica de `wait` para incluir un límite temporal. La siguiente sentencia ilustra cómo un proceso podría suspenderse a sí mismo en el semáforo `INVOCACION` especificando un tiempo límite de espera de 10 segundos:

```
wait INVOCACION noLongerThan 10 : 200
```

Si el proceso no recibe una señal en esos 10 segundos, se generará la señal 200 (las excepciones en el sistema de Euclid de tiempo real están numeradas). Además, hay que tener en cuenta, que no existe obligación de elegir el proceso para ejecución y ejecutarlo dentro de los 10 segundos. Lo mismo rige en cualquier mecanismo de tiempo límitado de espera; en ellos sólo se indica cuándo el proceso debería ser ejecutable de nuevo.

POSIX soporta un tiempo límite de espera explícito en las esperas en semáforos y en variables de condición (véanse los Programas 8.2 y 8.3). La siguiente sentencia ilustra la forma en que un proceso podría suspenderse a sí mismo en el semáforo `INVOCACION` con un tiempo límite de espera de valor `tiempolimite`:

```
if(sem_timedwait(&invocacion, &tiempolimite) < 0) {
  if (errno == ETIMEDOUT) {
    /* tiempo límite cumplido */
  }
  else {
  /* algún otro error */
}
} else {
  /* semaforo bloqueado */
};
```

Si no pudiera bloquearse el semáforo en el tiempo `tiempolimite`, se asignará el valor `ETIMEDOUT` a la variable `errno`.

En los objetos protegidos de Ada, se utiliza la sincronización de evitación, y, por lo tanto, es en el punto de entrada protegida donde se debe aplicar el tiempo límite de espera. Como Ada trata la invocación a una entrada protegida del mismo modo que una invocación de entrada a una tarea, se utiliza un mecanismo similar de limitación del tiempo de espera. A continuación, se describe todo ello en relación con el paso de mensajes.

En Java, puede utilizarse el método `wait` con un tiempo límite de espera, ya sea con una granularidad de milisegundos o de nanosegundos.

12.4.2 Paso de mensajes y tiempo límite de espera

El Capítulo 9 trató sobre las diversas formas de paso de mensajes. Todas ellas requieren sincronización. Incluso con los sistemas asíncronos, un proceso puede desear esperar un mensaje (o el

búfer de mensajes del proceso emisor podría llenarse). Con el paso de mensajes síncrono, una vez que un proceso se ha involucrado en una comunicación debe esperar hasta que tal evento haya ocurrido. Por estas razones, se necesita limitar el tiempo de espera. Para ilustrar la programación de tiempos límite de espera considere primero una tarea `controlador` (en Ada), a la que se invoca desde alguna otra tarea con una nueva lectura de temperatura:

```
task Controlador is
  entry Invoca(T : Temperatura);
end Controlador;

task body Controlador is
  -- declaraciones
begin
  loop
    accept Invoca(T : Temperatura) do
      Nueva_Temp := T;
    end Invoca;
    -- otras acciones
  end loop;
end Controlador;
```

Ahora hay que modificar el controlador de forma que reaccione ante la ausencia de una invocación. Este requisito se puede solucionar utilizando los constructores vistos anteriormente. Se utiliza una segunda tarea que se retrasa a sí misma por un periodo de tiempo, y entonces invoca al controlador. Si el controlador la acepta antes de un `Invoca` normal, entonces es que se ha cumplido el tiempo límite de espera:

```
task Controlador is
  entry Invoca(T : Temperatura);
private
  entry TiempoLimiteEspera;
end Controlador;

task body Controlador is
  task Reloj is
    entry Avanza(D : Duracion);
  end Reloj;
  -- otras declaraciones
  task body Reloj is
    Valor_TimempoLimiteEspera : Duracion;
  begin
    accept Avanza(D : Duracion) do
      Valor_TiempoLimiteEspera := D;
```

```
      end Avanza;
      delay Value_TiempoLimiteEspera;
      Controlador.TiempoLimiteEspera;
   end Reloj;
begin
   loop
      Reloj.Avanza(10.0);
      select
         accept Invoca(T : Temperatura) do
            Nueva_Temp := T;
         end Invoca;
      or
         accept TiempoLimiteEspera;
         -- acción para TiempoLimiteEspera
      end select;
      -- otras acciones
   end loop;
end Controlador;
```

Aunque este código sólo trate con el primer periodo de tiempo límite de espera, puede modificarse (aunque no resulte trivial) para conseguir una ejecución continua. De todas maneras, como la necesidad de un tiempo límite de espera es un problema muy común, resultaría deseable contar con una forma de expresarlo más concisa. Normalmente esto se consigue en los lenguajes de tiempo real mediante una forma especial de alternativa en una espera selectiva. El ejemplo anterior se podría codificar de forma más apropiada como sigue:

```
task Controlador is
   entry Invoca(T : Temperatura);
end Controlador;

task body Controlador is
   -- declaraciones
begin
   loop
      select
         accept Invoca(T : Temperatura) do
            Nueva_Temp := T;
         end Invoca;
      or
         delay 10.0;
         -- acción del timempo límite de espera
      end select;
      -- otras acciones
```

```
  end loop;
end Controlador;
```

La alternativa de retardo se activa cuando el tiempo de espera se ha cumplido. Si esta alternativa es elegida (esto es, si no se registra `Invoca` en 10 segundos), entonces se ejecutan las sentencias que se encuentran trás el `delay`.

El ejemplo anterior utiliza un retardo relativo, pero también se pueden encontrar retardos absolutos. Considérese el código siguiente, que posibilita que una tarea acepte reservas hasta el instante `Tiempo_de_Cierre`:

```
task Agente_Tickets is
  entry Reserva(...);
end Agente_Tickets;

task body Agente_Tickets is
  -- declaraciones
  Tienda_Abierta : Boolean := True;
begin
  while Tienda_Abierta loop
    select
      accept Reserva(...) do
        -- detalles de cada reserva
      end Reserva;
    or
      delay until Tiempo_de_Cierre;
      Tienda_Abierta := False;
    end select;
    -- procesamiento de las reservas
  end loop;
end Agente_Tickets;
```

Dentro del modelo Ada, no tendría sentido mezclar una parte else, una alternativa terminate y una alternativa delay. Estas tres estructuras son, por lo tanto, mutuamente excluyentes; una sentencia de selección puede tener, como mucho, sólo una de ellas. Sin embargo, si existe una alternativa delay, la selección puede tener varios retardos, pero deben ser todos del mismo tipo (esto es, todos delay o delay until). En cada ocasión será operativo aquél con una duración más corta o con el tiempo absoluto más temprano.

Los mecanismos de tiempo límite de espera son comunes en los lenguajes de programación concurrente basada en mensajes. Occam2, como Ada, utiliza la primitiva «delay» como parte del constructor de una espera selectiva para indicar el tiempo límite de espera:

```
WHILE TRUE
  SEQ
```

```
ALT
  invocacion ? nueva_temp
     -- otras acciones
  reloj ? AFTER (10 * G)
     -- acción en caso de tiempo límite de espera
```

donde `reloj` es un temporizador (`TIMER`).

Tanto el ejemplo de occam2 como el de Ada muestran cómo se programa un tiempo límite de espera para la recepción de un mensaje. Ada realmente va más allá, y permite fijar un tiempo límite de espera en un mensaje enviado. Para ilustrar esto, considere el controlador de un dispositivo que está proporcionando temperaturas al anterior controlador en Ada:

```
loop
  -- conseguir nueva temperatura T
  Controlador.Invoca(T);
end loop;
```

Como las lecturas de las nuevas temperaturas están disponibles continuamente (y no existe interés en mandarle al controlador una lectura caducada), el controlador del dispositivo puede desear ser suspendido esperando al controlador durante sólo medio segundo antes de desechar la llamada. Esto se obtiene utilizando una forma especial de la sentencia `select` que tiene un solo punto de entrada y una sola alternativa de retardo.

```
loop
  -- obtener nueva temperatura T
  select
     Controlador.Invoca(T);
  or
     delay 0.5;
     null;
  end select;
end loop;
```

El **`null`** no es estrictamente necesario, pero muestra que el `delay` puede tener a continuación sentencias arbitrarias, que serán ejecutadas si el retardo expira antes de que la invocación de entrada sea aceptada. Ésta es una forma especial de selección. No puede tener más de una invocación de entrada, y no puede mezclar invocaciones de entrada y sentencias de aceptación. La acción invocada se denomina **invocación de entrada temporizada**. Debe destacarse que el periodo de tiempo especificado en la invocación es un valor del tiempo límite de espera para que la invocación sea aceptada; *no es un tiempo límite de espera para la terminación de la sentencia de aceptación asociada.*

Si una tarea sólo deseara efectuar una invocación de entrada si la tarea invocada está inmediatamente preparada para aceptar la llamada, entonces, en lugar de realizar una invocación de

entrada temporizada con un tiempo igual a cero, se puede hacer una **invocación de entrada condicional**:

```
select
  T.E -- entrada E en la tarea T
else
  -- otras acciones
end select;
```

Las «otras acciones» se ejecutan sólo si T no está preparada para aceptar E de forma inmediata. «Inmediatamente» significa que T ya está suspendida en **accept** E o en una sentencia de selección con una alternativa abierta (y que ha sido elegida).

En los ejemplos anteriores se han utilizado tiempos límite de espera en la comunicación entre procesos; en Ada también se pueden realizar invocaciones de entrada temporizadas (y condicionales) sobre objetos protegidos:

```
select
  P.E ; -- E es una entrada en el objeto protegido P
or
  delay 0.5;
end select;
```

12.4.3 Tiempos límite de espera en acciones

En la Sección 10.5, se discutieron los mecanismos que posibilitan que los procesos alteren sus flujos de control en virtud de notificaciones asíncronas. Un tiempo límite de espera puede verse como un tipo de notificación, por lo que si se soportan las notificaciones asíncronas, también se podrán utilizar los tiempos límite de espera. Tanto el modelo de reanudación (eventos asíncronos) como el de terminación (transferencia de control asíncrona) se pueden enriquecer utilizando tiempos límite de espera. Aunque para los tiempos límite de espera en las acciones el modelo necesario es el de terminación.

Por ejemplo, en la Sección 10.8 se presentó la transferencia de control asíncrona de Ada (ATC). Con ella, una acción puede ser abortada si un «evento disparador» sucede antes de que la acción haya sido completada. Uno de los eventos disparadores permitidos es el paso del tiempo. Concretamente, considere una tarea que contiene una acción que debe ser completada dentro de 100 milisegundos. El código siguiente soporta directamente este requisito:

```
select
  delay 0.1;
then abort
  -- acción
end select;
```

Programa 12.8. La clase Timed de Java para tiempo real.

```
public class Timed extends AsynchronouslyInterruptedException
                   implements java.io.Serializable
{
   public Timed(HighResolutionTime tiempo) throws IllegalArgumentException;

   public boolean doInterruptible(Interruptible logica);

   public void resetTime(HighResolutionTime tiempo);
}
```

Si la acción se demora demasiado, entrará en acción el evento disparador y se abortará la acción. Ésta resulta una efectiva manera de capturar «código descontrolado».

Los tiempos límite de espera suelen asociarse con condiciones de error; si una comunicación no ha ocurrido en *X* milisegundos, entonces ha sucedido algo malo, y habrá que emprender alguna acción correctora. Sin embargo, no es éste su único uso. Considérese una tarea con un componente obligatorio y una parte opcional. Los cálculos obligados producen (rápidamente) un resultado adecuado que se asigna a un objeto protegido. La tarea debe completarse en un tiempo fijo; pero si quedara tiempo disponible tras ejecutar el componente obligado, se puede utilizar un algoritmo opcional para mejorar el valor de salida. Para programar esto se necesita limitar de nuevo el tiempo de la acción.

```
declare
  Resultado_Preciso : Boolean;
begin
  Tiempo_Disponible:= ...
  -- parte obligatoria
  Resultados.Escribe(...); -- llamada al procedimiento
                           -- sobre un objeto protegido externo
  select
     delay until Tiempo_Disponible;
     Resultado_Preciso := False;
  then abort
     while Puede_Ser_Mejorado loop
        -- mejorar el resultado
        Resultados.Escribe(...);
     end loop;
     Resultado_Preciso := True;
  end select;
end;
```

Observe que si se consume el tiempo límite durante la escritura en el objeto protegido, la escritura se completará adecuadamente, ya que la invocación a un objeto protegido es una acción respetada por los abortos (esto es, el efecto del aborto se pospone hasta que la tarea abandona el objeto protegido).

En Java para tiempo real, los tiempos límite de espera sobre las acciones provienen de una subclase de `AsynchronouslyInterruptedException` denominada `Timed`. Esta clase está definida en el Programa 12.8 (véase la Sección 14.4.3 para más información relativa a `java.io.Serializable`).

El ejemplo anterior se puede escribir, en Java para tiempo real, como sigue:

```
public class ResultadoPreciso
{
  public tipoResultado valor; // el resultado
  public boolean resultadoPreciso; // indica si es impreciso
}


public class CalculoImpreciso
{
  private HighResolutionTime TiempoDisponible;
  private ResultadoPreciso resultado = new ResultadoPreciso();

  public CalculoImpreciso(HighResolutionTime T)
  {
    TiempoDisponible = T; //puede ser absoluto o relativo
  }

  private tipoResultado parteObligatoria()
  {
    // función que implementa la parte obligatoria
  };

  public ResultadoPreciso Servicio() // servicio público
  {
    Interruptible I = new Interruptible()
    {
      public void run(AsynchronouslyInterruptedException excepcion)
                    throws AsynchronouslyInterruptedException
      {
        // ésta es la función opcional que mejora
        // la parte obligatoria
```

```
            boolean puedeSerMejorado = true;

            while(puedeSerMejorado)
            {
               // mejorar el resultado
               synchronized(this){
                  // escribir el resultado --
                  // la sentencia synchronized asegura la
                  // atomicidad de la operación de escritura
               }
            }
            resultado.resultadoPreciso = true;
         }

         public void interruptAction(
               AsynchronouslyInterruptedException excepcion)
         {
            resultado.resultadoPreciso = false;
         }
      };

      Timed t = new Timed(TiempoDisponible);

      resultado.valor = parteObligatoria(); // calcular la parte obligatoria
      if(t.doInterruptible(I))
         // ejecutar la parte opcional con el reloj
         return resultado;
      else ... ;
   }
}
```

Los tiempos límite de espera son una característica importante de los sistemas de tiempo real; sin embargo, están lejos de ser la única restricción temporal significativa. El resto de este capítulo y el siguiente tratarán el tema más general de los tiempos límites y la forma de asegurar su cumplimiento.

12.5 Especificación de requisitos de temporización

Para muchos sistemas de tiempo real, no resulta suficiente que el software sea lógicamente correcto; los programas también deben satisfacer restricciones temporales provenientes del sistema

físico subyacente. Estas restricciones pueden ir más allá de los simples tiempos límite de espera. Desafortunadamente, las técnicas de ingeniería existentes para grandes sistemas de tiempo real son, en general, todavía demasiado *ad hoc*. A menudo, el sistema lógicamente correcto se especifica, diseña y construye (quizás como prototipo), y finalmente se prueba para ver si cumple los requisitos temporales. Si no lo hace, se aplicarán varios ajustes finos y reescrituras. El resultado es un sistema que puede ser difícil de entender y caro de mantener y de actualizar. Se necesita un tratamiento del tiempo más sistemático.

Los trabajos existentes siguen dos caminos muy distintos. Una dirección de desarrollo se basa en el uso de lenguajes de semántica formalmente definida y requisitos temporales, junto con notaciones y lógicas que posibiliten que se puedan representar y analizar las propiedades temporales. El otro camino se ha centrado en las prestaciones de los sistemas de tiempo real en cuanto a la factibilidad de planificación de la carga de trabajo exigida en función de los requisitos disponibles (procesadores y otros).

En este libro, la atención se centrará principalmente en este último camino. Las razones para ello son tres. Primeramente, las técnicas formales no están todavía lo suficientemente maduras como para basar en ellas sistemas de tiempo real grandes y complejos. Segundo, existe poca experiencia publicada sobre el uso de estas técnicas en sistemas de tiempo real «reales». Finalmente, para conseguir una descripción completa de tales métodos se debería añadir una cantidad considerable de temas, lo que está fuera del alcance de este libro. Esto no significa que el área sea irrelevante respecto a los sistemas de tiempo real. Cada vez se está volviendo más importante la comprensión de, por ejemplo, técnicas formales basadas en CSP, lógicas temporales, lógicas de tiempo real, comprobación de modelos y técnicas de especificación que incorporen la noción de tiempo. Por ejemplo, se puede utilizar RTL (Real-Time Logic) para verificar los requisitos temporales de un sistema, complementando el uso de métodos como VDM y Z en el análisis de los requisitos funcionales.

El éxito de la comprobación de modelos en la verificación de las propiedades funcionales se ha extendido recientemente al dominio del tiempo real. El sistema se modela como un autómata temporizado (esto es, una máquina de estados finitos con relojes), y entonces se utiliza la comprobación del modelo para «probar» que los estados no deseados no pueden ser alcanzados, o que los estados deseados serán alcanzados antes de que cierto reloj interno llegue a cierto tiempo crítico. Esta última propiedad se conoce como *actividad limitada*. Aunque las búsquedas efectuadas en virtud de la comprobación de modelos pueden ser objeto de una explosión de estados, en la actualidad se puede disponer de herramientas con las que hacer frente a problemas de tamaño razonable. Parece que, con el paso de los años, esta técnica se está convirtiendo en una práctica estándar en la industria.

La verificación de un sistema de tiempo real se puede considerar como un proceso en dos etapas:

(1) Verificar los requisitos/diseños: dada una máquina infinitamente rápida y fiable, ¿los requisitos temporales son coherentes y consistentes?; ¿son potencialmente satisfacibles?

(2) Verificar la implementación: con un conjunto finito de recursos hardware (posiblemente no fiables), ¿se pueden satisfacer los requisitos temporales?

Como ya se indicó, (1) puede ser necesario un razonamiento formal (y/o la comprobación de modelos) para verificar que se satisfacen los órdenes temporales (y causales) necesarios. Por ejemplo, si el evento *A* debe ser concluido antes que el evento *B*, pero depende de algún evento *C* que ocurre después de *B*, entonces no importa lo rápido que sea el procesador, ya que nunca se podrán satisfacer esos requisitos. La detección temprana de esta dificultad será, por lo tanto, muy útil. La segunda fase (la verificación de la implementación) es el tema tratado en el siguiente capítulo. El resto de este capítulo se centrará en analizar la forma en que se pueden representar los requisitos temporales en los lenguajes.

12.6 Ámbitos temporales

Para facilitar la especificación de las diversas restricciones temporales presentes en las aplicaciones de tiempo real, resulta útil introducir la noción de **ámbitos temporales**. Dichos ámbitos identifican a un conjunto de sentencias asociadas con una restricción temporal. Los posibles atributos de un ámbito temporal (AT) se muestran en la Figura 12.2, e incluyen, entre otros:

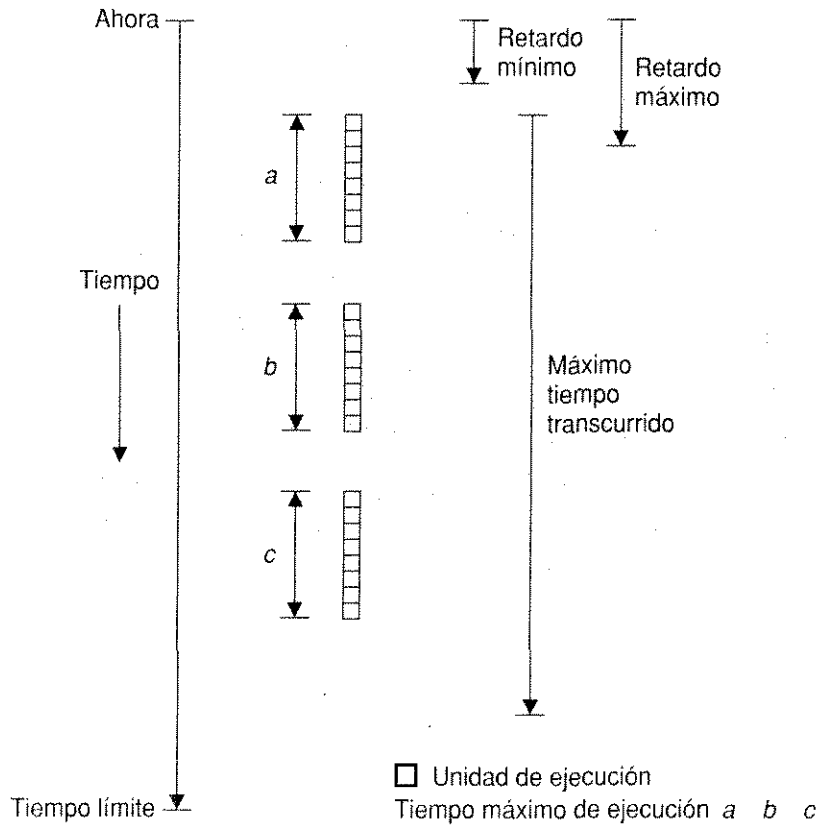


Figura 12.2. Ámbitos temporales.

(1) Límite temporal: el instante de tiempo en el cual la ejecución de un AT debe haber finalizado.

(2) Retardo mínimo: la mínima cantidad de tiempo que debe pasar antes del comienzo de la ejecución de un AT.

(3) Retardo máximo: la máxima cantidad de tiempo que puede pasar antes del comienzo de la ejecución de un AT.

(4) Tiempo máximo de ejecución de un AT.

(5) Lapso de tiempo máximo de un AT.

También pueden darse ámbitos temporales con combinaciones de estos atributos.

Los mismos ámbitos temporales pueden ser **periódicos** o **esporádicos**. Habitualmente, los ámbitos temporales periódicos toman datos o ejecutan un bucle de control, y tienen tiempos límite de espera que deben cumplir. Los ámbitos temporales esporádicos normalmente surgen como consecuencia de eventos asíncronos externos al computador embebido. Estos ámbitos llevan asociados tiempos de respuesta.

En general, la activación de los ámbitos temporales se suele considerar arbitraria, siguiendo, por ejemplo, una distribución de Poisson. Tal distribución permite la llegada de ráfagas de eventos externos, pero no excluye una posible concentración de actividad esporádica. No resulta posible, por lo tanto, realizar un análisis basado en el peor caso (existe una probabilidad no nula de que ocurra cualquier número de eventos no periódicos en un tiempo dado). Para permitir los cálculos del peor caso, se suele definir un periodo mínimo entre dos eventos cualesquiera no periódicos (de un mismo origen). Si este fuera el caso, el proceso involucrado se dice que es **esporádico**. En este libro, el término «aperiódico» se utilizará para el caso general, y el término «esporádico» se reservará para aquellas situaciones en las que se necesite un retardo mínimo.

En muchos lenguajes de tiempo real, los ámbitos temporales están, de hecho, asociados con los procesos que los encarnan. Los procesos se pueden describir como periódicos, aperiódicos o esporádicos, dependiendo de las propiedades de sus ámbitos temporales internos. La mayoría de los atributos temporales enumerados en la anterior lista pueden ser satisfechos mediante:

(1) Ejecución de procesos periódicos a una tasa correcta.

(2) Ejecución de todos los procesos dentro de sus límites temporales.

El problema de satisfacer las restricciones temporales se convierte, por tanto, en el problema de la planificación de los procesos para cumplir los tiempos límite, o **planificación con tiempo límite**. Hay que darse cuenta de que el requisito del retardo máximo se puede conseguir repartiendo el ámbito entre dos procesos con una relación de precedencia. El primero, que representa la fase inicial del AT, puede tener un tiempo límite que asegure el cumplimiento del máximo retardo en el comienzo. Las aplicaciones que necesitan este tipo de estructura suelen ser aquéllas en las que primero se lee de un sensor y luego se produce una salida. Para conseguir un mejor control sobre el momento en el que se lee el sensor, se necesita un límite temporal estricto en es-

ta acción inicial. La salida se puede producir, entonces, con un límite temporal posterior. La variación del momento en el que se lee el sensor o en la salida de un actuador sobre una válvula se denomina **fluctuación** (jitter) de **entrada** o **salida**.

Aunque todos los sistemas informáticos se esfuerzan por ser eficientes, y muchos son descritos como de tiempo real, se necesita una clasificación adicional para reflejar de forma adecuada los distintos niveles de importancia que tiene el tiempo dentro de las aplicaciones. Según se dijo en la introducción, un sistema de tiempo real se dice que es **estricto** si tiene tiempos límite cuyo incumplimiento producirá el fallo del sistema. Por el contrario, un sistema de tiempo real será **flexible** si la aplicación es tolerante con el incumplimiento de los tiempos límite. Un sistema es meramente **interactivo** si no tiene fijado ningún tiempo límite pero se esfuerza por conseguir «tiempos de respuesta adecuados». Para ser más precisos, un proceso con un tiempo límite no estricto puede realizar su servicio más tarde (el sistema será capaz de extraer algún valor de un servicio tardío). Un proceso no estricto que tiene establecido un tiempo límite rígido (en el que, por tanto, un servicio tardío carece de valor), se dice que es **firme**.

La diferencia entre los sistemas de tiempo real estrictos, firmes y flexibles se convierte en algo difuso en el caso de los sistemas tolerantes a fallos. Sin embargo, normalmente resulta apropiado utilizar el término «estricto» si el incumplimiento de un límite temporal produce el disparo de una rutina de recuperación (o de seguridad ante fallos), y los de «firme» o «flexible» si la naturaleza de la aplicación es tolerante a incumplimientos ocasionales de tiempos límite o a incumplimientos no demasiado grandes de tiempos límite. Finalmente, hay que destacar que en la mayoría de los sistemas de tiempo real algúnos de sus tiempos límite serán flexibles, o firmes.

12.6.1 Especificación de procesos y ámbitos temporales

En los sistemas de tiempo real, resulta necesario tratar explícitamente los requisitos temporales, cinco de los cuales fueron explicados con anterioridad. Un esquema general para los procesos periódicos es el siguiente:

```
process periodico_P;
  ...
begin
  loop
    OCIOSO
    comienzo del ámbito temporal
      ...
    final del ámbito temporal
  end;
end;
```

Las restricciones temporales consisten en la existencia de un tiempo máximo y/o mínimo para la fase `OCIOSO`, además de en el requisito de que el final del ámbito temporal se dé en algún tiempo límite. Este tiempo límite puede ser expresado como:

- Un instante de tiempo absoluto.
- El tiempo de ejecución desde el comienzo del ámbito temporal.
- El tiempo transcurrido desde el comienzo del ámbito temporal.

Si se trata de un proceso que se encuentra muestreando datos, entonces debe tener lugar al comienzo del ámbito temporal; la precisión del periodo `OCIOSO` resulta, por lo tanto, importante. El ámbito temporal debería contener el procesamiento necesario de estos datos (o simplemente el almacenamiento), y el tiempo límite al final del ámbito temporal está solo para asegurarse de que el proceso puede ejecutar el ciclo a tiempo para realizar la siguiente lectura.

En el caso de un proceso que está emitiendo señales de control, el ámbito temporal incorporará cuantos cálculos sean necesarios para fijar el valor de la señal (esto puede incluir la toma de lecturas externas). La señal en sí misma se envia al final del ámbito temporal, y por lo tanto los límites temporales estarán asociados con este evento.

En los procesos no periódicos se necesitarán límites temporales similares; en este caso, el ámbito temporal se activará por un evento externo, que normalmente tomará la forma de una interrupción:

```
process noperiodico_P;
  ...
begin
  loop
    esperando una interrupción
    comienzo del ámbito temporal
      ...
    final del ámbito temporal
  end;
end;
```

Es evidente que un proceso periódico tiene un período definido (la frecuencia con la que se ejecuta su ciclo de procesamiento); esta medida también se puede aplicar a los procesos no periódicos, y en este caso significa la tasa máxima a la que el proceso puede realizar un ciclo (o lo que es lo mismo, la tasa más alta de llegadas de interrupciones). Como se ha establecido antes, los procesos no periódicos también se conocen como esporádicos.

En algunos sistemas de tiempo real, los tiempos límite pueden estar asociados con datos que pasan a través de una serie de procesos (lo cual se suele denominar transacción de tiempo real). Para poder planificar estos procesos es necesario repartir el tiempo disponible entre los procesos que manipulan esos datos. Este reparto puede resultar complicado si los tiempos de cada proceso son dinámicos (dependientes de los datos) o, peor aún, si el camino que toman los datos a través de los procesos también es dependiente de los datos. Los lenguajes que no son de tiempo real suelen tratar este problema de forma explícita.

12.7 Los ámbitos temporales en los lenguajes de programación

En las secciones siguientes, se describen varios lenguajes (en lo concerniente a la forma en que soportan los ámbitos temporales). Los lenguajes elegidos representan un amplio espectro de los lenguajes disponibles para ingeniería:

- Ada, occam2 y C/POSIX.
- Euclid de tiempo real y Pearl.
- Java para tiempo real.
- DPS.
- Esterel.

12.7.1 Ada, occam2 y C/POSIX

En consonancia con la mayoría de los lenguajes de tiempo real (las notables excepciones se comentan más adelante), ni Ada, ni occam2 ni C/POSIX soportan la especificación explícita de procesos periódicos o esporádicos con límites temporales; en su lugar se debe utilizar una primitiva de retardo, un temporizador o algo similar dentro de los procesos cíclicos (aunque POSIX permite el establecimiento de un temporizador periódico).

Por ejemplo, en Ada, una tarea periódica debe tomar la siguiente forma:

```
task body T_Periodica is
  Siguiente_Aparicion : Time;
  Intervalo_Aparicion : constant Duration := ...; -- o
  Intervalo_Aparicion : constant Time_Span := Milliseconds(...);
begin
  -- lee el reloj y calcula el siguiente
  -- instante de aparición (Siguiente_Aparicion)
  loop
    -- toma los datos (por ejemplo) o
    -- calcula y envia una señal de control
    delay until Siguiente_Aparicion;
    Siguiente_Aparicion := Siguiente_Aparicion + Intervalo_Aparicion;
  end loop;
end T_Periodica;
```

Compárese esto con la más verborreica representación C/POSIX, que requiere el uso explícito de un temporizador y el manejo de señales.

```
#include <signal.h>
#include <time.h>
#include <pthread.h>
void hilo_periodico() /* destinado para ser un hilo */
{

  int numseñal;                /* señal capturada */
  sigset_t conjunto;                  /* señales por las que esperar */
  struct sigevent señal;       /* informacion sobre la señal */
  timer_t temporizador_periodico;  /* temporizador para un hilo periódico */
  struct itimerspec requerido, anterior;     /* detalles del temporizador */
  struct timespec primero, periodo;          /* comienzo y repeticion */
  long Periodo_Hilo = ....     /* periodo efectivo en nanosegundos */

  /* configurar la interfaz de señales */
  señal.sigev_notify = SIGEV_SIGNAL;
  señal.sigev_signo = SIGRTMIN; /* por ejemplo */
  /* permitir, p.e. 1 seg desde ahora para la inicialización del sistema */
  CLOCK_GETTIME(CLOCK_REALTIME, &primero);   /* conseguir el tiempo actual */
  primero.tv_sec = primero.tv_sec + 1;
  periodo.tv_sec = 0;    /* establecer el valor de repetición del periodo */
  periodo.tv_nsec = Periodo_Hilo;
  requerido.it_value = primero;   /* detalles de inicialización del reloj */
  requerido.it_interval = periodo;
  TIMER_CREATE(CLOCK_REALTIME, &señal, &reloj_periodico);
  SIGEMPTYSET(&conjunto);              /* inicializar conj. señales a null */
  SIGADDSET(&conjunto, SIGRTMIN);      /* sólo interrupciones del reloj */
  TIMER_SETTIME(reloj_periodico, 0, &requerido, &anterior);

  /* pasar al bucle periódico */
  while(1) {
     SIGWAIT(&conjunto, &numseñal);
     /* aquí el código a ejecutar cada periodo */
  }
}

int init()
{
  pthread_attr_t atributos;           /* atributos del hilo */
  pthread_t PT;                       /* apuntador del hilo */

  PTHREAD_ATTR_INIT(&atributos);      /* atributos por defecto */
```

```
  PTHREAD_CREATE(&PT, &atributos,
                 (void *) hilo_periodico, (void *)0);
}
```

Una tarea esporádica Ada que se dispare mediante una interrupción, podría no contener información explícita sobre el tiempo, pero debería normalmente utilizar un objeto protegido para gestionar la interrupción y liberar la tarea para la ejecución:

```
protected Controlador_Esporadico is
  procedure Interrupcion; -- asociado con una interrupción
  entry Espera_Siguiente_Interrupcion;
private
  Llamada_Pendiente : Boolean := False;
end Controlador_Esporadico;
protected body Controlador_Esporadico is

  procedure Interrupcion is
  begin
    Llamada_Pendiente := True;
  end Interrupcion;

  entry Espera_Siguiente_Interrupcion when Llamada_Pendiente is
  begin
    Llamada_Pendiente := False;
  end Espera_Siguiente_Interrupcion;
end Controlador_Esporadico;

task body T_Esporadica is
begin
  loop
    Controlador_Esporadico.Espera_Siguiente_Interrupcion;
    -- accion
  end loop;
end T_Esporadica;
```

Los ejemplos anteriores muestran que en Ada (y en muchos de los otros lenguajes denominados de tiempo real), la única restricción temporal cuyo cumplimiento se puede garantizar es el tiempo mínimo antes del comienzo de un ámbito temporal. Esto se consigue con la primitiva retardo.

12.7.2 Euclid de tiempo real y Pearl

Los lenguajes que soportan la planificación de tiempos límite disponen de las primitivas de temporización apropiadas para la especificación de esos tiempos límite. En Euclid de tiempo real

(Kligerman y Stoyenko, 1986), los procesos son estáticos y no anidados. La definición de cada proceso debe contener información de activación asociada con su comportamiento de tiempo real (se utiliza el término **marco** en lugar de ámbito temporal). Esta información puede tomar una de las dos siguientes formas posibles, dependiendo de si se trata de procesos periódicos o esporádicos:

(1) periodic *infoMarco* first activation *tiempoOEvento*

(2) atEvent *idCondicion infoMarco*

La clausula *infoMarco* define la periodicidad del proceso (incluyendo la tasa máxima de los procesos esporádicos). La forma más simple que puede adoptar es una expresión en unidades de tiempo real:

```
frame exprTiempoReal
```

El valor de las unidades se establece al principio del programa.

Un proceso periódico se puede activar la primera vez de dos maneras diferentes. Puede tener un tiempo de comienzo definido, o puede esperar a que ocurra una interrupción. Incluso, puede responder a cualquiera de estas dos situaciones. La sintaxis para *tiempoOEvento* debe ser, por lo tanto, una de las siguientes

(1) atTime *exprTiempoReal*

(2) atEvent *idCondicion*

(3) atTime *exprTiempoReal* or atEvent *idCondicion*

idCondicion es una variable de condición asociada con una interrupción. También se puede utilizar en los procesos esporádicos.

Para aportar un ejemplo de parte de un programa en Euclid de tiempo real, consideraremos un controlador cíclico de temperatura. Su periodicidad es de 60 unidades (esto es, de un minuto si la unidad de tiempo fijada ha sido el segundo), y tiene que ser activado después de 600 unidades (10 minutos), o cuando se reciba una interrupción *comenzarMonitorizacion*:

```
realTimeUnit := 1.0 % unidad de tiempo = 1 segundo

var Reactor: module % Euclid está basado en módulos
var comenzarMonitorizacion : activation condition atLocation 16#A10D
% Se define una variable de condición asociada con
% una interrupción

  process ControladorTemp : periodic frame 60 first activation
                          atTime 600 or atEvent comienzaMonitorizacion
  % lista de importación
  %
```

```
  % parte de ejecución
  %
  end ControladorTemp
end Reactor
```

Hay que destacar que no existe un bucle en este proceso. El planificador es quien controla que la ejecución se realice con la periodicidad requerida.

Para ilustrar la forma en que se prodría construir este código en Ada, se muestra la parte de proceso o tarea (se tiene que añadir un bucle para forzar la ejecución cíclica de la tarea):

```
task body ControladorTemp is
  -- definiciones, incluyendo
  Siguiente_Aparicion : Duration;
begin
  select
    accept ComienzaMonitorizacion;    -- o una llamada de entrada
                                      -- temporizada en el objeto protegido
  or
    delay 600.0;
  end select;
  Siguiente_Aparicion := Clock + 60.0; -- siguiente instante de liberación
  loop
    -- parte con la ejecución
    delay until Siguiente_Aparicion;
    Siguiente_Aparicion := Siguiente_Aparicion + 60.0;
  end loop;
end ControladorTemp;
```

Esto no sólo resulta más engorroso, sino que el planificador no es consciente de los limites temporales asociados con la tarea. La ejecución correcta dependerá de que la tarea se active otra vez casi inmediatamente después de que el retardo haya expirado.

Pearl

El lenguaje Pearl (Werum y Windauer, 1985) también proporciona información explícita relativa al comienzo, a la frecuencia y a la terminación de los procesos. Una tarea periódica simple, `T`, a repetir cada 10 segundos, se activa mediante:

```
EVERY 10 SEC ACTIVATE T
```

Para activarla en un instante de tiempo concreto (digamos las doce del mediodía de cada día):

```
AT 12:00:00 ACTIVATE ALMUERZO
```

Una tarea esporádica, `S`, activada por una interrupción, `IRT`, se define como:

```
WHEN IRT ACTIVATE S;
```

o, en el caso de que se necesite un retardo inicial de un segundo:

```
WHEN IRT AFTER 1 SEC ACTIVATE S;
```

Aunque la sintaxis es diferente, Pearl proporciona casi la misma funcionalidad que Euclid de tiempo real. Sin embargo, el ejemplo del controlador de temperatura muestra una diferencia significativa: una tarea en Pearl se puede activar por un tiempo planeado o por una interrupción, pero *no* por ambas cosas. Por lo tanto, cualquiera de los dos casos siguientes son admisibles en Pearl:

```
AFTER 10 MIN ALL 60 SEC ACTIVATE ControladorTemp;
WHEN comienzaMonitorizacion ALL 60 SEC ACTIVATE ControladorTemp;
```

El término `ALL 60 SEC` significa repetir periódicamente, después de la primera ejecución, cada 60 segundos.

12.7.3 Java para tiempo real

Java para tiempo real proporciona un soporte similar al de Euclid de tiempo real y Pearl, pero en un marco de trabajo orientado al objeto. Los objetos que que se han de planificar deben implementar la interfaz `Schedulable`, que será considerada en el siguiente capítulo. Además, los objetos deben especificar:

- Sus requisitos de memoria a través de la clase `MemoryParameters` (véase el Capítulo 15).
- Sus requisitos de planificación a través de la clase `SchedulingParameters` (véase el Capítulo 13).
- Sus requisitos de temporización con la clase `ReleaseParameters`.

Esto último se consigue mediante la jerarquía de clases `ReleaseParameters`. La clase abstracta base, definida en el Programa 12.9, proporciona los parámetros generales que necesitan todos los objetos planificables (schedulable). Un objeto planificable puede tener un límite temporal (deadline) y un coste (cost) asociados con cada tiempo en que se activa para su ejecución (release). El coste es la cantidad de tiempo de ejecución que habrá de darle el planificador. Si el objeto está aún en ejecución cuando expira su límite temporal o su coste, se planifica el manejador de eventos asociado. Hay que tener en cuenta que Java para tiempo real no exige que la implementación soporte la monitorización en tiempo de ejecución. Sin embargo, requiere que se detecte el incumplimiento de límites temporales. También se debería destacar que actualmente los eventos de activación para los hilos esporádicos y aperiódicos no están bien definidos. Consecuentemente, resulta difícil entender cómo cualquier implementación podrá detectar que se ha sobrepasado un límite temporal. Por supuesto, un programa puede indicar que no está interesado en los límites temporales incumplidos, mediante el paso de un manejador nulo.

Programa 12.9. La clase `ReleaseParameters`.

```
public abstract class ReleaseParameters
{
   protected ReleaseParameters(RelativeTime coste,
                               RelativeTime tiempoLimite,
                               AsyncEventHandler manejadorDesbordamiento,
                               AsyncEventHandler manejadorIncumplimiento);
   public RelativeTime getCost();
   public AsyncEventHandler getCostOverrunHandler();

   public RelativeTime getDeadline();
   public AsyncEventHandler getDeadlineMissHandler();

   public void setCost(RelativeTime coste);
   public void setCostOverrunHandler(AsyncEventHandler manejador);

   public void setDeadline(RelativeTime tiempoLimite);
   public void setDeadlineMissHandler(AsyncEventHandler manejador);
}
```

Cada subclase de la clase `ReleaseParameters` soporta parámetros de activación periódicos, aperiódicos y esporádicos (según se muestra en el Programa 12.10). Todos los valores de tiempo son relativos al instante de tiempo en el que fue iniciado (activado) el hilo asociado.

Utilizando las clases de parámetros anteriores, es posible expresar las propiedades de temporización de los siguientes objetos planificables (schedulable):

- `RealtimeThread`
- `NoHeapRealtimeThread` (tratado en la Sección 13.14.3)
- `AsyncEventHandler`

Hilos de tiempo real

Un hilo en Java para tiempo real es una extensión de los hilos básicos del lenguaje. La clase se define en el Programa 12.11 (la clase `ProcessingGroupParameters` se considerará en la Sección 13.8.2). Cuando se crea un hilo de tiempo real, es posible hacerlo sin los parámetros de activación. Así, no tendrá especificado ningún requisito temporal.

Los hilos periódicos son aquellos hilos de tiempo real creados con `PeriodicParameters`. Al activar el hilo se invoca el método `start`. Una vez ejecutado, invoca a `waitForNextPeriod` para indicar al planificador que debería ser ejecutable otra vez cuando se cumpla el siguiente periodo. El siguiente fragmento de programa muestra la estructura de un hilo con una periodicidad de 10 milisegundos y un tiempo límite de 5 milisegundos, cuya primera ejecución

Programa 12.10. Las clases `PeriodicParameters`, `AperiodicParameters` y `SporadicParameters`.

```
public class PeriodicParameters extends ReleaseParameters
{
   public PeriodicParameters(HighResolutionTime inicio,
          RelativeTime periodo, RelativeTime coste,
          RelativeTime tiempoLimite,
          AsyncEventHandler manejadorDesbordamiento,
          AsyncEventHandler manejadorIncumplimiento);

   public RelativeTime getPeriod();
   public HighResolutionTime getStart();
   public void setPeriod(RelativeTime periodo);
   public void setStart(HighResolutionTime inicio);
}

public class AperiodicParameters extends ReleaseParameters
{
   public AperiodicParameters(RelativeTime coste,
          RelativeTime tiempoLimite,
          AsyncEventHandler manejadorDesbordamiento,
          AsyncEventHandler manejadorIncumplimiento);
}

public class SporadicParameters extends AperiodicParameters
{
   public SporadicParameters(RelativeTime interLlegadasMin,
          RelativeTime coste, RelativeTime tiempoLimite,
          AsyncEventHandler manejadorDesbordamiento,
          AsyncEventHandler manejadorIncumplimiento);

   public RelativeTime getMinimumInterarrival();
   public void setMinimumInterarrival(RelativeTime minimo);
}
```

es retardada hasta el instante de tiempo absoluto A. El hilo no debería consumir más de 1 milisegundo de tiempo de procesador.

```
public class Periodico extends RealtimeThread
{
     public Periodico (PriorityParameters PP, PeriodicParameters P)
     { ... };

     public void run()
     {
```

Programa 12.11. Un resumen de la clase `RealtimeThread`.

```
public class RealtimeThread extends java.lang.Thread
        implements Schedulable
{

   public RealtimeThread(SchedulingParameters s);
   public RealtimeThread(SchedulingParameters s, ReleaseParameters r);
   public RealtimeThread(SchedulingParameters s, ReleaseParameters rp,
          MemoryParameters m, ProcessingGroupParameters p,
          java.lang.Runnable r);

   // métodos para la interfaz Schedulable
   public synchronized void addToFeasibility();
   public MemoryParameters getMemoryParameters();
   public ReleaseParameters getReleaseParameters();
   public Scheduler getScheduler();
   public SchedulingParameters getSchedulingParameters();
   public synchronized void removeFromFeasibility();
   public void setMemoryParameters(MemoryParameters p);
   public void setReleaseParameters(ReleaseParameters p);
   public void setScheduler(Scheduler s)
          throws IllegalThreadStateException;
   public void setSchedulingParameters(SchedulingParameters s);

   public static RealtimeThread currentRealtimeThread();

   public synchronized void schedulePeriodic();
   // añadir el hilo periódico a la lista de objetos planificables
   public synchronized void deschedulePeriodic();
   // eliminar el hilo periódico de la lista de objetos planificables
   // esto sólo tiene efecto cuando el hilo ha generado waitForNextPeriod
   public boolean waitForNextPeriod() throws IllegalThreadStateException;

   public MemoryArea getMemoryArea();
   public ProcessingGroupParameters getProcessingGroupParameters();
   public void setProcessingGroupParameters(ProcessingGroupParameters p);

   public synchronized void interrupt();
   // sobrecarga java.lang.Thread.interrupt()
   public static void sleep(Clock reloj, HighResolutionTime tiempo)
          throws InterruptedException;
   public static void sleep(HighResolutionTime tiempo)
          throws InterruptedException;
}
```

```
    while(true) {
      // código a ejecutar en cada periodo
      ...
      waitForNextPeriod();
    }
  }
}
```

que podría utilizarse de la siguiente forma:

```
{
  AbsoluteTime A = new AbsoluteTime(...);
  PeriodicParameters P = new PeriodicParameters(
          A, new RelativeTime(10,0),
          new RelativeTime(1,0), new RelativeTime(5,0),
          null, null );
  PriorityParameters PP = new PriorityParameters(...);

  Periodico nuestroHilo = new Periodico(PP, P); //crear el hilo

  nuestroHilo.start(); // liberarlo

}
```

Los hilos aperiódicos y esporádicos se pueden crear de una forma similar. En este caso, sin embargo, para que el hilo sea activado deberá esperar un evento de activación, sin invocar a `waitForNextPeriod`. Ya se mencionó que el evento liberador para los hilos aperiódicos no está actualmente bien definido.

En la Sección 10.5 se trataron las notificaciones asíncronas, y se presentó tanto el modelo de terminación como el de reanudación para la gestión de eventos. Se señaló que un modelo de reanudación para la gestión de notificaciones asíncronas en un proceso multihilo era equivalente a ejecutar un hilo aperiódico/esporádico en respuesta a esa notificación. Éste es el modelo adoptado por Java para tiempo real. Sin embargo, el manejador no es visto como un hilo explícito, sino como un objeto que soporta la interfaz `Schedulable`. En la práctica, un hilo puede estar ligado con más de un manejador, enlazados dinámicamente cuando se dispara el evento. Se puede utilizar la clase `BoundAsyncEventHandler` para enlazar permanentemente el manejador de eventos al hilo (permitiendo así una respuesta más rápida). Dado que el manejador de un evento asíncrono es una clase planificable, debe proporcionar los parámetros de activación apropiados. Como cada manejador de eventos se activa explícitamente, se puede detectar fácilmente cuando sobrepasa su tiempo límite.

12.7.4 DPS

Mientras que Pearl, Euclid de tiempo real y Java asocian los ámbitos temporales con procesos (hilos), y por lo tanto necesitan especificar las restricciones temporales en el proceso mismo,

otros lenguajes, como DPS (Lee y Gehlot, 1985), proporcionan capacidades de temporización local que se aplican a nivel de bloque.

En general, un bloque (ámbito) temporal DPS puede tener que especificar tres requisitos temporales distintos (similares a los requisitos más globales discutidos anteriormente):

- Retardo del comienzo en cierta cantidad de tiempo conocida.
- Ejecución completa en cierto tiempo límite conocido.
- No consumir en la ejecución más de cierto tiempo especificado.

Para ilustrar estas estructuras, considere la importante, aunque simplificada, actividad de tiempo real consistente en hacer y beber café instantáneo:

```
toma_la_taza
pon_cafe_en_la_taza
hierve_agua
pon_agua_en_la_taza
bebe_el_cafe
deja_la_taza
```

El acto de hacer una taza de café no debería llevar más de 10 minutos; beberla es más complicado. Un retardo de 3 minutos debería ser suficiente para no quemarse los labios; la taza debería estar vacía en 25 minutos (entonces debería estar fria) o antes de las 17.00 (las 5 es la hora de irse a casa). Se necesitan dos ámbitos temporales:

```
start elapse 10 do
  toma_la_taza
  pon_cafe_en_la_taza
  hierve_agua
  pon_agua_en_la_taza
end

start after 3 elapse 25 by 17:00 do
  bebe_el_cafe
  deja_la_taza
end
```

En un ámbito temporal que se ejecute repetidamente, resulta útil una construcción cíclica controlada por el tiempo; esto es:

```
from <comienzo> to <fin> every <periodo>
```

Por ejemplo, muchos ingenieros informáticos necesitan tomar café regularmente a lo largo de la jornada de trabajo:

```
from 9:00 to 16:15 every 45 do
  haz_y_bebe_cafe
```

donde la frase `haz_y_bebe_cafe` podría estar compuesta por los dos ámbitos temporales anteriores (excepto la restricción «by» en el bloque dedicado a beber). Hay que destacar que si esto se hiciera así, el máximo tiempo ocupado por cada iteración del bucle sería de 35 minutos; esto es menos que el periodo de cada ciclo, y por lo tanto se necesita un hueco entre dos tazas de café.

Aunque las restricciones temporales a nivel de bloque se pueden especificar de este modo, pueden corresponder con procesos que experimenten distintos límites temporales durante su ejecución; e incluso en ciertas ocasiones pueden no tener límite temporal alguno. Descomponiendo estos procesos en subprocesos, donde cada subproceso es un bloque simple, resulta posible representar todos los límites temporales como restricciones a nivel de proceso. De este modo, el planificador de tiempo de ejecución es más sencillo de implementar; por ejemplo, en algunos de los algoritmos discutidos en el próximo capítulo, será suficiente un esquema de prioridad estático, y el planificador no necesitará estar al tanto explícitamente de los límites temporales.

12.7.5 Esterel

Esterel (Boussinot y de Simone, 1991) es un ejemplo de lenguaje síncrono; otros lenguajes de este tipo son Signal (le Guernic et al., 1991) y Lustre (Halbwachs et al., 1991). Estos lenguajes pretender soportar verificación realizando ciertas suposiciones sobre el comportamiento temporal de sus programas. La suposición principal sobre la que se asienta este modelo computacional es la **hipótesis de sincronía** *ideal* (o *perfecta*) (Berry, 1989):

Los sistemas ideales producen sus salidas en sincronía con sus entradas.

Por lo tanto, se asume que todos los cálculos y las comunicaciones necesitan tiempo cero (esto es, todos los ámbitos temporales son ejecutados instantáneamente). Resulta obvio que ésta es una suposición muy fuerte y no realista; sin embargo, facilita la ordenación temporal de los eventos. Durante la implementación, la *hipótesis de sincronía ideal* se interpreta como que «el sistema debe ejecutarse lo suficientemente rápido como para que se den los efectos de la hipótesis de sincronía». Lo que esto significa, en realidad, es que después de cualquier evento de entrada, todas las salidas asociadas deben ocurrir antes de cualquier otra entrada. Se dice entonces que el sistema «sigue el ritmo» de su entorno. Un programa Esterel es dirigido por eventos, y utiliza señales para la comunicación (las cuales son de difusión). Se parte de una señal regular: `tick` (aunque su granularidad no está definida). Lo siguiente es un módulo periódico (se repite cada 10 pulsos):

```
module periodico;
input tick;
output resultado(integer);
var V : integer in
  loop
```

```
    await 10 tick;
    -- cálculos necesarios para obtener V
    emit resultado(v);
  end
end
```

Un módulo esporádico tiene una forma similar.

Una consecuencia de la hipótesis de sincronía es que todas las acciones son atómicas. Las interacciones entre acciones concurrentes son imposibles, ya que las acciones son instantáneas. En el ejemplo anterior, el resultado es emitido en el mismo instante en el que se dispara el *tick* esperado (y por lo tanto el módulo no sufre deriva local o acumulativa). Un módulo esporádico que esté esperando `resultado` se ejecutará «en el mismo instante» que este módulo periódico. Este comportamiento reduce significativamente el no determinismo. Desafortunadamente, también es una causa potencial de problemas. Considérese lo siguiente:

```
signal S in
  present S else emit S end
end
```

Este programa es incoherente. Si `S` está ausente, entonces es emitida; si estuviera presente, no sería emitida.

Una definición formal de la semántica del comportamiento de Esterel ayuda a eliminar estos problemas (Boussinot y de Simone, 1991). Se puede comprobar la coherencia de un programa. Implementar un programa Esterel legal es directo; con la hipótesis de sincronía siembre es posible construir una máquina de estado finito. Por lo tanto, un programa se mueve de un estado inicial (en el que lee las entradas) a un estado final (en el que produce las salidas). Mientras se mueve entre los estados, no se produce ninguna otra interacción con el entorno. Según se indicó antes, siempre que la máquina de estados finitos (**automaton**) resulte lo suficientemente rápida, se puede considerar cumplida la suposición de atomicidad.

12.8 Tolerancia a fallos

A lo largo de este libro se ha supuesto que los sistemas de tiempo real tienen unos requisitos de fiabilidad altos. Una forma de conseguir esta fiabilidad es incorporar tolerancia a fallos en el software. La inclusión de restricciones temporales introduce la posibilidad de que estas restricciones sean incumplidas; por ejemplo, que los tiempos límite de espera expiren, o que los tiempos límite no se puedan cumplir.

En el caso de los sistemas flexibles, un proceso pudiera necesitar conocer si ha incumplido algún tiempo límite, incluso aunque esto pueda suceder durante su ejecución normal. Más importante aún es que en un sistema estricto (o subsistema), donde los tiempos límite son críticos, incumplir un tiempo límite tiene que disparar alguna rutina de recuperación del error. Si se ha de-

mostrado que el sistema es planificable incluso con los tiempos de ejecución del peor caso, se puede afirmar que no se podrán incumplir los tiempos límite. Sin embargo, en las discusiones de los capítulos anteriores sobre fiabilidad se recomendaba con fuerza la necesidad de una aproximación polifacética a la fiabilidad: probar que nada puede ir mal, e incluir rutinas para tratar adecuadamente los problemas cuando surgen. En esta situación concreta, se puede incumplir un tiempo límite en un sistema «probado» si:

- Los cálculos del tiempo de ejecución en el peor caso, (WCET; worst-case execution time) fueron inexactos.
- Las suposiciones realizadas en la comprobación de planificabilidad no fueron válidas.
- El comprobador de planificabilidad tiene un error.
- El algoritmo de planificación no puede soportar una cierta carga incluso aunque en teoría fuera planificable.
- El sistema está trabajando fuera de sus parámetros de diseño.

En este último caso (por ejemplo, cuando un desbordamiento de información se manifiesta como una tasa inaceptable de interrupciones), los diseñadores del sistema podrían desear un comportamiento de degradación gradual o de fallo seguro.

En resumen, para considerarse tolerante a fallos respecto a los fallos de temporización, es necesario ser capaz de detectar:

- El desbordamiento de un tiempo límite.
- El desbordamiento del tiempo de ejecución en el peor caso.
- Eventos esporádicos que ocurran más a menudo de lo previsto.
- Tiempos límite de espera en las comunicaciones.

Por supuesto que los tres últimos «fallos» de esta lista no indican necesariamente que los tiempos límite vayan a ser incumplidos; por ejemplo, desbordar un WCET en un proceso podría ser compensado por un evento esporádico que ocurriera menos frecuentemente del máximo permitido. Por lo tanto, las fases de confinamiento del daño y de valoración de daños de la tolerancia a fallos (analizadas en la Sección 5.5) deben determinar las acciones a tomar. Se puede utilizar tanto recuperación hacia adelante como hacia atrás. La subsección siguiente discutirá los tres primeros fallos de temporización, ya que los tiempos límite de espera en las comunicaciones fueron tratados en la Sección 12.4.

12.8.1 Detección de errores de temporización y recuperación de errores hacia adelante

Si se tienen que gestionar los errores de temporización, antes deberán ser detectados. Si el entorno de ejecución o el sistema operativo están al tanto de las principales características de un

proceso (tal y como sucede en Java para tiempo real, por ejemplo), serán capaces de detectar problemas y de llamar la atención de la aplicación sobre ellos. En otro caso, será necesario proporcionar primitivas a las aplicaciones que les permitan detectar sus propios errores de temporización.

Detección del desbordamiento de tiempos límite

Este capítulo ya ha mostrado las distintas posibilidades de los lenguajes y de los sistemas operativos para especificar procesos esporádicos y periódicos y sus ámbitos temporales.

El sistema de tiempo de ejecución de Ada no cumple los requisitos de temporización de las tareas de sus aplicaciones, y por lo tanto tiene que proporcionar mecanismos para detectar el desbordamiento de los tiempos límite. Esto se consigue utilizando la capacidad de transferencia asíncrona de control estudiada en la Sección 10.8. Por lo tanto, para detectar el desbordamiento de un tiempo límite de la tarea periódica mostrada en la Sección 12.7.1 se necesita incluir en el bucle principal una sentencia `select then abort`.

```
task body T_Periodica is
  Siguiente_Aparicion : Time;
  Siguiente_TiempoLimite : Time;
  Intervalo_Aparicion : constant Duration := ...; -- o
  Intervalo_Aparicion : constant Time_Span := Milliseconds(...);
  TiempoLimite : constant Time_Span := Milliseconds(...);
begin
  -- lee el reloj y calcula el siguiente
  -- instante de aparición (Siguiente_Aparicion) y
  -- el siguiente tiempo limite (Siguiente_TiempoLimite)
  loop
    select
      delay until Siguiente_TiempoLimite;
      -- detectado aquí un desbordamiento de un tiempo limite
      -- lleva a cabo la recuperacion
    then abort
      -- toma datos (por ejemplo) o
      -- calcula y envia señales de control
    end select;
    delay until Siguiente_Aparicion;
    Siguiente_Aparicion := Siguiente_Aparicion + Intervalo_Aparicion;
    Siguiente_TiempoLimite := Siguiente_Aparicion + TiempoLimite;
  end loop;
end T_Periodica;
```

Puede utilizarse una aproximación similar para detectar un desbordamiento de un tiempo límite en una tarea esporádica.

Uno de los problemas de este enfoque es que asume que la estrategia de recuperación necesita parar la tarea. Esto es, obviamente, una opción, pero existen otros enfoques; por ejemplo, permitir que continúe la ejecución de la tarea pero con otra prioridad distinta. En este caso, una respuesta más apropiada para detectar el desbordamiento de un tiempo límite es desencadenar un evento asíncrono. En Java para tiempo real, la máquina virtual emitirá un evento asíncrono cuando un hilo periódico permanezca en ejecución si el tiempo límite ya ha pasado. Los manejadores de eventos esporádicos en Java para tiempo real no poseen un mecanismo de detección explícita del desbordamiento de un tiempo límite; se supone que son flexibles.

C/POSIX permite crear temporizadores y configurarlos para que cuando expiren generen señales definidas por el usuario (por defecto `SIGALRM`). Por lo tanto, esto permitirá que los procesos puedan decidir cuál es el curso correcto de la acción a perseguir. El Programa 12.12 muestra una interfaz C típica.

En la Sección 5.5.1 se presentó el enfoque de temporizador guardián para la detección de fallos. Éste puede programarse fácilmente usando señales POSIX. Por ejemplo, considere el caso de un hilo (`monitor`) que crea otro hilo (`servidor`) y desea monitorizar su progreso para ver si cumple un tiempo límite. El tiempo límite del `servidor` viene dado por `struct timespec tiempolimite`. El hilo monitor crea un temporizador por proceso indicando el manejador de la señal que deberá ejecutarse si expira el tiempo. Entonces crea el hilo `servidor` y le pasa un apuntador al temporizador. El hilo `servidor` ejecuta su acción, y entonces borra el temporizador. Si la alarma se dispara, es que el `servidor` se ha retrasado.

```
#include <signal.h>
#include <timer.h>
#include <pthread.h>

timer_t temporizador; /* compartido entre el monitor y el servidor */

struct timespec tiempolimite = ...;
struct timespec cero = ...;

struct itimerspec tiempo_alarma, anterior_alarma;

struct sigevent s;

void servidor(timer_t *guardian)
{
  /* realizar el servicio especificado */
  TIMER_DELETE(*guardian);
}

void manejador_guardian(int numseñal, siginfo_t *datos, void *extra)
{
```

Programa 12.12. Una interfaz C para los temporizadores POSIX.

```
#define TIMER_ABSTIME ..

struct itimerspec {
   struct timespec it_value;    /* primera señal del temporizador */
   struct timespec it_interval; /* intervalos sucesivos */
};
typedef ... timer_t;

int timer_create(clockid_t id_reloj, struct sigevent *evp,
                 timer_t *id_temporizador);
   /* Crea un temporizador por proceso utilizando el reloj */
   /* especificado como tiempo base. evp apunta a la estructura */
   /* que contiene toda la información necesaria respecto a */
   /* la señal que ha de generarse. */

int timer_delete(timer_t id_temporizador);
   /* borra el temporizador del proceso */

int timer_settime(timer_t id_temporizador, int indicadores,
                  const struct itimerspec *valor,
                  struct itimerspec *antvalor);
   /* Establece el siguiente tiempo de expiración para el temporizador */
   /* concreto. Si los indicadores estan puestos a TIMER_ABSTIME, */
   /* entonces el temporizador expirará cuando el reloj alcance el */
   /* valor absoluto especificado en *valor.it_value */
   /* si los indicadores NO están puestos a TIMER_ABSTIME, entonces el */
   /* temporizador expirará cuando pase el intervalo valor->it_value */
   /* si *valor.it_interval no es cero, entonces un temporizado periódico */
   /* eliminará cada valor->it_interval tras el pasado valor->it_value. */
   /* Cualquier valor previo de temporizador será devuelto en *antvalor. */

int timer_gettime(timer_t id_temporizador, struct itimerspec *valor);
   /* conseguir los detalles actuales del temporizador */

int timer_getoverrun(timer_t id_temporizador);
   /* si se soportan señales tiempo real, devuelve el número de ellas */
   /* generadas por este temporizador pero no capturadas */

/* Todas estas funciones, excepto timer_getoverrun, devuelven 0 si */
/* tienen éxito, y -1 en caso contrario. timer_getoverrun devuelve */
/* el número de desbordamientos. */
/* Cuando se devuelve una condición de error en cualquiera de ellas */
/* la variable compartida errno contiene la causa del error */
```

```
  /* manejador SIGALRM */

  /* el servidor se ha retrasado: iniciar recuperación */
}

void monitor()
{
  pthread_attr_t atributos;
  pthread_t servidor;

  sigset_t mascara, mascaraant;
  struct sigaction sa, saant;
  int modo_local;

  SIGEMPTYSET(&mascara);
  SIGADDSET(&mascara, SIGALRM);

  sa.sa_flags = SA_SIGINFO;
  sa.sa_mask = mascara;
  sa.sa_sigaction = &manejador_guardian;

  SIGACTION(SIGALRM, &sa, &saant); /* asigna manejador */

  alarm_time.it_value = tiempolimite;
  alarm_time.it_interval = cero; /* temporizador de un disparo */
  s.sigev_notify = SIGEV_SIGNAL;
  s.sigev_signo = SIGALRM;

  TIMER_CREATE(CLOCK_REALTIME, &s, &temporizador);

  TIMER_SETTIME(temporizador, TIMER_ABSTIME, &tiempo_alarma,
                &anterior_alarma);

  PTHREAD_ATTR_INIT(&atributos);
  PTHREAD_CREATE(&servidor, &atributos, (void *)servidor, &temporizador);

  }
```

Sin embargo, según se destacó en la Sección 10.6.6, si el proceso es multihilo, la señal es mandada a todo el proceso, no a un hilo individual. Por ello, es difícil en general determinar cuál de los hilos está desbordando su tiempo límite.

Java para tiempo real también soporta temporizadores. Los Programas 12.13 y 12.14 definen las clases asociadas.

Programa 12.13. La clase `Timer` de Java para tiempo real.

```
public abstract class Timer extends AsyncEvent
{
  protected Timer(HighResolutionTimer tiempo, Clock reloj,
                  AsyncEventHandler manejador);

  public ReleaseParameters createReleaseParameters();

  public AbsoluteTime getFireTime();
  // Obtiene el instante en el que se disparó el evento.

  public void reschedule(HighResolutionTimer tiempo);
  // Modifica el tiempo planificado para este evento.

  public Clock getClock();

  public void disable();
  // Inhabilita el temporizador, evitando que se dispare. Sin embargo,
  // un temporizador inhabilitado continúa la cuenta mientras está
  // inhabilitado.

  public void enable();
  // Habilita el temporizador después de haber sido inhabilitado.
  // El temporizador se disparará inmediatamente si su
  // tiempo ha expirado.

  public void start();
  // comienzo de la cuenta del temporizador
}
```

Con las estructuras de tiempo local de DPS, resulta apropiado asociar los errores de temporización con excepciones:

```
start <restricciones temporización> do
  -- sentencias
exception
  -- manejadores
end
```

Además de los cálculos necesarios para limitar el error, recuperar el error, etc., el manejador podría desear extender el tiempo límite y continuar la ejecución del bloque original. Por ello resulta más apropiado un modelo de *reanudación* que un modelo de *terminación* (véase el Capítulo 6).

En un sistema dependiente del tiempo, también podría ser necesario proporcionar las restricciones temporales a los manejadores. Normalmente, el tiempo de ejecución para el manejador se toma del ámbito temporal mismo; por ejemplo, en el siguiente caso, la secuencia de sentencias será terminada prematuramente después de 19 unidades de tiempo:

Programa 12.14. Las clases `OneShotTimer` y `PeriodicTimer` de Java para tiempo real.

```
public class OneShotTimer extends Timer
{
   public OneShotTimer(HighResolutionTimer tiempo,
                       AsyncEventHandler manejador);
   // Crea una instancia de AsyncEvent que ejecutará su
   // método fire cuando expire el tiempo dado.

   public OneShotTimer(HighResolutionTimer comienzo, Clock reloj,
                       AsyncEventHandler manejador);
   // Crea una instancia de AsyncEvent, basada en el reloj dado,
   // que ejecutará su método fire cuando expire el tiempo dado.
}

public class PeriodicTimer extends Timer
{
   public PeriodicTimer(HighResolutionTimer comienzo,
                        RelativeTime intervalo, AsyncEventHandler manejador);
   // Crea una instancia de AsyncEvent que ejecuta su método fire
   // periódicamente

   public PeriodicTimer(HighResolutionTimer comienzo,
                        RelativeTime intervalo,
                        Clock reloj, AsyncEventHandler manejador);
   // Crea una instancia de AsyncEvent que ejecuta su método
   // fire periódicamente desde el comienzo basado en un reloj concreto

   public ReleaseParameters createReleaseParameters();

   public void setInterval(RelativeTime intervalo);
   // Establece el intervalo de este temporizador
   public RelativeTime getInterval();

   public void fire();
   // Sobrecarga fire de la clase AsyncEvent
   public AbsoluteTime getFireTime();
}
```

```
start elapse 22 do
  -- sentencias
exception
```

```
  when elapse_error within 3 do
     -- manejador
end
```

Como ocurre con todos los modelos de excepción, si el manejador mismo dispara una excepción, ésta sólo se puede tratar en el nivel superior dentro de la jerarquía del programa. Si ocurre un error de temporización dentro de un manejador en el nivel de proceso, entonces el proceso debe terminar (o al menos la iteración actual del mismo). Podría existir algún manejador a nivel de sistema para tratar con los procesos fallidos, o podría dejarse que la aplicación reconociera y tratara tales eventos.

Si se añadieran manejadores de excepciones al ejemplo de hacer café estudiado con anterioridad, el código debería tener la siguiente forma (no se consideran excepciones para errores lógicos como «no hay tazas»). Se supone que sólo `hierve_agua` y `bebe_el_cafe` tienen alguna propiedad de temporización significativa; los errores de temporización se deberán, por lo tanto, al desbordamiento de estas actividades.

```
from 9:00 to 16:15 every 45 do
  start elapse 11 do
     toma_la_taza
     hierve_el_agua
     pon_cafe_en_la_taza
     pon_agua_en_la_taza
  exception
     when elapse_error within 1 do
        apaga_el_hervidor -- por seguridad
        informa_del_fallo
        toma_nueva_taza
        pon_naranja_en_la_taza
        pon_agua_en_la_taza
     end
  end

  start after 3 elapse 26 do
     beber
  exception
     when elapse_error within 1 do
        vacia_la_taza
     end
  end
  reemplaza_la_taza
exception
  when any_exception do
     null -- vé a la siguiente iteración
  end
end
```

Desbordamiento del tiempo de ejecución en el peor caso

Las buenas prácticas de la ingeniería del software intentan confinar las consecuencias de un error a una región bien definida del programa. Entre las posibilidades que ayudan a conseguir esto están los módulos, los paquetes y las acciones atómicas. Sin embargo, que un proceso consuma más tiempo de CPU del previsto, no implica necesariamente que haya incumplido su tiempo límite. Por ejemplo, podría ser un proceso de alta prioridad con un buen margen de tiempo disponible; los procesos que incumplirían su tiempo límite serían procesos con menor prioridad y menos tiempo disponible. Esto implica que hay que ser capaz de detectar cuándo un proceso sobrepasa el peor tiempo de ejecución que el programador había previsto para él. Por supuesto, si un proceso se planifica de forma no apropiativa (y no se bloquea en espera de recursos), su tiempo de ejecución en la CPU será igual al tiempo transcurrido, y entonces se podrán utilizar los mismos mecanismos usados para detectar los desbordamientos de los tiempos límite. Sin embargo, normalmente los procesos se planifican de forma apropiativa, por lo que se complica la medida del tiempo de CPU.

POSIX soporta la monitorización del tiempo de ejecución utilizando su reloj y sus mecanismos de temporización. Hay dos relojes definidos: `CLOCK_PROCESS_CPUTIME_ID` y `CLOCK_THREAD_CPUTIME_ID`. Se pueden utilizar de la misma manera que `CLOCK_REALTIME`. Cada proceso/hilo tiene un reloj asociado con el tiempo de ejecución; las llamadas a:

```
clock_settime(CLOCK_PROCESS_CPUTIME_ID, &algun_valor_de_tiempo);

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &algun_valor_de_tiempo);

clock_getres(CLOCK_PROCESS_CPUTIME_ID, &algun_valor_de_tiempo)
```

establecerán/conseguirán el tiempo de ejecución u obtendrán la resolución del reloj del tiempo de ejecución asociado con el proceso invocador (de forma similar en hilos).

Dos funciones permiten a un proceso/hilo obtener y despues acceder al reloj de otro proceso/hilo.

```
int clock_getcpuclockid(pid_t pid, clockid_t *reloj_id);

int pthread_getcpuclockid(pthread_t hilo_id, clockid_t *reloj_id);
```

Se pueden utilizar los temporizadores definidos en el Programa 12.12 para crear temporizadores, los cuales, sucesivamente, pueden utilizarse para generar señales de proceso cuando el tiempo de ejecución establecido haya expirado. Es la implementación la que decide qué sucede si se utiliza un `timer_create` con un `id_reloj` diferente al del proceso/hilo invocador. Como la señal generada por la expiración del temporizador se dirige al proceso, dependerá de la aplicación el establecer qué hilo atrapará esa señal si expira el temporizador del tiempo de ejecución. Cualquier aplicación puede inhabilitar el uso de temporizadores para un hilo (a consecuencia de la sobrecarga que supone soportar esta capacidad).

Como sucede con la monitorización del tiempo de ejecución, resulta difícil garantizar la precisión del reloj de tiempo de ejecución en presencia de cambios de contexto e interrupciones.

Java para tiempo real permite asociar un valor de «coste» con la ejecución de un objeto planificable. Si la aplicación lo soporta, permitirá que se pueda disparar un evento asíncrono si se supera el coste establecido.

Desbordamiento de eventos esporádicos

Un evento esporádico que se dispare más frecuentemente de lo previsto puede acarrear importantes consecuencias para un sistema que intenta cumplir unos tiempos límite estrictos. Por lo tanto resulta necesario asegurarse de que está prohibido, o bien detectarlo cuando suceda (para emprender alguna acción correctora).

Hay dos caminos básicos para prohibir el desbordamiento de eventos esporádicos. El primero está asociado a los eventos disparados por interrupciones hardware. En la mayoría de las ocasiones, puede inhibirse la interrupción manipulando los correspondientes registros de control del dispositivo.

El otro enfoque es usar una tecnología de servidor esporádico (véase la Sección 13.8.2). Si no se utilizan servidores esporádicos y no se pueden prohibir las interrupciones, entonces es necesario detectar cuándo se están produciendo con excesiva frecuencia. Lamentablemente, la mayoría de los lenguajes de tiempo real y de los sistemas operativos carecen de esta habilidad.

12.8.2 Detección de errores de temporización y recuperación de errores hacia atrás

Como alternativa a la recuperación de errores hacia adelante, un error de temporización puede iniciar una recuperación de errores hacia atrás. Todas las técnicas de recuperación de errores hacia atrás implican test de aceptación. Resulta, por lo tanto, posible incluir un requisito temporal en estos test. El sistema de ejecución puede no superar un test de aceptación si se ha desbordado un límite temporal. Esto se muestra utilizando una capacidad de tiempo límite de espera incorporada dentro de los diálogos y los coloquios (Gregory y Knight, 1985), los cuales fueron presentados en el Capitulo 10. La secuencia de dialogo se convierte ahora en:

```
SELECT
  dialogo_1
OR
  dialogo_2
OR
  dialogo_3
TIMEOUT valor
  -- secuencia de sentencias
ELSE
  -- secuencia de sentencias
END SELECT;
```

Como antes, en la ejecución el proceso intenta primero `dialogo_1`. Si esto tiene éxito, el control se pasa a la sentencia que está a continuación de la sentencia de selección. Si el dialogo fa-

lla, entonces intenta `dialogo_2`, y así sucesivamente. Sin embargo, para permitir que el concepto de coloquio maneje tiempos límite incumplidos, es posible asociar al «select» un tiempo límite. La restricción temporal especifica un intervalo durante el cual el proceso puede ejecutar tantos intentos de diálogo como le sea posible. Los diferentes procesos involucrados en los diálogos pueden tener diferentes valores de tiempo de espera máximo. Si el tiempo de espera máximo expira, entonces el dialogo actual falla, el estado del proceso se restaura al vigente cuando se ejecutó la selección, y se ejecutan las sentencias que se encuentran a continuación de la cláusula `TIMEOUT`. Los demás procesos implicados en el dialogo también fallan, y sus acciones estarán determinadas por las opciones establecidas en sus respectivas sentencias «select». Podrán intentar otro diálogo, esperar un tiempo máximo de espera, o fallar completamente. Junto con la cláusula `else`, las sentencias que siguen al `TIMEOUT` se pueden considerar como el último intento del proceso de tener éxito. Si fallan, entonces, el coloquio falla.

Ahora es posible programar un mecanismo simple de tiempo límite mediante una secuencia de dialogo consistente en un intento de diálogo simple y en un tiempo límite de espera. Por ejemplo, considere un lenguaje de tiempo real parecido a Ada que no soporte explícitamente la especificación de tiempos límite. Una tarea podría recuperarse de un error de tiempo límite mientras se comunica con otras tareas, como sigue:

```
task body example_tiempo_limite is
begin
  loop
     ...
     tiempo := tiempo_calculado_hasta_tiempolimite;
     margen := tiempo_calculado_para_algoritmo_degradado;
     restauracion := tiempo_para_restauracion_estado;
     valor_tiempo_limite := tiempo - (margen + restauracion);
     ...
     SELECT
        dialogo_1;
     TIMEOUT valor_tiempo_limite
        -- secuencia de sentencias para recuperar
        -- un tiempo limite incumplido
     ELSE
        fail;
     END SELECT;
  end loop;
end ejemplo_tiempo_limite;
```

La tarea calcula el intervalo de tiempo hasta el siguiente tiempo límite. Entonces resta de este tiempo el estimado para recuperar un tiempo limite incumplido y el necesario para la restauración del estado. La diferencia entre estos valores es el margen de tiempo. Si el dialogo no se ha completado durante este tiempo, debido a un error de temporización o a un error de diseño, entonces se inicia la secuencia de recuperación.

12.8.3 Cambios de modo y reconfiguración basada en eventos

En general, en las discusiones anteriores se ha supuesto que un tiempo límite incumplido puede ser resuelto por el proceso/hilo que es, en realidad, responsable del tiempo límite. Esto no siempre es así. A menudo las consecuencias de un error de temporización son:

- Otros procesos deben alterar sus límites temporales, o incluso terminar lo que esten haciendo.
- Hay que arrancar nuevos procesos.
- Si los cálculos importantes requieren más tiempo de procesador del asignado, para obtener tiempo extra otros procesos menos importantes pueden tener que ser «suspendidos».
- Los procesos pueden tener que ser «interrumpidos», normalmente para emprender una de las siguientes acciones:
  - Devolver inmediatamente el mejor de los resultados obtenidos.
  - Cambiar a un algoritmo más rápido (aunque, presumiblemente, menos preciso).
  - Olvidarse de lo que estaba haciendo y pasar a estar disponible para recibir nuevas instrucciones: «reinicio sin carga».

Estas acciones suelen conocerse como **reconfiguración basada en eventos**.

Algunos sistemas pueden considerar, adicionalmente, situaciones propensas a incumplimientos de sus tiempos límite. Una buena ilustración de esto se encuentra en los sistemas que sufren **cambios de modo**. Esto se da cuando ocurre algún evento en el entorno que hace que ciertos cálculos ya iniciados no sean útiles. Si el sistema fuera el encargado de realizar estos cálculos, entonces podrían incumplirse otros límites temporales; resulta, pues, necesario finalizar prematuramente los procesos o ámbitos temporales que contienen dichos cálculos.

Para realizar la reconfiguración basada en eventos y cambiar de modo se necesita comunicación entre los procesos implicados. Debido a la naturaleza asíncrona de esta comunicación, es necesario utilizar los mecanismos de notificación asíncrona disponibles en lenguajes como Ada, Java y C/POSIX (véase la Sección 10.5). Estos mecanismos son de bajo nivel; probablemente, lo que realmente se necesite es indicarle al planificador que pare la invocación de ciertos procesos que ahora no resultan necesarios, para comenzar la invocación de otros procesos. Java para tiempo real sigue este camino proporcionando métodos, asociados con los hilos en tiempo real, que informan al planificador si actualmente el hilo no es necesario.

El caso de Euclid de tiempo real resulta interesante al respecto, ya que asocia su mecanismo de gestión de eventos asíncronos con las abstracciones de tiempo real. En Euclid de tiempo real las restricciones temporales están asociadas con procesos, y se pueden definir excepciones numeradas; en cada proceso hay que proporcionar manejadores adecuados. Por ejemplo, considere el siguiente proceso controlador de temperatura que define tres excepciones.

```
process ControlTemp : periodic frame 60 first activation
                      atTime 600 or atEvent iniciaMonitorizacion
  % lista de importación
  handler (num_excep)
     exceptions (200,201,304) % por ejemplo
     imports (var consulta, ...)
     var mensaje : string(80), ...
     case num_excep of
        label 200: % temperatura muy baja
           mensaje := "el reactor esta apagado"
           consulta := mensaje
        label 201: % temperatura muy alta
           mensaje := "la fusión ha comenzado - evacuar"
           consulta := mensaje
           alarma := true % activar dispositivo de alarma
        label 304: % tiempo de espera limite en el sensor
                   % reiniciar dispositivo sensor
     end case
  end handler
  %
  % parte ejecución
  %
end ControlTemp
```

Euclid de tiempo real permite que un proceso genere una excepción en otro proceso. Se soportan tres sentencias de generación distintas, *except*, *deactivate* y *kill*, las cuales, como indican sus nombres, tienen un grado de severidad creciente.

La sentencia *except* (excepción) es esencialmente la misma que la generada en Ada, siendo la diferencia el hecho de que una vez ejecutado el manejador el control se devuelve al punto de generación (modelo de reanudación). Por contra, la sentencia *deactivate* (desactiva) produce la terminación de la iteración del proceso (periódico). El proceso víctima aún ejecutará su manejador de excepción, pero sólo será reactivado en su siguiente período. Para terminar un proceso, se dispone de la sentencia *kill*, que, explícitamente, elimina a un proceso (posiblemente a sí mismo) del conjunto de procesos activos. Se diferencia de un aborto incondicional en que se ejecuta el manejador de excepciones antes de terminar. Esto tiene la ventaja de que un proceso puede realizar alguna «última voluntad» importante. Tiene la desventaja de que un error en el manejador podría aún causar el mal funcionamiento del proceso.

Para ilustrar el uso de estas excepciones, se añadirá algún detalle en la parte de ejecución del ejemplo anterior del controlador de temperatura. Adviertase que, en este ejemplo, las excepciones se generan y se gestionan dentro del mismo proceso de forma síncrona, y en otros procesos de forma asíncroma. Primero, el proceso espera una variable de condición; se especifica un tiempo límite de espera, y se da un número de excepción. (Si se cumple el tiempo límite de espera, se generará la excepción indicada como un *except*.) Entonces se lee y se almacena la temperatura. Los test sobre la temperatura podrán provocar la generación de otras excepciones. Un valor bajo producirá un mensaje apropiado y la desactivación hasta el siguiente periodo; un valor alto pro-

ducirá un mensaje aún más apropiado (aunque un tanto inútil), se generará una excepción en un proceso de alarma, y se terminará el control de temperatura. De este modo, ahora todo el tiempo de procesador disponible se dedicará al procesado de la alarma.

```
process ControlTemp : periodic frame 60 first activation
                      atTime 600 or atEvent iniciaMonitorización
  % lista de importación
  handler (num_excep)
     exceptions (200,201,304) % por ejemplo
     imports (var consulta, ...)
     var mensaje : string(80), ...
     case num_excep of
        ... % como antes
     end case
  end handler

  wait(temperatura_disponible) noLongerThan 10 : 304
  temperaturaActual := ... % E/S de bajo nivel
  log := temperaturaActual
  if temperaturaActual < 100 then
     deactivate ControlTemp : 200
  elseif temperaturaActual > 10000 then
     kill ControlTemp : 201
  end if
  % otros cálculos
end ControlTemp
```

Existen dos mecanismos disponibles en Ada para realizar esta forma de reconfiguración:

- Abort: similar a *kill*.
- ATC: similar a *deactivate*.

Ada permite programar las «últimas voluntades» utilizando una *variable controlada* (véase el Capítulo 4). Según se indicó en la Sección 10.8, la característica ATC es general, y por lo tanto puede tratar con *deactivate* y con la mayoría de las otras formas de reconfiguración basada en eventos. Véanse Real y Wellings (1999a) y Real y Wellings (1999b) para un estudio de los cambios de modo en Ada.

Resumen

La gestión del tiempo presenta tal cantidad de dificultades que coloca a los sistemas embebidos en una categoría aparte dentro de las aplicaciones informáticas. Los lenguajes de tiempo real actuales resultan, a menudo, inadecuados en cuanto la forma de tratar esta área vital.

La introducción de la noción de tiempo en los lenguajes de tiempo real se ha descrito en relación con cuatro requisitos:

- Acceso a un reloj.
- Retardo.
- Tiempos límite de espera.
- Especificación y planificación de tiempos límite.

La sofisticación de los medios aportados para medir el paso del tiempo varía mucho de un lenguaje a otro. Occam2 tiene la utilidad `TIMER`, que solamente devuelve un entero con significado dependiente de la implementación. Ada y Java para tiempo real van algo más lejos, al proporcionar dos tipos abstractos de datos para el tiempo y un conjunto de operadores relacionados con el tiempo. C/POSIX proporciona un conjunto completo de posibilidades para manejar relojes y temporizadores, incluyendo temporizadores periódicos.

Si un proceso desea parar durante un periodo de tiempo, se necesita disponer de una primitiva de retardo (o demora) para evitar la espera ocupada. Tales primitivas siempre garantizan que el proceso será suspendido al menos durante el tiempo indicado, pero no pueden obligar al planificador a que ejecute el proceso inmediatamente después de que haya expirado el retardo. No se puede, por lo tanto, evitar la deriva local, aunque se puede limitar la deriva acumulada que podría surgir a partir de una ejecución repetida de retardos.

Para muchos sistemas de tiempo real, no es suficiente con que el software sea lógicamente correcto; los programas deben también satisfacer restricciones temporales. Lamentablemente, las técnicas usuales de la ingeniería del software son, en general, todavía bastante *ad hoc*. Para facilitar la especificación de las restricciones y los requisitos temporales, resulta útil la introducción de la noción de «ámbito temporal». Los posibles atributos de los ámbitos temporales son entre otros:

- Tiempo límite para completar la ejecución.
- Retardo mínimo antes del comienzo de la ejecución.
- Retardo máximo antes del comienzo de la ejecución.
- Tiempo de ejecución máximo.
- Lapso de tiempo máximo.

En este capítulo se han hecho consideraciones acerca de cómo se pueden especificar los ámbitos temporales.

El grado de importancia de los requisitos de temporización es una forma útil de caracterizar a los sistemas de tiempo real. Las restricciones que deben ser cumplidas se denominan estrictas; aquéllas que pueden ser incumplidas ocasionalmente, o en una pequeña cantidad, se llaman firmes o flexibles.

Para ser tolerante a los fallos de temporización, es necesario ser capaz de detectar:

- El desbordamiento de un tiempo límite.
- El desbordamiento del peor caso del tiempo de ejecución.
- Los eventos esporádicos que ocurren más a menudo de lo previsto.
- El tiempo de espera límite en las comunicaciones.

Después de la detección, se necesitará emprender una reconfiguración basada en eventos.

Lecturas complementarias

Buttazzo, G. C. (1997), *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, New York: Kluwer Academic.

Galton, A. (ed.) (1987), *Temporal Logics and their Application*, London: Academic Press.

Halang, W. A., y Stoyenko, A. D. (1991), *Constructing Predictable Real-Time Systems*, New York: Kluwer.

Joseph, M. (ed.) (1996), *Real-Time Systems: Specification, Verification and Analysis*, Englewood Cliffs, NJ: Prentice Hall.

Koptez, H. (1997), *Real-Time Systems*, New York: Kluwer Academic.

Turski, W. M. (1988), «Time Considered Irrelevant For Real-time Systems», *BIT*, 28(3), pp. 473-488.

Ejercicios

12.1 Explique cómo se puede transformar un sistema de forma que todos los fallos de temporización se manifiesten como fallos de valor. ¿Se puede conseguir tal conversión?

12.2 ¿Debería la llamada temporizada de Ada especificar un tiempo límite de espera en la *terminación* de una cita, en lugar de hacerlo en el comienzo de la misma? Muestre un ejemplo de un caso en el que esto sería útil. ¿Cómo se podría obtener el mismo efecto?

12.3 En un lenguaje de tiempo real parecido a Ada que utilice excepciones para la recuperación de errores hacia adelante, se proponen dos nuevas excepciones predefinidas (potencialmente generables por el sistema de ejecución):

ERROR-TIEMPOLIMITE, generada cuando un bloque de código no ha sido concluido en el intervalo de tiempo requerido (relativo al comienzo del bloque), y ERROR-WCET, ge-

nerada cuando un bloque ha utilizado más tiempo de CPU del especificado (o lo que es lo mismo, ha desbordado su tiempo de ejecución en el peor caso).

Hable sobre:

(1) El modo en que se podrían integrar estas excepciones en un lenguaje parecido a Ada, y en particular cómo se podrían especificar el tiempo límite y el WCET.

(2) Cómo se podrían implementar estas dos excepciones (en el sistema de ejecución).

(3) La forma en que se podrían utilizar en una aplicación.

12.4 Considere la siguiente interfaz de un monitor para un simple mecanismo de retardo:

```
monitor TIEMPO is
   procedure PULSO;
   procedure RETARDO (D : NATURAL);
end TIEMPO;
```

El invocador de `RETARDO` desea ser suspendido durante `D` pulsos. El procedimiento `PULSO` es invocado por alguna rutina de reloj. Al ser invocado, cada proceso bloqueado en `RETARDO` se despierta, se decrementa algún contador (el cual se inicializa a `D`), y se sale del monitor si el contador ha llegado a cero (en otro caso se vuelve a bloquear).

Muestre cómo se puede implementar el cuerpo de este monitor. Utilice variables de condición con las operaciones `wait` y `signal` definidas sobre ellas. Indique cuál es la semántica asociada a `signal`.

12.5 Discuta el modo en que un paquete `Tiempo` en Ada podría proporcionar el mismo mecanismo de retardo que el dado en el Ejercicio 12.4 (*no* utilice la sentencia `delay` de Ada).

12.6 ¿Se puede, en Java para tiempo real, detectar el desbordamiento de un tiempo límite en un hilo periódico?

12.7 ¿Cuál es el papel del tiempo límite en la especificación de un manejador de eventos de Java para tiempo real?

12.8 ¿Hasta qué punto se puede realizar la reconfiguración basada en eventos en Java para tiempo real?

Capítulo 13

Planificación

En un programa concurrente, no es necesario especificar el orden exacto en el cual se ejecutan los procesos. Para forzar las restricciones de ordenación locales se utilizan primitivas de sincronización tales como la exclusión mutua, pero el comportamiento general del programa muestra un no determinismo significativo. Si el programa es correcto, entonces su comportamiento funcional sera el mismo independientemente del comportamiento interno o de los detalles de implementación. Por ejemplo, cinco procesos independientes pueden ejecutarse de forma no apropiativa de 120 formas distintas en un único procesador. En un sistema multiprocesador o con un comportamiento apropiativo, existen muchas más formas de entrelazarlos.

Aunque las salidas del programa sean idénticas en todos estos posibles entrelazamientos, el comportamiento temporal puede variar considerablemente. Si uno de los cinco procesos tiene un tiempo límite ajustado, entonces quizás sólo se cumplirán los requisitos temporales en aquellos entrelazamientos en los que se ejecute el primero. Un sistema de tiempo real necesita restringir el no determinismo que se da en los sistemas concurrentes. Este proceso se conoce como planificación (scheduling). En general, un esquema de planificación proporciona dos elementos:

- Un algoritmo para ordenar el uso de los recursos del sistema (en concreto de las CPU).
- Un modo de predecir el comportamiento en el peor caso del sistema cuando se aplica el algoritmo de planificación.

Las predicciones son útiles para confirmar que se pueden satisfacer los requisitos temporales del sistema.

Un esquema de planificación puede ser estático (si las predicciones se realizan antes de la ejecución) o dinámico (si se toman decisiones en tiempo de ejecución). Este capítulo estudia principalmente los esquemas estáticos. La mayor atención se centra en los esquemas apropiativos basados en prioridad. En ellos, se asignan prioridades a los procesos de forma que siempre se ejecute el proceso con la mayor prioridad (si no es retrasado o suspendido). Un esquema de planificación implicará un algoritmo de asignación de prioridades y un test de planificabilidad.

13.1 Modelo de proceso simple

Dado un programa arbitrariamente complejo, no es tarea fácil analizarlo para poder predecir su comportamiento en el peor caso. Por tanto, resulta imprescindible imponer algunas restricciones en la estructura de los programas concurrentes de tiempo real. Esta sección presentará un modelo muy simple, que permitirá describir algunos esquemas de planificación estándar. El modelo se generalizará más tarde en las últimas secciones de este capítulo (y será examinado en más profundidad en los Capítulos 14 y 16). Este modelo básico tiene las siguientes características:

- Se supone que la aplicación está compuesta por un conjunto fijo de procesos.
- Todos los procesos son periódicos, con periodos conocidos.
- Los procesos son completamente independientes unos de otros.
- Se ignoran todas las sobrecargas del sistema, tiempos de cambio de contexto y demás (se supone que tienen coste cero).
- Todos los procesos tienen tiempos límite iguales a sus periodos (o, lo que es lo mismo, cada proceso debe acabar antes de ser ejecutado nuevamente).
- Todos los procesos tienen tiempos de ejecución constantes en el peor caso.

Una consecuencia de la independencia entre procesos es que se puede suponer que en algún instante de tiempo todos los procesos son ejecutados a la vez. Esto representará la carga máxima para el procesador, y se conoce como un **instante crítico**.

La Tabla 13.1 muestra un conjunto estándar de notaciones relacionadas con las características de los procesos.

Tabla 13.1. Notación estándar.

Notación	Descripción
B	Tiempo de bloqueo del proceso en el peor caso (si fuera aplicable)
C	Tiempo de ejecución del proceso en el peor caso (WCET)
D	Tiempo límite del proceso
I	Tiempo de interferencia del proceso
J	Fluctuación en la ejecución del proceso
N	Número de procesos en el sistema
P	Prioridad asignada al proceso (si fuera aplicable)
R	Tiempo de respuesta del proceso en el peor caso
T	Tiempo mínimo entre ejecuciones del proceso (periodo del proceso)
U	Utilización de cada proceso (igual a $C=T$)
$a-z$	Nombre del proceso

13.2 El enfoque del ejecutivo cíclico

Con un conjunto fijo de procesos puramente periódicos, resulta posible presentar un esquema de planificación completo en el que la ejecución repetida de la planificación consiga que todos los procesos se ejecuten a la tasa apropiada. El ejecutivo cíclico es, en esencia, una tabla de llamadas a procedimientos, donde cada procedimiento representa parte del código de un «proceso». A la tabla completa se le conoce como **ciclo principal**, y habitualmente consta de cierto número de **ciclos secundarios**, cada uno de ellos de duración fija. De esta forma, por ejemplo, cuatro ciclos secundarios de 25 ms de duración conforman un ciclo principal de 100 ms. Durante la ejecución, una interrupción de reloj cada 25 ms permite que el planificador realice rondas entre los cuatro ciclos secundarios. La Tabla 13.2 proporciona un conjunto de procesos que deben ser implementados mediante un ciclo principal simple de cuatro ranuras. En la Figura 13.1 se muestra una posible implementación de un ejecutivo cíclico que ilustra el trabajo del procesador en un instante de tiempo. El código de tal sistema podría tener una forma simple como la siguiente:

```
loop
   espera_interrupción;
   procedimiento_para_a;
   procedimiento_para_b;
   procedimiento_para_c;
   espera_interrupción;
   procedimiento_para_a;
   procedimiento_para_b;
   procedimiento_para_d;
   procedimiento_para_e;
   espera_interrupción;
   procedimiento_para_a;
   procedimiento_para_b;
   procedimiento_para_c;
   espera_interrupción;
   procedimiento_para_a;
   procedimiento_para_b;
   procedimiento_para_d;
end loop;
```

Tabla 13.2. Ejecutivo cíclico de un conjunto de procesos.

Proceso	Período, T	Tiempo de ejecución, C
a	25	10
b	25	8
c	50	5
d	50	4
e	100	2

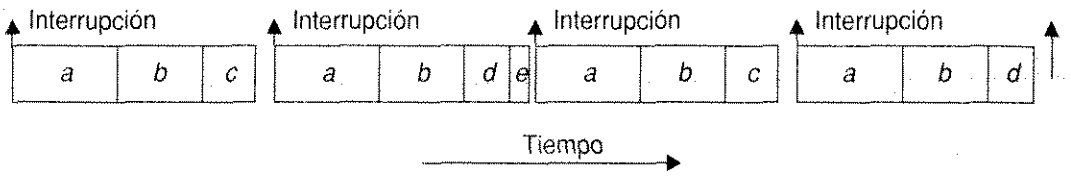


Figura 13.1. Línea de tiempo para el conjunto de procesos.

Incluso este simple ejemplo permite ilustrar algunas características importantes de este enfoque:

- En realidad, estos procesos no existen en tiempo de ejecución: cada ciclo secundario es una secuencia de llamadas a procedimientos.
- Estos procedimientos comparten un espacio de direcciones, por lo que pueden pasar datos entre ellos. Éstos no necesitan ser protegidos (con un semáforo, por ejemplo), porque es imposible el acceso concurrente.
- Todos los periodos de los «procesos» deben ser múltiplos del tiempo de ciclo secundario.

Esta última propiedad representa uno de los principales inconvenientes del enfoque del ejecutivo cíclico; el resto son (Locke, 1992):

- La dificultad para incorporar procesos esporádicos.
- La dificultad para incorporar procesos con periodos grandes; el tiempo del ciclo principal es el máximo periodo acomodable sin planificación secundaria (un procedimiento en el ciclo principal que llame a un proceso secundario cada N ciclos principales).
- La dificultad a la hora de construir el ejecutivo cíclico.
- Cualquier «proceso» con un tiempo de cálculo notable podrá tener que ser dividido en un número fijo de procedimientos de tamaño fijo (lo que podría romper la estructura del código desde una perspectiva de la ingeniería del software, y por lo tanto ser propenso a error).

Si puede construirse un ejecutivo cíclico, no será necesario ningún test de planificabilidad más (el esquema es «prueba mediante construcción»). Sin embargo, en sistemas de alta utilización, la construcción del ejecutivo es problemática. Se puede establecer una analogía con el clásico problema del empaquetado en cajas. En ese problema, hay que colocar ciertos elementos de distintos tamaños (de una sola dimensión) en el menor número posible de cajas, de forma que éstas no se desborden. El problema del empaquetado, o de la mochila, es NP-completo, y por lo tanto es irresoluble para problemas de cierto tamaño (un sistema realista típico quizás puede contener 40 ciclos secundarios y 400 entradas). En este caso habrá que utilizar esquemas heurísticos subóptimos.

Aunque para el caso de sistemas periódicos simples el ejecutivo cíclico se mantiene como una estrategia de implementación apropiada, los esquemas de planificación basados en procesos aportan un enfoque más flexible y fácil de acomodar. Estas aproximaciones serán, por lo tanto, el tema del resto de este capítulo.

13.3 Planificación basada en procesos

Con el enfoque del ejecutivo cíclico, la ejecución consiste en una secuencia de invocaciones a procedimientos. La noción de proceso (hilo) no se preserva durante la ejecución. Un enfoque alternativo es soportar la ejecución de procesos de forma directa (como es norma en los sistemas operativos de propósito general), y determinar cuál es el proceso que deberá ejecutarse en cada instante de tiempo, mediante uno o más atributos de planificación. Con este enfoque, un proceso está limitado a estar en uno de los posibles *estados* siguientes (suponiendo que no exista comunicación entre procesos):

- Ejecutable.
- Suspendido en espera de un evento temporizado (apropiado para procesos periódicos).
- Suspendido en espera de un evento no temporizado (apropiado para procesos esporádicos).

13.3.1 Alternativas de planificación

En general, existe un gran número de aproximaciones distintas a la planificación. En este libro se van a considerar tres de ellas.

- Planificación de prioridad estática (FPS; Fixed-Priority Scheduling). Ésta es la modalidad más ampliamente utilizada, y es el principal objetivo de este capítulo. Cada proceso tiene cierta prioridad fija, **estática**, que se calcula antes de su ejecución. Los procesos ejecutables se ejecutan en el orden determinado por su prioridad. *En los sistemas de tiempo real, la «prioridad» de un proceso se deriva de sus requisitos de temporización, no de su importancia para el correcto funcionamiento o para la integridad del sistema.*
- Planificación primero el tiempo límite más temprano (EDF; Earliest Deadline First). En este caso, los procesos ejecutables se ejecutan en el orden determinado por los tiempos límite absolutos: el siguiente proceso a ejecutar es aquél con el tiempo límite más próximo (el más cercano). Aunque se suelen conocer los tiempos límite relativos de cada proceso (por ejemplo, 25 ms después de la ejecución), los tiempos límite absolutos se calculan en tiempo de ejecución, y por ello se dice que el esquema es **dinámico**.
- Planificación basada en el valor (VBS; Value-Based Scheduling). Si el sistema puede llegar a sobrecargarse, no será suficiente el simple uso de prioridades estáticas o de tiempos límite; se precisará un esquema más **adaptable**. Esto suele realizarse asignando cierto *valor* a cada proceso y empleando un algoritmo de planificación en línea basado en el valor para decidir cuál será el siguiente proceso a ejecutar.

Como ya se indicó, el grueso de este capítulo se centra en el estudio de la forma en que los distintos lenguajes de tiempo real y los sistemas operativos estándar soportan FPS. El uso de EDF también es importante, y en la siguientes argumentaciones se realiza alguna consideración sobre sus bases analíticas. Al final de este capítulo, en la Sección 13.13, se añade una breve descripción del uso de VBS.

13.3.2 Apropiación y no apropiación

En un esquema de planificación basado en prioridad, puede que un proceso de alta prioridad tenga que ser ejecutado mientras se está ejecutando otro de menor prioridad. En un esquema **apropiativo**, se producirá inmediatamente un cambio de procesos a favor del proceso de alta prioridad. Alternativamente, en un caso **no apropiativo** el proceso de menor prioridad podrá acabar su ejecución antes de que se ejecute el otro proceso. En general, los esquemas apropiativos mejoran la reacción de los procesos de alta prioridad, y por lo tanto se les prefiere. Entre los extremos de la apropiación y la no apropiación existen estrategias alternativas que permiten que un proceso de menor prioridad continúe su ejecución durante un tiempo limitado (aunque no necesariamente hasta que termine). Estos esquemas se conocen como de **apropiación diferida** o de **distribución cooperativa**, y se considerarán de nuevo en la Sección 13.12.1. Pero antes, se supondrá que el despacho es apropiativo. Los esquemas como EDF y VBS pueden tomar formas apropiativas o no apropiativas.

13.3.3 FPS y la asignación de prioridad de tasa monotónica

A partir del modelo básico presentado en la Sección 13.1, aparece un esquema óptimo de asignación de prioridades denominado **tasa monotónica** (rate monotonic). A cada proceso se le asigna una (única) prioridad basada en su periodo: mayor prioridad para periodos menores (esto es, para dos procesos i y j, $T_i < T_j \Rightarrow P_i > P_j$). Esta asignación es óptima en el sentido de que si cualquier conjunto de procesos puede ser planificado con un esquema de prioridades estático (utilizando una planificación apropiativa basada en prioridades), entonces el conjunto dado también se puede planificar con un esquema de planificación de tasa monotónica. La Tabla 13.3 ilustra un conjunto de cinco procesos y sus prioridades relativas para un comportamiento óptimo. Téngase en cuenta que las prioridades se representan mediante enteros, y que a enteros más grandes mayor prioridad. Tenga precaución al consultar otros libros y artículos de planificación basada en prioridades, ya que a menudo se ordenan las prioridades de otro modo; la prioridad 1 es la mayor. En este libro, *la prioridad 1 es la menor*, ya que éste es el caso habitual en la mayoría de los lenguajes de programación y sistemas operativos.

Tabla 13.3. Ejemplo de asignación de prioridades.

Proceso	Periodo, T	Prioridad, P
a	25	5
b	60	3
c	42	4
d	105	1
e	75	2

Tabla 13.4. Límites de utilización.

N	Límite de utilización
1	100,0%
2	82,8%
3	78,0%
4	75,7%
5	74,3%
10	71,8%

13.4 Test de planificabilidad basados en la utilización

Esta sección describe un test de planificabilidad muy simple para FPS, el cuál, aunque no es exacto, resulta atractivo debido a su simplicidad.

Liu y Layland (1973) demostraron que considerando sólo la utilización del conjunto de procesos, puede obtenerse un test de planificabilidad (cuando se emplea una ordenación de tasa monotónica). Si se cumple la siguiente condición, todos los N procesos cumplirán sus tiempos límite (el sumatorio calcula la utilización total del conjunto de procesos):

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq N(2^{1/N} - 1) \qquad \textbf{(13.1)}$$

La Tabla 13.4 muestra los límites de utilización (como porcentajes) para valores pequeños de N. Para valores de N grandes, el límite se aproxima asintóticamente al 69,3 por ciento, y siempre será planificable, utilizando un esquema de planificación apropiativo basado en prioridades, con las prioridades calculadas mediante el algoritmo de tasa monotónica.

Se tomarán tres ejemplos sencillos para ilustrar el modo de utilización de este test. En estos ejemplos, no se definen las unidades de tiempo (en magnitudes absolutas), ya que todos los valores (*Ts*, *Cs*, etc.) se expresan en las mismas unidades. Así pues, la unidad de tiempo considerada, en estos y en los anteriores ejemplos, será un *tick* (pulso) en cada base temporal concreta.

La Tabla 13.5 contiene tres procesos a los que se les han asignado prioridades mediante el algoritmo de tasa monotónica (el proceso *c* tiene la prioridad más alta y el *a* la menor). Su utilización combinada es 0,82 (o el 82 por ciento). Esto está por encima del umbral para tres procesos (0,78), por lo que este conjunto de procesos no pasa el test de utilización.

El comportamiento real de este conjunto de procesos se muestra dibujando su línea temporal. La Figura 13.2 muestra cómo los tres procesos se podrían ejecutar si todos ellos comenzaran sus

Tabla 13.5. El conjunto de procesos *A*.

Proceso	Periodo, *T*	Tiempo ejecución, *C*	Prioridad, *P*	Utilización, *U*
a	50	12	1	0,24
b	40	10	2	0,25
c	30	10	3	0,33

ejecuciones en el instante 0. Hay que destacar que, en el instante 50, el proceso *a* ha consumido sólo 10 pulsos de ejecución (cuando necesita 12), y por lo tanto ha incumplido su primer límite temporal.

Las líneas temporales son un medio útil para mostrar los patrones de ejecución. A modo de ejemplo, la Figura 13.2 se ha representado como un **diagrama de Gantt** en la Figura 13.3.

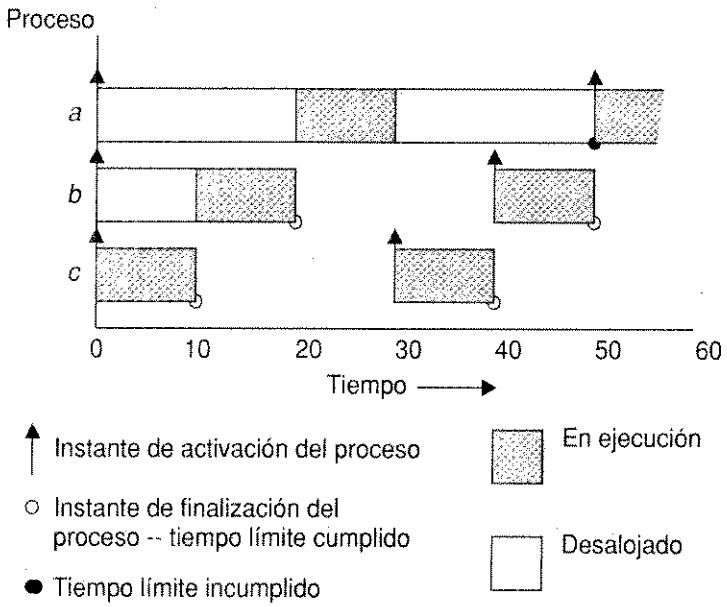


Figura 13.2. Línea de tiempo para el conjunto de procesos *A*.

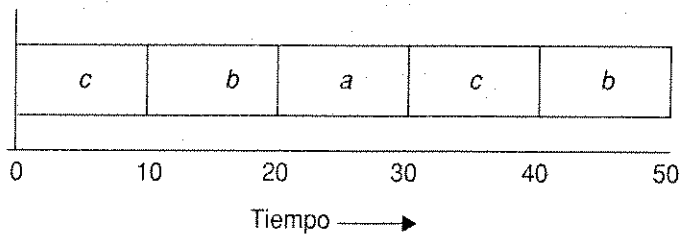


Figura 13.3. Diagrama de Gantt para el conjunto de procesos *A*.

El segundo ejemplo está contenido en la Tabla 13.6. Ahora la utilización combinada es de 0,775, que está por debajo del límite, y por lo tanto se garantiza que el conjunto de procesos cum-

plirá todos sus tiempos límite. Si se dibujara una línea de tiempo para este conjunto, todos los límites temporales serían satisfechos.

Tabla 13.6. Conjunto de procesos *B*.

Proceso	Periodo, *T*	Tiempo ejecución, *C*	Prioridad, *P*	Utilización, *U*
a	80	32	1	0,400
b	40	5	2	0,125
c	16	4	3	0,250

Aunque resulten engorrosas, las líneas temporales pueden utilizarse para probar la planificabilidad. Pero, ¿hasta dónde se ha de continuar dibujando la temporal antes de que se pueda concluir que el futuro no guarda sorpresas? Para conjuntos de procesos que comparten un tiempo de activación común (esto es, un *instante crítico*), se puede demostrar que una línea temporal igual al tamaño del periodo más largo es suficiente (Liu y Layland, 1973). De forma que, si todos los procesos cumplen su primer límite temporal, entonces cumplirán todos sus límites futuros.

En la Tabla 13.7 se muestra un último ejemplo. Éste es, otra vez, un sistema de tres procesos, pero la utilización combinada es del 100 por ciento, de forma que, evidentemente, no aprueban el test. Sin embargo, en tiempo de ejecución el comportamiento parece correcto: todos los límites temporales se cumplen hasta el instante 80 (véase la Figura 13.4). Aunque el conjunto de procesos no pasa el test, en tiempo de ejecución no se incumple ningún tiempo límite. Por lo tanto, se dice que el test es **suficiente** pero no **necesario**. Si un conjunto de procesos pasa el test, entonces *cumplirá* todos los tiempos límite; si lo falla, *puede o no* fallar en tiempo de ejecución. Un último punto digno de atención es que este test basado en la utilización sólo aporta un respuesta del tipo sí/no. No da ninguna indicación sobre los tiempos de respuesta de los procesos. Esto se remedia mediante el enfoque de tiempo de respuesta descrito en la Sección 13.5.

Tabla 13.7. Conjunto de procesos *C*.

Proceso	Periodo, *T*	Tiempo ejecución, *C*	Prioridad, *P*	Utilización, *U*
a	80	40	1	0,50
b	40	10	2	0,25
c	20	5	3	0,25

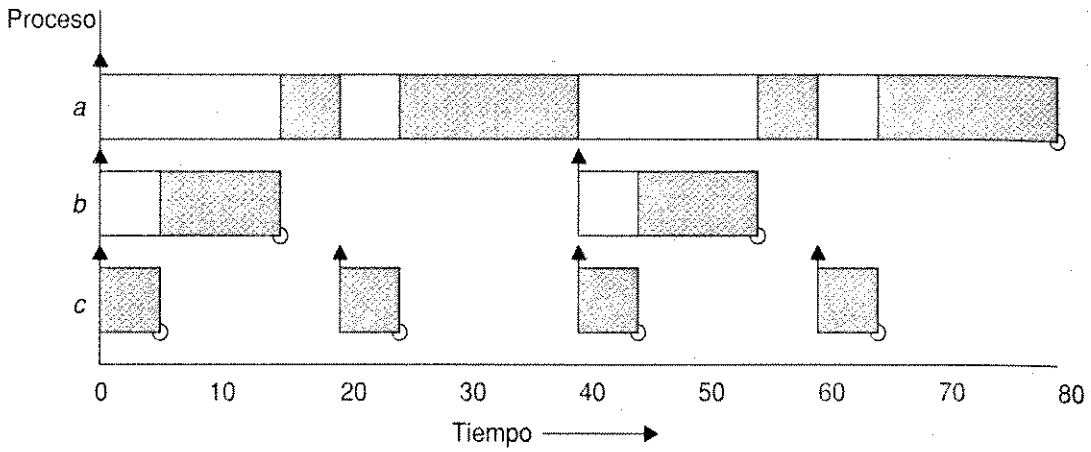


Figura 13.4. Línea de tiempo para el conjunto de procesos *C*.

13.4.1 Test de planificabilidad basado en la utilización para EDF

El artículo original de Liu y Layland (1973) no sólo introdujo un test basado en la utilización para la planificación con prioridades estáticas, sino que también propuso uno para EDF:

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i} \right) \leq 1 \tag{13.2}$$

Evidentemente, éste es un test mucho más simple. Si la utilización del conjunto de procesos es menor que la capacidad total del procesador, entonces todos los tiempos límite se cumplirán (para el modelo de proceso simple). En este sentido, EDF es superior a FPS; siempre puede planificar cualquier conjunto de procesos que sea planificable por FPS, pero no todos los conjuntos de procesos que pasan el test podrán planificarse utilizando prioridades estáticas. Dada esta ventaja, resulta razonable preguntarse por qué EDF no es el método preferido para la planificación basada en prioridades. La razón es que FPS tiene varias ventajas sobre EDF:

- FPS es más sencillo de implementar, ya que el atributo de planificación (*prioridad*) es estático; EDF es dinámico, y por lo tanto necesita un sistema de tiempo de ejecución más complejo, que inducirá una mayor sobrecarga en el sistema.
- Resulta más sencillo incorporar procesos sin tiempos límites en FPS (no hay más que asignarles una prioridad); otorgar a un proceso un tiempo límite arbitrario es más artificial.
- El atributo de tiempo límite no es el único parámetro importante. De nuevo, resulta más sencillo incorporar los otros factores en la noción de prioridad que en la noción de tiempo límite.
- Durante las situaciones de sobrecarga (donde se puede dar una condición de fallo), el comportamiento de FPS es más predecible (los procesos con menor prioridad son aquéllos que pueden incumplir sus tiempos límites); EDF es impredecible bajo situaciones de sobrecarga, y esto puede acarrear un efecto dominó en el cual un gran número de procesos incumplan sus tiempos límite. Esto se considera de nuevo en la Sección 13.13.

- El test basado en la utilización, para el modelo simple, es engañoso, ya que es necesario y suficiente para EDF pero sólo suficiente para FPS. Por eso, las mayores utilizaciones suelen conseguirse con FPS.

A pesar de este último punto, EDF tiene una ventaja sobre FPS producto de su alta utilización, y por eso continua siendo estudiado y utilizado en algunos sistemas experimentales.

13.5 Análisis del tiempo de respuesta para FPS

Los test basados en la utilización para FPS tienen dos inconvenientes significativos: no son exactos, y no son realmente aplicables a un modelo de proceso más general. Esta sección proporciona una forma diferente de test. El test tiene dos etapas. Primero, se realiza una aproximación analítica para predecir el tiempo de respuesta en el peor caso para cada proceso. Estos valores son comparados, trivialmente, con los tiempos límites de los procesos. Eso requiere analizar individualmente cada proceso.

Para el proceso de mayor prioridad, el tiempo de respuesta en el peor caso será igual a su tiempo de ejecución (esto es, $R = C$). Los demás procesos sufrirán cierta **interferencia** por parte de los procesos con mayor prioridad, debido al tiempo invertido en los procesos de mayor prioridad mientras un proceso de menor prioridad es ejecutable. Para cualquier proceso i:

$$R_i = C_i + I_i \tag{13.3}$$

donde I_i es la máxima interferencia que el proceso i puede sufrir en cualquier intervalo de tiempo $[t; t + R_i)$.[1] La máxima interferencia se da cuando se activan todos los procesos de alta prioridad al mismo tiempo que el proceso i (es decir, en un instante crítico). Sin pérdida de generalidad, se puede suponer que todos los procesos se activan en el instante 0. Considere un proceso (j) con una prioridad mayor que i. Dentro del intervalo $[0, R_i)$, será activado cierto número de veces (al menos una). Una expresión simple para este número de activaciones se obtiene mediante una función techo:

$$Número\_De\_Liberaciones = \left\lceil \frac{R_i}{T_j} \right\rceil$$

La función techo ($\lceil\rceil$) obtiene el menor entero mayor que el fraccionario sobre el que actúa. De forma que el techo de 1/3 es 1; el de 6/5 es 2; y el de 6/3 es 2. No se considera la definición del techo para un número negativo.

Por ello, si R_i es 15 y T_j es 6, entonces se producen 3 activaciones del proceso j (en los instantes 0, 6 y 12). Cada activación del proceso j producirá una interferencia de C_j. Por lo tanto:

$$Máxima\_Interferencia = \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

[1] Para este análisis se ha utilizado un modelo de tiempo discreto, toda vez que los intervalos de tiempo deben ser cerrados al principio (denotado por «[») y abiertos al final (denotado por «)»). Por eso, un proceso puede acabar su ejecución en el mismo pulso en el que se activa un proceso de mayor prioridad.

Si $C_j = 2$, entonces en el intervalo [0, 15) existen 6 unidades de interferencia. Cada proceso de alta prioridad interfiere con el proceso i, y por lo tanto:

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_i} \right\rceil C_j$$

donde $hp(i)$ es el conjunto de procesos de prioridad mayor (que i). Sustituyendo este valor en la Ecuación (13.3), se obtiene (Joseph y Pandya, 1986):

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_i} \right\rceil C_j \qquad \textbf{(13.4)}$$

Aunque la formulación de la ecuación de interferencia es exacta, la cantidad real de interferencia es desconocida, ya que se desconoce R_i (éste es el valor que estamos calculando). La Ecuación (13.4) tiene R_i en ambos lados, pero es difícil de resolver debido a las funciones techo. En realidad, es un ejemplo de ecuación de punto fijo. En general, existirán varios valores de R_i que satisfarán la Ecuación (13.4). El más pequeño de los valores de R_i representa el tiempo de ejecución para el proceso en el peor caso. El modo más simple de resolver la Ecuación (13.4) es mediante una relación de recurrencia (Audsley et al., 1993a):

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_i} \right\rceil C_j$$

El conjunto de valores $\{w_i^0, w_i^1, w_i^2, ..., w_i^n, ...\}$ es, claramente, monótono no decreciente. Cuando $w_i^n = w_i^{n+1}$, se habrá encontrado la solución a la ecuación. Si $w_i^0 < R_i$, entonces w_i^n será la solución más pequeña, y por lo tanto será el valor buscado. Si la ecuación no tiene una solución, entonces los valores continuarán creciendo (esto se dará en el caso de procesos de baja prioridad cuando el conjunto completo tenga una utilización mayor del 100 por ciento). Cuando sean mayores que el periodo del proceso, T, se puede suponer que el proceso no cumplirá su tiempo límite. El análisis anterior conduce al siguiente algoritmo para calcular los tiempos de respuesta:

```
for i in 1..N loop -- para cada proceso por turnos
  n :=0
  w_i^n := C_i
  loop
     calcula nuevo w_i^(n+1) según Ecuación (13.5)
     if w_i^(n+1) = w_i^n then
        R_i := w_i^n
        exit {valor encontrado}
     end if
     if w_i^(n+1) > T_i then
        exit {valor no encontrado}
     end if
     n :=n + 1
  end loop
end loop
```

Por todo ello, si se encuentra un tiempo de respuesta, será menor que T_i, y por lo tanto menor que D_i, su tiempo límite. Recuerde que en el modelo de proceso simple $D_i = T_i$.

En la discusión anterior, se ha utilizado w_i como una mera entidad matemática útil para resolver la ecuación de punto fijo. Sin embargo, es posible concretar w_i de forma intuitiva en el dominio del problema. Considere el punto de activación del proceso i. A partir de ese punto, y hasta que acabe el proceso, el procesador estará ejecutando procesos de prioridad P_i o mayor. Se dice que el procesador está ejecutando un **periodo ocupado por** P_i. Considere w_i como la ventana de tiempo que se mueve a lo largo del periodo de tiempo ocupado. En el instante 0 (el instante en el que se activa el proceso i), se supone que también se han activado todos los procesos de prioridad mayor, y por lo tanto:

$$w_i^1 = C_i + \sum_{j \in hp(i)} C_j$$

Esto será el final del periodo ocupado, a menos que algún otro periodo de mayor prioridad sea activado una segunda vez. Si esto es así, entonces habrá que desplazar la ventana más allá. Si esta expansión continua indefinidamente, el periodo ocupado es ilimitado (esto es, no hay solución). Sin embargo, si en un punto cualquiera una ventana en expansión no recibe un «empujón» extra por parte de un proceso de mayor prioridad, entonces el periodo ocupado habrá concluido, y el tamaño del periodo ocupado será el tiempo de respuesta del proceso.

Para ilustrar cómo se emplea el análisis del tiempo de respuesta, considere el conjunto de procesos D de la Tabla 13.8.

Tabla 13.8. Conjunto de procesos D.

Proceso	Periodo, T	Tiempo ejecución, C	Prioridad, P
a	7	3	3
b	12	3	2
c	20	5	1

El proceso de más alta prioridad, a, tendrá un tiempo de respuesta igual a su tiempo de ejecución (por ejemplo, $R_a = 3$). Para el siguiente proceso habrá que calcular su tiempo de respuesta. Sea w_b^0 el tiempo de ejecución del proceso a, que es 3. Mediante la Ecuación (13.5) se obtiene el siguiente valor de w:

$$w_b^1 = 3 + \left\lceil \frac{3}{7} \right\rceil 3$$

esto es, $w_b^1 = 6$. Dicho valor equilibra la Ecuación ($w_b^2 = w_b^1 = 6$), y permite encontrar el tiempo de respuesta del proceso b (que es $R_b = 6$).

El último proceso conducirá a los siguientes cálculos:

$$w_c^0 = 5$$

$$w_c^1 = 5 + \left\lceil \frac{5}{7} \right\rceil 3 + \left\lceil \frac{5}{12} \right\rceil 3 = 11$$

$$w_c^2 = 5 + \left\lceil \frac{11}{7} \right\rceil 3 + \left\lceil \frac{11}{12} \right\rceil 3 = 14$$

$$w_c^3 = 5 + \left\lceil \frac{14}{7} \right\rceil 3 + \left\lceil \frac{14}{12} \right\rceil 3 = 17$$

$$w_c^4 = 5 + \left\lceil \frac{17}{7} \right\rceil 3 + \left\lceil \frac{17}{12} \right\rceil 3 = 20$$

$$w_c^5 = 5 + \left\lceil \frac{20}{7} \right\rceil 3 + \left\lceil \frac{20}{12} \right\rceil 3 = 20$$

Por lo tanto, R_c tiene un tiempo de respuesta de 20 en el peor caso, lo que significa que cumplirá justo con su tiempo límite. En la Figura 13.5 se ilustra este comportamiento mediante un diagrama de Gantt.

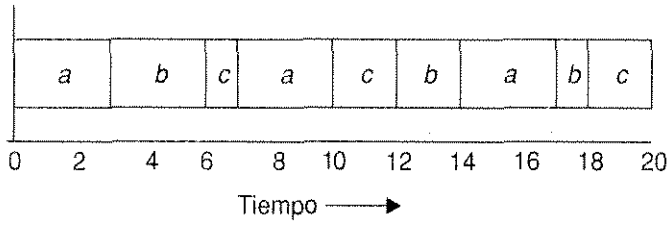


Figura 13.5. Diagrama de Gantt para el conjunto de procesos *D*.

Considere otra vez el conjunto de procesos *C*. Este conjunto fracasa en el test basado en la utilización, aunque se ha visto cómo cumple con todos los tiempos límite hasta el instante 80. La Tabla 13.9 muestra los tiempos de respuesta para este conjunto, calculados mediante el método anterior. Observe que se predice que todos los procesos acabarán antes de completar sus tiempos límite.

El cálculo del tiempo de respuesta tiene la ventaja de que es suficiente y necesario. Si el conjunto de procesos pasa el test, los procesos cumplirán sus tiempos límite. Si falla en el test, entonces, en tiempo de ejecución, algún proceso incumplirá su tiempo límite (a menos que la estimación del tiempo de ejecución, *C*, haya sido excesivamente pesimista). Como estos test son superiores en calidad a los basados en la utilización, este capítulo se concentrará en extender la aplicación del método del tiempo de respuesta.

Tabla 13.9. Tiempos de respuesta para el conjunto de procesos *C*.

Proceso	Periodo, *T*	Tiempo ejecución, *C*	Prioridad, *P*	Tiempo respuesta, *R*
a	80	40	1	80
b	40	10	2	15
c	20	5	3	5

13.6 Análisis del tiempo de respuesta para EDF

Una de las desventajas del esquema EDF es que no proporciona el tiempo de respuesta en el peor caso para cada proceso, cuando se activan todos los procesos en un instante crítico. En esta situación sólo los procesos con un tiempo límite relativo más pequeño interferirán. Pero después puede haber una situación en la que todos (o al menos la mayoría) de los procesos tengan un tiempo límite absoluto más corto. Considere, por ejemplo, un sistema de tres procesos como el mostrado en la Tabla 13.10.

Tabla 13.10. Un conjunto de procesos para EDF.

Proceso	$T(=D)$	C
a	4	1
b	12	3
c	16	8

El comportamiento del proceso b ilustra el problema. En el instante 0, un instante crítico (b) sólo interfiere con el proceso a (una vez), y tiene un tiempo de respuesta de 4. Pero en la siguiente activación (en el instante 12) el proceso c aún está activo, y tiene un tiempo límite más pequeño (16 frente a 24), por lo que c tiene precedencia. El tiempo de respuesta para esta segunda ejecución de b es de 8: el doble del valor obtenido en el instante crítico. Posteriores activaciones pueden producir, incluso, valores mayores, aunque el valor está limitado a 12, ya que el sistema es planificable según EDF (con una utilización de 1). Por ello, encontrar el peor caso es mucho más complicado. Es necesario tener en cuenta todas las activaciones de todos los procesos para ver cuál de ellos sufre la máxima interferencia del resto de procesos con tiempos límite más pequeños.

En el modelo simple, con todos los procesos periódicos, el conjunto completo de procesos repetirá su ejecución cada *hiperperiodo* –el mínimo común múltiplo (MCM) de los periodos de los procesos–. Por ejemplo, en un sistema pequeño con sólo cuatro procesos con periodos de 24, 50, 73 y 101 unidades de tiempo, el MCM es 4.423.800. Para encontrar el peor caso de tiempo de respuesta en EDF habrá que considerar cada activación dentro de las 4.423.800 unidades de tiempo: recuerde que con FPS sólo se necesitaba analizar la primera ejecución (esto es, el máximo tiempo a considerar es 101 unidades).

Aunque haya que considerar más activaciones, es posible obtener una fórmula para calcular cada tiempo de respuesta de forma similar a la dada para FPS. No mostraremos tal formula aquí, pero los lectores interesados en ella pueden encontrarla (además de otros resultados relacionados con la planificación EDF) en el libro sobre EDF reseñado al final del capítulo.

13.7 Tiempo de ejecución en el peor caso

En todos los enfoques de planificación descritos hasta ahora (ejecutivo cíclico, FPS y EDF), se ha supuesto que se conoce el peor caso del tiempo de ejecución de cada proceso, es decir, el máximo tiempo que puede requerir la invocación de cualquier proceso.

La estimación del tiempo de ejecución en el peor caso (representada por el símbolo *C*), puede obtenerse o por análisis o por medición. El problema de la medición es la dificultad para asegurar que realmente se ha observado el peor caso. El inconveniente del análisis es que se debe contar con un modelo efectivo del procesador (incluyendo cachés, encadenado de instrucciones, estados de espera de la memoria, y otros).

La mayoría de las técnicas de análisis implican dos actividades distintas. La primera toma el proceso y descompone su código en un grafo dirigido de bloques básicos. Estos bloques básicos representan el código básico. El segundo componente del análisis toma el código máquina correspondiente a cada bloque básico, y utiliza el modelo de procesador para estimar su peor caso de tiempo de ejecución.

Una vez conocido el tiempo para todos los bloques básicos, se colapsa el grafo dirigido. Por ejemplo, una simple construcción de elección entre dos bloques básicos se colapsará en un único valor (que será el mayor de los dos valores de los bloques básicos). Los bucles se colapsan utilizando el número máximo de iteraciones.

Se pueden utilizar técnicas de reducción de grafos más sofisticadas si se dispone de información semántica suficiente. Para dar sólo un ejemplo de esto, considere el siguiente código:

```
for I in 1.. 10 loop
  if Cond then
    -- bloque básico con coste 100
  else
    -- bloque básico con coste 10
  end if;
end loop;
```

Sin más información, el «coste» total de esta construcción será de 10×100 + el costo del bucle en sí mismo, dando un total de, digamos, 1.005 unidades de tiempo. Sin embargo, se podría deducir (a través del análisis estático del código) que la condición `Cond` sólo puede ser verdadera como mucho en tres ocasiones, por lo que se podrá obtener un valor del coste menos pesimista: 374 unidades de tiempo.

Otras relaciones dentro del código pueden reducir el número de caminos posibles eliminando aquéllos que no se pueden dar; por ejemplo, cuando la bifurcación de «if» en una sentencia condicional excluye una bifurcación «else» posterior. Las técnicas que emprenden este tipo de análisis semánticos suelen requerir anotaciones sobre el código. El proceso de reducción del grafo puede hacer uso de herramientas tales como la programación lineal entera (ILP; integer linear

programming) para obtener una estimación ajustada del tiempo de ejecución en el peor caso. Las anotaciones también pueden advertir sobre los datos de entrada necesarios para encauzar el programa en el camino que conduce a tal estimación.

Evidentemete, si hay que analizar un proceso para buscar su tiempo de ejecución en el peor caso, el código mismo requiere algunas restricciones. Por ejemplo, todos los bucles y recursiones deben estar limitados; de otro modo, sería imposible predecir cuándo va a terminar el código. Más aún, también debería ser analizable el código generado por el compilador.

El principal desafío que afronta el análisis del tiempo de ejecución en el peor caso proviene del uso de los modernos procesadores con cachés en el chip, encadenamiento de instrucciones (pipelining), predicciones de bifurcaciones, etc. Todas estas características intentan reducir el tiempo de ejecución medio, pero influyen haciendo difícil de predecir el peor caso. Si se ignoran estas características, la estimación resultante puede ser demasiado pesimista, pero no siempre resulta fácil incluirlas. Un posible enfoque es suponer una ejecución no apropiativa; de esta forma se podrán tener en cuenta todos los beneficios de la caché. En una fase posterior del análisis, se puede calcular el número de apropiaciones reales, y aplicar una penalización debida a los fallos de la caché y a los encadenamientos de instrucciones.

Modelar en detalle el comportamiento temporal de un procesador moderno no resulta trivial, y se necesita información privilegiada que puede ser difícil de obtener. En el caso de los sistemas de tiempo real queda la opción de elegir entre utilizar arquitecturas de procesadores más simples (pero menos potentes), o poner más empeño en la medición. Dado que todos los sistemas de tiempo real de alta integridad serán objeto de muchas pruebas, parece apropiado un enfoque que combine las pruebas y las medidas en las unidades de código (bloques básicos), con el análisis de caminos para los componentes completos, teniendo en cuenta la tecnología disponible hoy en día.

13.8 Procesos esporádicos y aperiódicos

Para expandir el modelo simple de la Sección 13.1, e incluir los requisitos de los procesos esporádicos (y aperiódicos), el valor T se interpreta como el mínimo (o la media) del intervalo entre llegadas (Audsley et al., 1993a). Un proceso esporádico con un valor T de 20 ms garantiza que no será activado más de una vez cada 20 ms. En realidad, pueden llegar a ser más de 20 ms, pero el test del tiempo de respuesta asegurará que se puede sostener la tasa máxima (¡si se pasa el test!).

El otro requisito que demanda la inclusión de procesos esporádicos tiene que ver con la definición de tiempo límite. El modelo simple supone que $D = T$. Para los procesos esporádicos esto no resulta razonable. A menudo, un proceso esporádico se utiliza para encapsular una rutina de gestión de error o para responder a una señal de aviso. El modelo de fallo del sistema puede establecer que la rutina de error sea invocada muy raramente (y que cuando lo sea, significará que se trata de algo urgente, por lo que habrá un tiempo límite pequeño). Nuestro modelo debe distinguir, por lo tanto, entre D y T, y permitir que $D < T$. En efecto, para muchos procesos periódicos, será útil permitir definir valores de tiempos límite menores que el periodo.

Una inspección del algoritmo del tiempo de respuesta para una planificación de prioridades estáticas, descrito en la Sección 13.5, revela que:

- Funciona perfectamente para valores de D menores que T, siempre que el criterio de parada sea $w_i^{n+1} > D_i$.
- Funciona perfectamente bien con cualquier orden de prioridad (*hp(i)* siempre se refiere al conjunto de procesos de mayor prioridad).

Aunque algunas ordenaciones de prioridad son mejores que otras, el test proporcionará tiempos de respuesta en el peor caso para la ordenación dada.

En la Sección 13.9, se definió un orden óptimo para la prioridad cuando $D < T$ (además de demostrarlo). Una sección posterior considerará una algoritmo extendido y una ordenación óptima para el caso general de $D < T$, $D = T$, o $D > T$.

13.8.1 Procesos estrictos y flexibles

Para los procesos esporádicos puede definirse una tasa de llegada media y máxima. Sin embargo, en muchas situaciones el peor caso es considerablemente mayor que la media. Las interrupciones suelen llegar en ráfagas, y un sensor que realiza lecturas anormales puede provocar la realización de una cantidad considerable de cálculos adicionales. Por consiguiente, según se ha observado en sistemas de ejecución reales, la medida de planificabilidad que considera los peores casos puede conducir a utilizaciones muy bajas del procesador. Como guía para encontrar los requisitos mínimos, deberían aplicarse siempre estas dos reglas:

- Regla 1: todos los procesos deberían ser planificables utilizando los tiempos medios y las tasas medias de llegada.
- Regla 2: todos los procesos de tiempo real estrictos deberían ser planificables utilizando los peores casos del tiempo de ejecución y de la tasa de llegada de todos los procesos (incluidos los flexibles).

Una consecuencia de la Regla 1 es que pueden existir situaciones en las cuales no sea posible cumplir todos los tiempos límite. Esta condición se conoce como **sobrecarga transitoria**. La Regla 2, sin embargo, asegura que ningún proceso de tiempo real estricto incumplirá su tiempo límite. Si la Regla 2 conduce a una utilización inaceptablemente baja en «ejecución normal», deberían tomarse medidas para intentar reducir los tiempos de ejecución (o las tasas de llegada) en el peor caso.

13.8.2 Procesos aperiódicos y servidores de prioridad estática

Una forma simple de planificar procesos aperiódicos dentro de un esquema basado en prioridades, es ejecutar tales procesos con una prioridad por debajo de las prioridades asignadas a los procesos estrictos. Aunque éste es un esquema seguro, no soporta adecuadamente los procesos

flexibles, ya que si sólo se ejecutan como actividades en segundo plano, incumplirán frecuentemente sus tiempos límite. Para mejorar la situación de los procesos flexibles, se puede emplear un **servidor**. Los servidores protegen los recursos necesarios para los procesos estrictos, pero también permiten que los procesos flexibles se ejecuten tan pronto como sea posible.

Desde que fueran introducidos en 1987, se han definido varios métodos de servidores. Aquí sólo vamos a considerar dos de ellos: el servidor diferible (DS; Deferrable Server) y el servidor esporádico (SS; Sporadic Server) (Lehoczky et al., 1987).

Con el DS, se realiza un análisis (tomando, por ejemplo, el enfoque del tiempo de respuesta) que hace posible añadir procesos nuevos con la máxima prioridad.[2] Este servidor tiene un periodo, T_s, y una capacidad, C_s. Estos valores se eligen de forma que todos los procesos estrictos del sistema sigan siendo planificables, incluso si el servidor se ejecuta periódicamente con periodo T_s y tiempo de ejecución C_s. En tiempo de ejecución, cuando llega un proceso aperiódico y existe capacidad disponible, comienza a ejecutarse inmediatamente y continúa hasta que finaliza o se agota la capacidad. En este último caso, el proceso aperiódico se suspende (o se transfiere a una prioridad de segundo plano). Con el modelo DS, la capacidad se repone cada T_s unidades de tiempo.

El funcionamiento del SS difiere del funcionamiento del DS en la política de reposición. Con el SS, si llega un proceso en el instante de tiempo t y utiliza una capacidad c, entonces el servidor tendrá repuesta esta capacidad c tras T_s unidades de tiempo después de t. En general, el SS puede proporcionar una capacidad mayor que el DS, pero a costa de sobrecargar la implementación. La Sección 13.14.2 describe el modo en que se soporta el SS en POSIX. El DS y el SS pueden analizarse mediante los análisis de tiempo de respuesta (Bernat y Burns, 1999).

Como todos los servidores limitan la capacidad disponible para los procesos flexibles aperiódicos, también se pueden utilizar para asegurar que los procesos esporádicos no se ejecuten más a menudo de lo esperado. Si un proceso esporádico con un intervalo entre llegadas de T_i y un tiempo de ejecución en el peor caso de C_i no es implementado directamente como un proceso, sino a través de un servidor con $T_s = T_i$ y $C_s = C_i$, entonces su impacto (interferencia) sobre los procesos de baja prioridad será limitado, incluso si el proceso esporádico llega demasiado rápidamente (lo cual podría ser una condición de error).

Se dice que todos los servidores (el DS, el SS y los demás) *preservan el ancho de banda*, ya que intentan:

- Que los recursos de la CPU estén disponibles inmediatamente para los procesos aperiódicos (si existe capacidad).
- Retener la capacidad tanto como les sea posible ante la no existencia de procesos aperiódicos (permitiendo que se ejecuten los procesos estrictos).

Otro esquema que preserva el ancho de banda, y que a menudo se comporta mejor que las técnicas de servidor, es la **planificación de prioridad dual** (Davis y Wellings, 1995). En este caso, el

[2] Son posibles servidores con otras prioridades, pero la descripción resulta más sencilla si el servidor recibe una prioridad mayor que todos los procesos estrictos.

rango de prioridades se divide en tres bandas: alta, media y baja. Todos los procesos aperiódicos se ejecutan en la banda media. Los procesos estrictos, cuando se activan, lo hacen en la banda baja, pero se promueven a la banda superior a tiempo para cumplir sus tiempos límite. Por eso, en la primera etapa de ejecución darán paso a las actividades aperiódicas (pero se ejecutarán si no existen tales actividades). En la segunda fase se promoverán a la prioridad más alta, y entonces tendrán prioridad sobre cualquier otro proceso aperiódico. En la banda alta, las prioridades se asignan de acuerdo con el enfoque de tiempo límite monotónico (véase a continuación). La promoción a esta banda se da en el instante $D - R$. Para implementar el esquema de prioridad dual se necesita contar con prioridades dinámicas.

13.8.3 Procesos aperiódicos y servidores EDF

Siguiendo el desarrollo de la tecnología de servidor para sistemas de prioridad estática, la mayoría de los enfoques comunes se han reinterpretado dentro del contexto de los sistemas dinámicos EDF. Por ejemplo, existe un servidor esporádico dinámico (Spuri y Buttazzo, 1994). Mientras que un sistema estático necesita una asignación de prioridades (anterior a la ejecución), la versión dinámica precisa calcular cada tiempo límite cada vez que tiene que ejecutarse. En esencia, el algoritmo de tiempo de ejecución asigna al servidor el tiempo límite actual más pequeño si (y sólo si) existe un proceso aperiódico pendiente de servir y existe capacidad suficiente. Una vez que la capacidad se ha agotado, el servidor será suspendido hasta que ésta sea repuesta.

Un esquema diferente, que tiene varias similitudes con el esquema de prioridad dual de FPS, es el servidor «el tiempo límite más temprano el último» (EDL; earliest deadline last), definido por Chetto y Chetto (1989). En este caso, los procesos estrictos se alternan entre la ejecución con EDF (si no hay procesos aperiódicos) y con EDL (si existen tareas aperiódicas que realizar). El esquema EDL asegura que todos los procesos estrictos cumplirán sus tiempos límite, pero pospone la activación de los procesos tanto como sea posible. Por eso, la capacidad permanece disponible para los procesos aperiódicos de forma inmediata (si existe capacidad disponible). Un proceso estricto podrá desalojar a los procesos flexibles cuando sea promovido y completar su ejecución justo en su tiempo límite.

13.9 Sistemas de procesos con $D < T$

En la discusión anterior sobre los procesos aperiódicos se argumentó que, en general, debe ser posible definir un tiempo límite para un proceso que sea menor que su intervalo entre llegadas (o periodo). También se consideró que, para $D = T$, la ordenación de prioridad de tasa monotónica resultaba óptima para un esquema de prioridad estática. Leung y Whitehead (1982) demostraron que para $D < T$ se podía definir una formulación similar: la ordenación de prioridades monotónica de tiempo límite (DMPO; deadline monotonic priority ordering). Por eso, la prioridad fija de un proceso es inversamente proporcional a su tiempo límite: $(D_i < D_j \Rightarrow P_i > P_j)$. La Tabla 13.11 muestra la asignación de prioridades apropiada para un conjunto simple de procesos. También incluye el tiempo de respuesta en el peor caso (según se calculó por el algoritmo en la Sec-

Tabla 13.11. Conjunto de procesos ejemplo para DMPO.

Proceso	Periódo, T	Tiempo límite, D	Tiempo ejecución, C	Prioridad, P	Tiempo respuesta, R
a	20	5	3	4	3
b	15	7	3	3	6
c	10	10	4	2	10
d	20	20	3	1	20

ción 13.5). Hay que destacar que una ordenación de prioridades de tasa monotónica podría no planificar estos procesos.

En la subsección siguiente, se demuestra la optimalidad de DMPS. Con este resultado y la aplicabilidad directa del análisis del tiempo de respuesta a este modelo de procesos, resulta evidente que la planificación de prioridades estáticas puede tratar de forma adecuada este conjunto más general de requisitos de planificación. Lo mismo no es cierto para la planificación EDF. Una vez que los procesos pueden tener $D < T$, no se puede aplicar el test simple de utilización (utilización total menor que uno). Además, el análisis del tiempo de respuesta, tratado en la Sección 13.6, resulta considerablemente más complejo para EDF que para FPS.

Después de plantear esta dificultad con EDF, debe recordarse que EDF es el esquema de planificación más efectivo. Por ello, cualquier conjunto de procesos que pase el test de planificabilidad en FPS *siempre* cumplirá con sus requisitos de temporización si se ejecuta en EDF. Los test de necesidad y suficiencia para FPS pueden verse, por lo tanto, como un test de suficiencia para EDF.

13.9.1 Demostración de que DMPO es óptima

La ordenación monotónica de prioridades de tiempo límite (DMPO) es óptima si cualquier conjunto de procesos, Q, planificable por el esquema de prioridades, W, también lo es mediante DMPO. La prueba de optimalidad de DMPO implica transformar las prioridades de Q (según son asignadas por W) hasta obtener una ordenación DMPO. Cada paso de la transformación preservará la planificabilidad.

Sean i y j dos procesos (con prioridades adyacentes) de Q tales que bajo W: $P_i > P_j$ y $D_i > D_j$. Sea el esquema W' idéntico a W excepto en que los procesos i y j han sido intercambiados. Considérese la planificabilidad de Q bajo W':

- Todos los procesos con prioridades mayores que P_i no serán afectados por este cambio en procesos de menor prioridad.
- Todos los procesos con prioridades menores que P_i no serán afectados. Recibirán la misma interferencia por parte de i y de j.

- El proceso j, que era planificable bajo W, ahora tiene una prioridad mayor, por lo que sufrirá menos interferencias, y por lo tanto también deberá ser planificable bajo W'.

Todo lo que queda por hacer es demostrar que el proceso i, cuya prioridad ha disminuido, sigue siendo planificable.

Bajo W, $R_j \leq D_j$, $D_j < D_i$, y $D_i \leq T_i$; por lo tanto, el proceso i sólo interfiere una vez durante la ejecución de j.

Una vez que los procesos han sido intercambiados, el nuevo tiempo de respuesta de i es el antiguo tiempo de respuesta de j. Esto es cierto porque bajo ambas ordenaciones de prioridad el tiempo de ejecución $C_j + C_i$ ha sido realizado con el mismo nivel de interferencia que los procesos de prioridad mayor. El proceso j ha sido relegado sólo una vez durante R_j, y por lo tanto interferirá sólo una vez durante la ejecución de i bajo W'. De esto se sigue que:

$$R_i' = R_j \leq D_j < D_i$$

Se puede concluir que el proceso i es planificable tras el intercambio.

El esquema de prioridades W' se puede transformar ahora (en W'') eligiendo otros dos procesos «que estén desordenados según DMPO», e intercambiándolos. Cada uno de estos intercambios conservan la planificabilidad. Finalmente no existirán más procesos que intercambiar; el orden será exactamente el exigido por DMPO, y el conjunto de procesos aún será planificable. Por todo ello, DMPO es óptimo.

Hay que destacar que en el caso especial de $D = T$, la demostración anterior también se puede utilizar para demostrar que, en estas circunstancias, la ordenación de tasa monotónica también es óptima.

13.10 Interacciones y bloqueos entre procesos

Una de las premisas simplificadoras incorporadas al modelo del sistema, descrito en la Sección 13.1, es la necesidad de que los procesos sean independientes. Esto es realista, ya que en casi todas las aplicaciones importantes los procesos deberán interactuar. En los Capítulos 8 y 9, se dijo que los procesos pueden interactuar de forma segura mediante algún mecanismo que proteja los datos compartidos (utilizando, por ejemplo, semáforos, monitores u objetos protegidos), o de forma directa (utilizando algún mecanismo de cita). Todas estas características de los lenguajes hacen que un proceso pueda ser suspendido hasta que ocurra cierto evento futuro (por ejemplo, esperar a obtener el control de un semáforo, o entrar en un monitor, o que algún otro proceso se encuentre en la posición de aceptar una cita). En general, la comunicación síncrona conduce a un análisis más pesimista, además de que resulta más difícil definir el peor caso real cuando existen varias dependencias entre los procesos en ejecución. El siguiente análisis es por lo tanto más preciso cuando se trata de la comunicación asíncrona entre procesos que intercambian datos mediante recursos compartidos protegidos. La mayoría del material presentado en las dos secciones siguientes concierne a la planificación de prioridad estática. Al final de la discusión, se tratará de cómo aplicar los resultados a la planificación EDF.

Si un proceso es suspendido a la espera de que otro proceso de menor prioridad realice cierto cálculo necesario, se está socavando, en algún sentido, el modelo de prioridad. En un mundo ideal, tal **inversión de la prioridad** (Lauer y Satterwaite, 1979) (esto es, que cierto proceso espere por un proceso de menor prioridad) no debe darse. En general, sin embargo, no es posible eliminar totalmente este efecto, aunque se pueden minimizar sus efectos adversos. Si un proceso está esperando por otro proceso de menor prioridad, se dice que está **bloqueado**. Para poder testar la planificabilidad, el bloqueo debe estar limitado y ser ponderable (además de ser pequeño).

Para ilustrar un ejemplo extremo de la inversión de prioridad, considere la ejecución de cuatro procesos periódicos: *a*, *b*, *c* y *d*. Suponga que las prioridades se han asignado según el esquema monotónico de tiempo límite, de forma que la prioridad del proceso *d* es la mayor y la del *a* es la menor. Además, suponga que los procesos *d* y *a* (y los procesos *d* y *c*) comparten una sección crítica (recurso), designada por *Q* (y *V*), protegida mediante exclusión mutua. La Tabla 13.12 muestra los detalles de los cuatro procesos y sus secuencias de ejecución. En esta tabla, *E* representa un momento de ejecución simple (pulso), y *Q* (o *V*) representan un pulso de ejecución con acceso a la sección crítica *Q* (o *V*). Así, el proceso *c* se ejecuta durante cuatro pulsos, de los cuales los dos de la mitad son para acceder a la sección crítica *V*.

Tabla 13.12. Secuencias de ejecución.

Proceso	Prioridad	Secuencia ejecución	Instante activación
a	1	EQQQQE	0
b	2	EE	2
c	3	EVVE	2
d	4	EEQVE	4

La Figura 13.6 muestra la secuencia de ejecución para los instantes de inicio indicados en la tabla. El proceso *a* se activa el primero, se ejecuta, y bloquea la sección crítica, *Q*. Entonces resulta desalojado por la activación del proceso *c*, que se ejecuta durante un único pulso, bloquea *V*, y resulta desalojado por la activación del proceso *d*.

Después, se ejecuta el proceso de mayor prioridad hasta que también desea bloquear la sección crítica, *Q*; entonces debe ser suspendido (ya que *a* obtuvo un bloqueo sobre la sección crítica). En este punto, *c* vuelve a conseguir el procesador y continúa. Una vez ha terminado, *b* comenzará a ejecutarse por derecho. Solamente cuando *b* haya acabado, volverá a ser posible ejecutar *a*; entonces, completará su uso de *Q* y permitirá que *d* continúe y acabe. Con este comportamiento, *d* finaliza en el instante 16, y por lo tanto su tiempo de respuesta es 12; el de *c* es 6, 8 el de *b*, y 17 el de *a*.

Una vistazo a la Figura 13.6 delata que el proceso *d* sufre una considerable inversión de prioridad. No solamente es bloqueado por el proceso *a*, sino también por los procesos *b* y *c*. Algún nivel de bloqueo resulta inevitable; si hay que mantener la integridad de la sección crítica (y por lo tanto de los datos compartidos), entonces *a* debe ejecutarse con preferencia sobre *d* (mientras tenga el

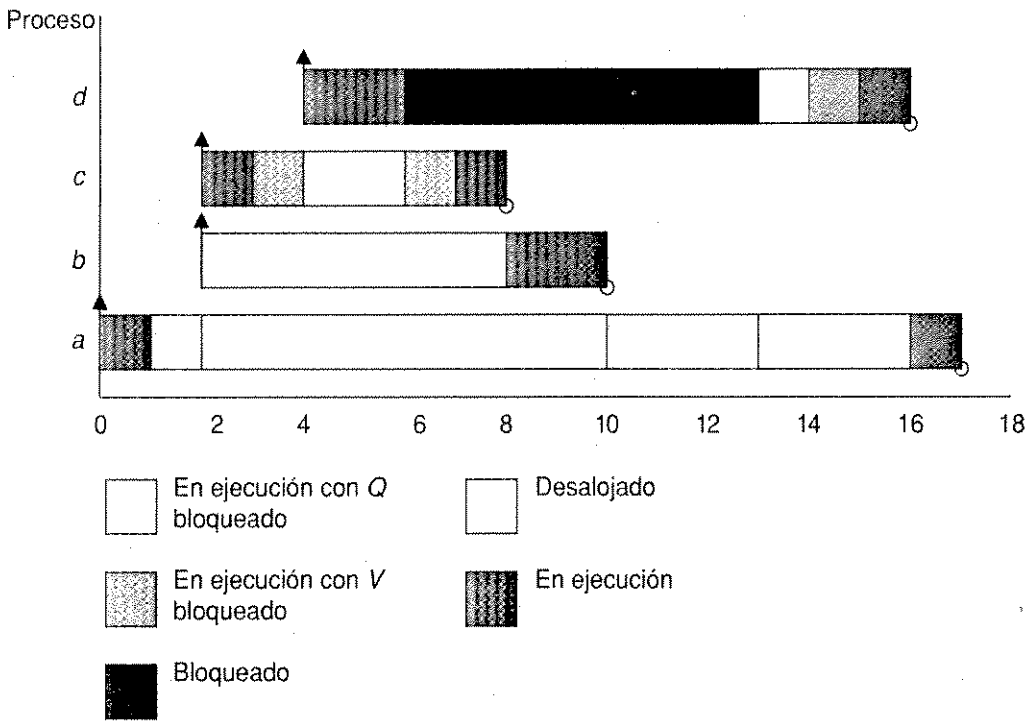


Figura 13.6. Ejemplo de inversión de prioridad.

bloqueo). Pero el bloqueo de *d* por los procesos *c* y *b* resulta improductivo, y afectará severamente a la planificabilidad del sistema (ya que el bloqueo sobre el proceso *d* es excesivo).

Este tipo de inversión de prioridad es el resultado de un esquema puro de prioridades estáticas. Una forma de limitar este efecto es utilizar la **herencia de prioridad** (Cornhill et al., 1987). Con la herencia de prioridad, la prioridad de un proceso ya no es estática: si un proceso *p* es suspendido esperando a que el proceso *q* realice alguna operación, entonces la prioridad de *q* se hace igual a la prioridad de *p* (en el caso de que fuera menor). En ejemplo anterior, el proceso *a* tomará la prioridad del proceso *d*, y por lo tanto se ejecutará preferentemente sobre los procesos *c* y *b*. Esto se muestra en la Figura 13.7. Hay que destacar que, como consecuencia de este algoritmo, el proceso *b* sufrirá un bloqueo, incluso aunque no utilice objeto compartido alguno. También hay que destacar que el proceso *d* tiene ahora un segundo bloqueo, pero su tiempo de respuesta se ha reducido a 9.

Con esta simple regla de herencia, la prioridad de un proceso será el máximo entre su prioridad por defecto y las prioridades de los otros procesos que en cada instante dependan de él.

En general, la herencia de prioridad podría no estar restringida a un único paso. Si el proceso *d* está esperando por el proceso *c*, pero *c* no puede responder a *d* porque a su vez está esperando por el proceso *b*, entonces tanto *b* como *c* podrían recibir la prioridad de *d*. La implicación en tiempo de ejecución para el distribuidor es que la prioridad de los procesos cambiará a menudo, y que en lugar de intentar gestionar una cola ordenada por prioridad, podría ser mejor elegir el proceso correcto para ejecutar (o para hacerle ejecutable) en el instante en el que se necesita la acción.

En el diseño de un lenguaje de tiempo real, la herencia de prioridad podría parecer de suma importancia. Cuestiones de efectividad sobre el modelo implican que el modelo de concurrencia

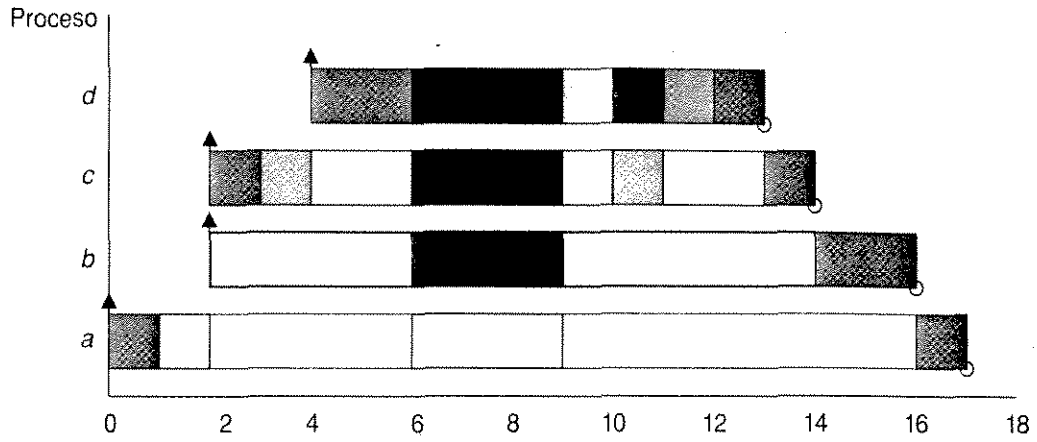


Figura 13.7. Ejemplo de herencia de prioridad.

debe adquirir cierta forma particular. Con semáforos estándares y variables de condición, no existe un relación directa entre el acto de ser suspendido y la identidad del proceso en que revierte esa situación. Por lo tanto, la herencia no será fácilmente implementable. Con el paso de mensajes síncrono, las referencias indirectas (por ejemplo, la utilización del canal en occam2) también pueden dificultar la identificación del proceso al que se está esperando. Para maximizar la efectividad de la herencia, lo más apropiado será utilizar nombrado directo y simétrico.

Sha et al. (1990) muestran que con un protocolo de herencia de prioridades existe un límite para el número de veces que un proceso puede ser bloqueado por procesos de menor prioridad. Si un proceso tiene m secciones críticas que pueden provocar su bloqueo, entonces el máximo número de veces que puede ser bloqueado será m. Esto es, en el peor caso, cada sección crítica será bloqueada por un proceso de menor prioridad (como sucede en la Figura 13.7). Si sólo hay n ($n < m$) procesos de menor prioridad, podrá reducirse considerablemente este máximo (a n).

Si B_i es el máximo tiempo de bloqueo que puede sufrir el proceso i, entonces se puede encontrar fácilmente una fórmula que calcule B para este modelo simple de herencia de prioridades. Sea K el número de secciones críticas (recursos). La Ecuación (13.6) proporciona un límite superior para B.

$$B_i = \sum_{k=1}^{K} \textit{utilización}(k, i)C(k) \qquad \textbf{(13.6)}$$

donde *utilización* es una función 0/1: *utilización*$(k, i) = 1$ si el recurso k se utiliza al menos por un proceso con una prioridad menor que P_i, y al menos por un proceso con prioridad mayor o igual a P_i. En otro caso será 0. $C(k)$ es el tiempo de ejecución de la sección crítica k en el peor caso.

Este algoritmo no es óptimo para este protocolo de herencia, pero sirve para ilustrar los factores que hay que tener en cuenta cuando se calcule B. En la Sección 13.11 se describirán mejores protocolos y se dará una fórmula mejorada para B.

13.10.1 Bloqueo y cálculo del tiempo de respuesta

Dado un valor obtenido para B, el algoritmo del tiempo de respuesta se puede modificar para tener en cuenta el factor bloqueo:[3]

$$R = C + B + I$$

es decir:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{13.7}$$

que, de nuevo, se puede resolver construyendo una relación de recurrencia:

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \tag{13.8}$$

Hay que destacar que esta formulación puede resultar pesimista (no necesariamente «suficiente y necesaria»). Que un proceso sufra realmente su máximo bloqueo dependerá del escalonamiento temporal de los procesos. Por ejemplo, si todos los procesos son periódicos y todos tienen el mismo periodo, entonces no serán desalojados, y por lo tanto no habrá inversión de la prioridad. Sin embargo, y en general, la Ecuación (13.7) representa un test de planificabilidad eficaz para sistemas de tiempo real con procesos cooperantes.

13.11 Protocolos de acotación de la prioridad

Aunque el protocolo estándar de herencia da un límite superior para el número de bloqueos con los que se puede encontrar un proceso de prioridad alta, este límite puede todavía conducir a un cálculo del peor caso inaceptablemente pesimista. Esto se debe a la posibilidad de desarrollar cadenas de bloqueos (bloqueos transitivos), es decir, situaciones en las que el proceso c es bloqueado por el proceso b, el cual está bloqueado por el proceso a, y así sucesivamente. Como los datos compartidos son un recurso del sistema, desde el punto de vista del gestor de recursos no sólo se debe minimizar el bloqueo, sino que las condiciones de fallo (como los interbloqueos) deben ser eliminadas. Los protocolos de acotación de la prioridad (Sha et al., 1990) abordan todas estas cuestiones. En este capítulo se considerarán dos de ellos: el **protocolo original de acotación de la prioridad** y el **protocolo inmediato de acotación de la prioridad**. Primero se describirá el protocolo original (OCPP; original ceiling priority protocol), para después pasar a estudiar su variante inmediata (ICPP; immediate ceiling priority inheritance), más sencilla. Cuando se utiliza cualquiera de estos protocolos en un sistema monoprocesador:

[3] El bloqueo también se puede incorporar en test basados en la utilización, pero entonces se debe considerar a cada proceso individualmente.

- Un proceso de alta prioridad puede ser bloqueado por procesos de prioridad baja en una sola ocasión como máximo durante su ejecución.
- Se previenen los bloqueos mutuos (interbloqueos).
- Se previenen los bloqueos transitivos.
- Se aseguran (por el protocolo mismo) los accesos mutuamente excluyentes a los recursos.

La mejor manera de describir los protocolos de acotación de la prioridad es en relación con los recursos protegidos por secciones críticas. En esencia, el protocolo asegura que si un recurso está bloqueado por cierto proceso *a*, y esto conduce a que se bloquee un proceso de mayor prioridad, *b*, entonces no se permite que ningún otro recurso que pueda bloquear a *b* sea bloqueado más que por *a*. Un proceso, por lo tanto, puede ser retardado no sólo mientras está intentando bloquear un recurso previamente bloqueado, sino también cuando ese bloqueo pudiera producir un bloqueo múltiple de procesos de mayor prioridad.

El protocolo original toma la siguiente forma:

(1) Cada proceso tiene asignada una prioridad estática por defecto (quizás según el esquema monotónico de tiempo límite).

(2) Cada recurso tiene definido un valor cota estático, que es la prioridad máxima de los procesos que lo están utilizando.

(3) Un proceso tiene una prioridad dinámica que es el máximo de su propia prioridad estática y de cualquiera de las que herede debido a que bloquea procesos de mayor prioridad.

(4) Un proceso sólo puede bloquear un recurso si su prioridad dinámica es mayor que la cota máxima de cualquier recurso actualmente bloqueado (excluyendo cualquiera que él mismo ya pudiera tener bloqueado).

Se permite el bloqueo del primer recurso del sistema. El efecto del protocolo es asegurar que el segundo recurso sólo pueda ser bloqueado si no existe un proceso de mayor prioridad que utilice ambos recursos. Consecuentemente, la cantidad máxima de tiempo que un proceso puede ser bloqueado es igual al tiempo de ejecución de la sección crítica más larga en cualquiera de los procesos de menor prioridad que son accedidos por procesos de alta prioridad; esto es, la Ecuación (13.6) se convierte en:

$$B_i = \max_{k=1}^{K} \textit{utilización}(k, i)C(k) \qquad \textbf{(13.9)}$$

La ventaja de los protocolos de acotación de la prioridad es que los procesos de alta prioridad sólo pueden ser bloqueados por procesos de prioridad baja en una ocasión (por cada activación). El coste de esto es que habrá más procesos que experimenten bloqueos.

No se pueden ilustrar todas las características del algoritmo con un único ejemplo, pero la secuencia de ejecución mostrada en la Figura 13.8 nos da una buena idea sobre el modo en que funciona el algoritmo, y proporciona una comparación con los enfoques anteriores (esta figura ilustra la misma secuencia de procesos utilizada en las Figuras 13.7 y 13.6).

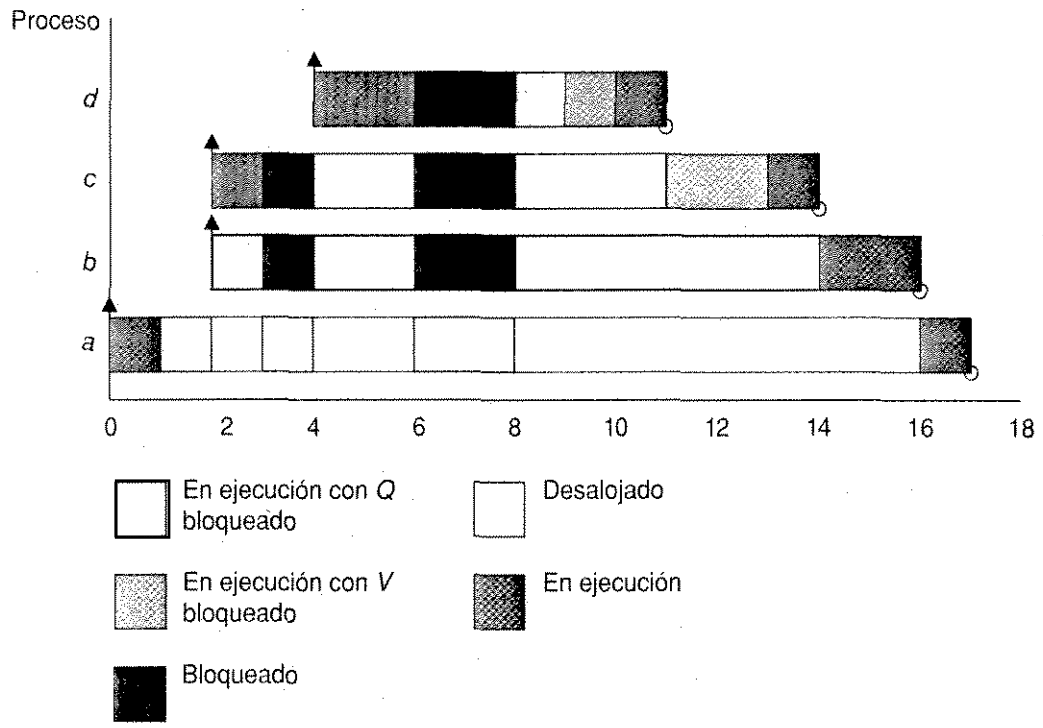


Figura 13.8. Ejemplo de la herencia de prioridad (OCPP).

En la Figura 13.8, el proceso *a* obtiene el bloqueo de su primera sección crítica, y ningún otro recurso ha sido bloqueado. De nuevo resulta expropiado por el proceso *c*, pero ahora el intento de *c* de bloquear su segunda sección crítica (*V*) no resulta exitoso, ya que su prioridad (3) no es mayor que la cota actual (que es 4, ya que *Q* está bloqueado y en uso por el proceso *d*). En el instante 3, *a* está bloqueando a *c*, y por lo tanto se ejecuta con su prioridad en el nivel 3, bloqueando por ello a *b*. El proceso de mayor prioridad, *d*, expropia a *a* en el instante 4, pero resulta bloqueado a continuación cuando intenta acceder a *Q*. Por eso *a* continuará (con prioridad 4) hasta que libere su bloqueo sobre *Q* y su prioridad caiga hasta 1. Ahora, *d* puede continuar hasta completar su tarea (con un tiempo de respuesta de 7).

Los protocolos de acotación de la prioridad aseguran que un proceso sólo será bloqueado en una ocasión durante cada invocación. La Figura 13.8, sin embargo, parece mostrar que el proceso *b* (y el proceso *c*) sufren dos bloqueos. Lo que sucede realmente es que un bloqueo simple está siendo dividido en dos por la expropiación del proceso *d*. La Ecuación (13.9) determina que todos los procesos (excepto el proceso *a*) sufrirán un único bloqueo máximo de 4. La Figura 13.8 muestra que para esta secuencia de ejecución concreta, el proceso *c* y el proceso *d* realmente sufren un bloqueo de 3, y el proceso *d* de sólo 2.

13.11.1 Protocolo inmediato de acotación de la prioridad

El algoritmo inmediato de acotación de la prioridad (ICPP) toma un enfoque más sencillo, y *fija* la prioridad de un proceso tan pronto bloquea un recurso (en lugar de hacerlo cuando realmente bloquea a un proceso de mayor prioridad). El protocolo se define como sigue:

- Cada proceso tiene asignada una prioridad por defecto (quizás mediante el esquema monotónico de tiempo límite).
- Cada recurso tiene definido un valor cota estático, que es la prioridad máxima de los procesos que lo utilizan.
- Un proceso tiene una prioridad dinámica, que es el máximo entre su propia prioridad estática y los valores techo de cualquier recurso que tenga bloqueado.

Como consecuencia de esta última regla, un proceso sólo podrá ser bloqueado al principio de su ejecución. Una vez que el proceso comience a ejecutarse, todos los recursos necesarios deberían estar libres; si no lo estuvieran, entonces algún proceso tendría una prioridad mayor o igual, y la ejecución del proceso deberá ser pospuesta.

El mismo conjunto de procesos anteriormente estudiados se pueden ejecutar ahora bajo ICPP (véase la Figura 13.9).

El proceso *a*, teniendo bloqueado *Q* en el instante 1, se ejecuta durante los siguientes 4 pulsos con prioridad 4. Por lo tanto, ninguno de los procesos *b*, *c* o *d* puede comenzar. Una vez que que *a* desbloquea *Q* (y reduce su prioridad por ello), los demás procesos se ejecutan por orden de prioridad. Hay que destacar que todos los bloqueos se producen antes de la ejecución real, y que el tiempo de respuesta de *d* es de sólo 6. Esto puede llevar a conclusiones erróneas, ya que el tiempo de bloqueo en el peor caso es el mismo para los dos protocolos (véase la Ecuación (13.9)).

Aunque el comportamiento en el peor caso de los dos esquemas de acotación es idéntico (desde el punto de vista de la planificación), existen algunas diferencias:

- ICCP es más sencillo de implementar que el original (OCPP), ya que no hay que monitorizar las relaciones de bloqueo.
- ICPP produce menos cambios de contexto, ya que el bloqueo es previo a la primera ejecución.
- ICPP requiere más cambios de prioridad, ya que éstos se producen con todas las utilizaciones de recursos; OCPP modifica la prioridad sólo si se producen bloqueos.

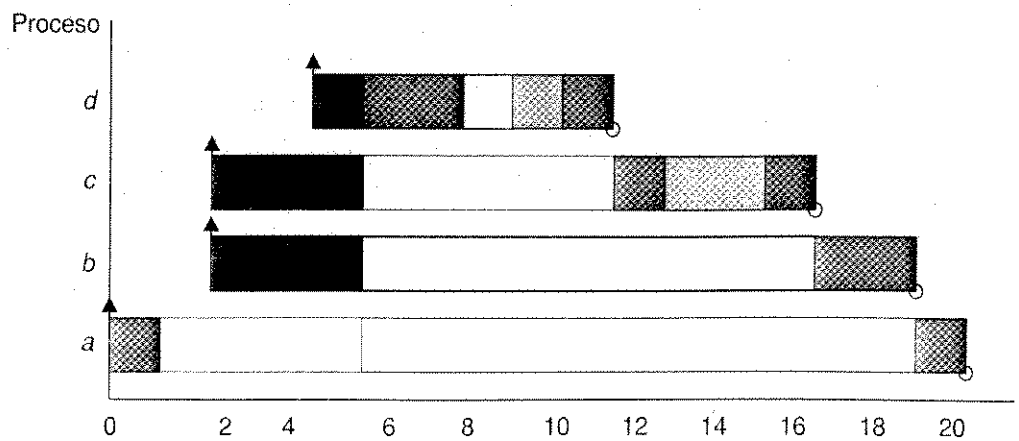


Figura 13.9. Ejemplo de herencia de prioridad (ICPP).

Finalmente, hay que destacar que el Protocolo ICPP se denomina *Priority Protect Protocol* en POSIX y *Priority Ceiling Emulation* en Java para tiempo real.

13.11.2 Protocolos de acotación, exclusión mutua e interbloqueos

Aunque los algoritmos anteriores para los dos protocolos de acotación se definieron en términos de bloqueos de recursos, hay que resaltar que los protocolos proporcionan por sí mismos un acceso mutuamente excluyente mejor que algunas otras primitivas de sincronización aportadas (al menos en sistemas monoprocesador). Considere ICPP: si un proceso tiene acceso a algún recurso, entonces estará ejecutándose con el valor máximo de prioridad. Ningún otro proceso que utilice ese recurso puede tener una prioridad mayor, y por lo tanto el proceso en ejecución podrá ejecutarse sin impedimentos mientras utilice el recurso; si es expropiado, el nuevo proceso no utilizará este recurso en particular. En cualquier caso, se asegura la exclusión mutua.

La otra propiedad principal de los protocolos de acotación de prioridad (otra vez para los sistemas monoprocesador) es que están libres de interbloqueos. En el Capítulo 11, se trató en detalle la utilización de recursos libres de interbloqueos. Los protocolos de acotación son una forma de prevención contra los interbloqueos. Si un proceso retiene un recurso mientras está reclamando otro, entonces la cota del segundo recurso no puede ser menor que la del primero. En efecto, si dos recursos se utilizan en distinto orden (por diferentes procesos), entonces sus cotas deben ser idénticas. Dado que un proceso no desaloja a otro con su misma prioridad, se concluye que una vez que uno de ellos ha accedido al recurso, todos los otros recursos estarán disponibles cuando sea preciso. No hay posibilidad de esperas circulares, y se previenen los interbloqueos.

13.11.3 Bloqueos y EDF

Al considerar recursos compartidos y bloqueos, existe una analogía directa entre EDF y FPS. Mientras que FPS adolece de inversión de prioridad, EDF sufre de inversión del tiempo límite (cuando un proceso necesita un recurso que actualmente está bloqueado por otro proceso con un tiempo límite más largo). No sorprende que se hayan desarrollado protocolos de acotación para EDF, aunque, como en las anteriores comparaciones, el esquema EDF resulta algo más complejo.

Debido a que las prioridades son estáticas, resulta fácil determinar qué proceso puede bloquear al proceso analizado actualmente. Con EDF, esta relación es dinámica: depende de cuáles sean los procesos (con tiempos límite más largos) que estén activos cuando el proceso se active, y esto varía de una activación a otra a lo largo del hiperperiodo.

Quizás, el mejor esquema para EDF es la *política de pila de recursos* (SRP; Stack Resource Policy) de Baker (1991). Ésta funciona de un modo muy similar al protocolo inmediato de acotación de prioridad para FPS (de hecho, SRP influyó en el desarrollo de ICPP). A cada proceso, bajo SRP, se le asigna un nivel de apropiación. Los niveles de apropiación reflejan los tiempos lí-

mite relativos de los procesos, de modo que cuanto más cortos sean, mayor será el nivel de apropiación (como si la prioridad estática de los procesos fuera asignada por el esquema de tiempo límite monotónico). En tiempo de ejecución, los recursos reciben una cota basada en el máximo nivel de apropiación de los procesos que utilizan el recurso. Cuando se activa un proceso sólo puede desalojar al proceso actualmente en ejecución si su tiempo límite absoluto es más corto y su nivel de apropiación es mayor que la cota más alta de los recursos actualmente bloqueados. El resultado de este protocolo es idéntico al ICPP. Los procesos sufren únicamente un bloqueo, que ocurre al ser activados. Así, se previenen los bloqueos mutuos y existe una fórmula simple para calcular el tiempo de bloqueo.

13.12 Un modelo de proceso extensible

Se ha destacado anteriormente que el modelo descrito en la Sección 13.1 era demasiado simplista para ser práctico. En las siguientes secciones, se van a eliminar tres restricciones importantes:

- Los tiempos límite pueden ser menores que el periodo ($D < T$).
- Además de los procesos periódicos, se pueden soportar procesos esporádicos y aperiódicos.
- Se contemplan las interacciones entre los procesos, considerando los bloqueos resultantes en las ecuaciones del tiempo de respuesta.

Dentro de esta sección, se van a realizar cinco generalizaciones más. Sólo se considera la planificación de prioridad estática, ya que el análisis EDF no está maduro en estas áreas. La sección concluirá con un algoritmo de propósito general de asignación de prioridades.

13.12.1 Planificación cooperativa

Todos los modelos descritos anteriormente necesitan un distribuidor realmente apropiativo. En esta sección, se va a describir un esquema alternativo (la utilización de la apropiación diferida). Ésta tiene varias ventajas, además de seguir siendo analizable mediante la técnica representada por la Ecuación (13.7). En la Ecuación (13.7), aparece un término de bloqueo, B, que cuenta el tiempo que puede estar ejecutándose un proceso de menor prioridad mientras un proceso de alta prioridad es ejecutable. En el dominio de aplicación, esto puede producirse por la existencia de datos compartidos (bajo exclusión mutua) por procesos de diferente prioridad. Sin embargo, el bloqueo también puede ser causado por el sistema de ejecución o por el núcleo del sistema operativo. Muchos sistemas considerarán el cambio de contexto como el bloqueo más largo (por ejemplo, la activación de un proceso de alta prioridad está retardada por el tiempo que dura el cambio de contexto de un proceso de menor prioridad, incluso cuando se va a tener que producir otro cambio de contexto inmediatamente para el proceso de alta prioridad).

Una de las ventajas de la utilización del protocolo inmediato de acotación de la prioridad (para calcular y delimitar B), es que el bloqueo no es acumulativo. Un proceso no puede ser blo-

queado a la vez por un proceso de aplicación y por una rutina del núcleo (realmente, sólo podría darse una de las dos situaciones cuando se activa un proceso de prioridad mayor).

La planificación cooperativa explota esta propiedad no acumulativa incrementando las situaciones en las que se pueden producir bloqueos. Sea B_{MAX} el máximo tiempo de bloqueo en el sistema (utilizando un enfoque convencional). El código de la aplicación se divide entonces en bloques no desalojables (o apropiables), cuyos tiempos de ejecución están limitados por B_{MAX}. Al final de cada uno de estos bloques, el código de la aplicación realiza una petición al núcleo del sistema para «desplanificar». Si un proceso de alta prioridad está en estado ejecutable ahora, el núcleo iniciará un cambio de contexto; si no, el proceso actualmente en ejecución continuará en el siguiente bloque no desalojable.

La ejecución normal del código de la aplicación es, por lo tanto, totalmente cooperativa. Un proceso continuará su ejecución hasta que se ofrezca como desplanificable. Por ello, se asegura la exclusión mutua siempre que cualquier sección crítica esté totalmente contenida entre dos invocaciones de desplanificación. Este método requiere, por lo tanto, ubicar cuidadosamente las llamadas de desplanificación.

Para ofrecer algún nivel de protección contra el software corrupto (o incorrecto), el núcleo podría utilizar una señal asíncrona, o abort, para eliminar el proceso de la aplicación si cualquier bloque no desalojable dura más que B_{MAX}.

La utilización de la apropiación diferida tiene dos ventajas importantes: incrementa la planificabilidad del sistema, y puede conducir a valores menores de C. En la solución de la Ecuación (13.4), según se extiende el valor de w, serán posibles nuevas activaciones de procesos de alta prioridad, lo que incrementará más el valor de w. En la apropiación diferida, no se puede producir interferencia alguna durante el último bloque de ejecución. Sea F_i el tiempo de ejecución del bloque final, tal que cuando el proceso ha consumido $C_i - F_i$ unidades de tiempo, el último bloque acaba de comenzar. Ahora se resuelve la Ecuación (13.4) para $C_i - F_i$, en lugar de para C_i.

$$w_i^{n+1} = B_{MAX} + C_i - F_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j \qquad \textbf{(13.10)}$$

Cuando converge (esto es, cuando $w_i^{n+1} = w_i^n$), el tiempo de respuesta viene dado por:

$$R_i = w_i^n + F_i \qquad \textbf{(13.11)}$$

En realidad, el último bloque del proceso se ha ejecutado con una prioridad más alta (la más alta) que la del resto de los procesos.

La otra ventaja de la planificación diferida viene de la mayor precisión con la que se predicen los tiempos de ejecución de los bloques no desalojables de los procesos. Los procesadores modernos tienen cachés, colas de precaptura y encadenamiento de instrucciones que reducen significativamente los tiempos de ejecución del código lineal. Habitualmente, las estimaciones simples del peor caso del tiempo de ejecución se ven obligadas a ignorar estas ventajas, y obtienen unos resultados muy pesimistas, debido a que la expropiación puede invalidar las cachés y los encadenamientos. En la práctica, el conocimiento de la no expropiación puede utilizarse pa-

ra predecir realmente la velocidad del código. Esto, sin embargo, puede volverse en contra si el costo de ofrecer la desplanificación es grande. Véase la Sección 16.3.4 para una discusión de cómo modelar el impacto de los efectos de la caché sobre el análisis de la planificabilidad.

13.12.2 Fluctuaciones en la activación

En el modelo simple, se supone que todos los procesos son periódicos y que van a ser activados con una periodicidad perfecta; esto es, si el proceso l tiene un periodo T_l, será activado exactamente con esa frecuencia. Los procesos esporádicos se incorporan en el modelo suponiendo que su intervalo mínimo entre llegadas es T. Sin embargo, esto no es siempre una suposición realista. Considere un proceso esporádico s que está siendo activado por un proceso periódico l (en otro procesador). El periodo del primer proceso es T_l, y el proceso esporádico tendrá la misma tasa, pero es incorrecto suponer que la carga máxima (interferencia) que s ejerce sobre los procesos de baja prioridad se pueda representar en las Ecuaciones (13.4) o (13.5) como un proceso periódico con periodo $T_s = T_l$.

Para entender por qué esto es insuficiente, considere dos ejecuciones consecutivas del proceso l. Suponga que el evento que activa el proceso s ocurre al final de cada ejecución del proceso periódico. Suponga también que en la primera ejecución del proceso l, el proceso no acaba hasta su último instante posible, es decir, R_l. Sin embargo, en la siguiente invocación imagine que no existe interferencia del proceso l, de forma que acaba dentro de C_l. Como este valor puede ser arbitrariamente pequeño, supongamos que es igual a cero. Las dos ejecuciones del proceso esporádico no están separadas por T_l, sino por $T_l - R_l$. La Figura 13.10 ilustra este comportamiento para T_l igual a 20, R_l igual a 15, y el C_l mínimo igual a 1 (esto es, dos activaciones del proceso esporádico en 6 unidades de tiempo). Tenga en cuenta que este fenómeno resulta interesante sólo si el proceso l es remoto. Si no fuera así, las variaciones en la activación del proceso s podrían ser consideradas en las ecuaciones estándares, donde se puede suponer un instante crítico entre el activador y el activado.

Para capturar correctamente la interferencia de los procesos esporádicos sobre los demás procesos, debe modificarse la relación de recurrencia. El valor máximo de la variación en la activación de un proceso se denomina *fluctuación* (jitter), y se representa por J. Por ejemplo, en el caso

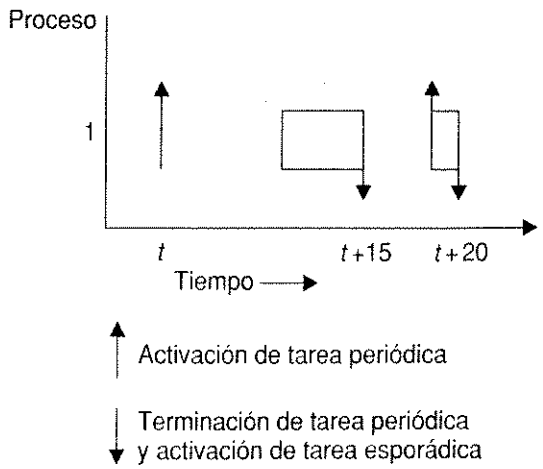


Figura 13.10. Activación de procesos esporádicos.

anterior, el proceso s podía tener un valor de fluctuación de 15. En cuanto a su impacto máximo sobre los procesos de menor prioridad, este proceso esporádico será activado en los instantes 0, 5, 25, 45, etc. Esto es, en los instantes 0, $T-J$, $2T-J$, $3T-J$, etc. El examen de la derivación de la ecuación de planificabilidad implica que el proceso i sufrirá una interferencia del proceso s si R_i está entre 0 y $T-J$, $R_i \in [0, T-J)$; dos interferencias si $R_i \in [T-J, 2T-J)$; tres si $R_i \in [2T-J; 3T-J)$; etc. Un ligero reacomodo de estas condiciones muestran un único efecto si $R_i + J \in [0, T)$; un doble efecto si $R_i + J \in [T, 2T)$; etc. Esto se puede representar, de la misma forma que las ecuaciones de tiempo de respuesta anteriores, como sigue (Audsley et al., 1993a):

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \qquad \textbf{(13.12)}$$

En general, los procesos periódicos no sufren fluctuaciones en su activación. Sin embargo, cada implementación puede restringir la granularidad del temporizador del sistema (el cual activa los procesos periódicos). En esta situación, un proceso periódico también puede sufrir fluctuaciones en su activación. Por ejemplo, un valor de T de 10 pero una granularidad del sistema de 8 implicará un valor de fluctuación de 6 (en el instante 16 el proceso será activado correspondiendo a su invocación del instante «10»). Si se tiene que medir el tiempo de respuesta (ahora designado como $R_i^{periódico}$) relativo al tiempo de respuesta real, entonces se debe añadir el valor de fluctuación al calculado previamente:

$$R_i^{periódico} = R_i + J_i \qquad \textbf{(13.13)}$$

Si este nuevo valor es mayor que T_i, entonces habrá que hacer el siguiente análisis.

13.12.3 Tiempos límite arbitrarios

Hay que adaptar de nuevo el análisis para atender a aquellas situaciones en las que D_i (y potencialmente R_i) puede ser mayor que T_i. Cuando el tiempo límite es menor (o igual) que el periodo, resulta necesario considerar sólo una única activacion de cada proceso. El instante crítico, cuando todos los procesos de alta prioridad se activen al mismo tiempo, representa la máxima interferencia, y por lo tanto, el peor caso del tiempo de respuesta debe darse en el instante crítico. Sin embargo, cuando el tiempo límite es mayor que el periodo, se pueden considerar varias activaciones. A continuación, se supone que la activación de un proceso será retardada hasta que haya acabado cualquier ejecución anterior del mismo proceso.

Si un proceso se ejecuta dentro del siguiente periodo, entonces ambas activaciones deberán ser analizadas para ver cuál conduce a un tiempo de respuesta mayor. Más aún, si la segunda activación no se completa antes de que ocurra una tercera, entonces habrá que considerar esta nueva activación también, y así sucesivamente.

Para cada activación potencialmente solapada, se define una ventana separada $w(q)$, donde q es un entero que identifica una ventana en particular ($q = 0, 1, 2, ...$). Se puede extender la Ecuación (13.5) para hacer que tenga la siguiente forma (ignorando las fluctuaciones) (Tindell et al., 1994):

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q)}{T_j} \right\rceil C_j \quad \textbf{(13.14)}$$

Por ejemplo, con q igual a 2, se producirán tres activaciones del proceso en la ventana. Para cada valor de q, se puede encontrar un valor estable para $w(q)$ por iteración: –como en la Ecuación (13.5)–. El tiempo de respuesta viene dado por:

$$R_i(q) = w_i^n(q) - qT_i \quad \textbf{(13.15)}$$

por ejemplo, con $q = 2$, el proceso comenzó en $2T_i$ dentro de la ventana, y por lo tanto el tiempo de respuesta es el tamaño de la ventana menos $2T_i$.

El número de activaciones a tener en cuenta está limitado por el valor más pequeño de q para el cual es cierta la siguiente relación:

$$R_i(q) \leq T_i \quad \textbf{(13.16)}$$

En este punto, el proceso acaba antes que la siguiente activación, y por lo tanto las siguientes ventanas no se solaparán. El peor caso del tiempo de respuesta se produce cuando se da el valor máximo para cada q:

$$R_i = \max_{q=0,1,2,\ldots} R_i(q) \quad \textbf{(13.17)}$$

Tenga en cuenta que para $D \leq T$, la relación en la Ecuación (13.16) es cierta cuando $q = 0$ (si se puede garantizar el proceso), en cuyo caso se simplifican las Ecuaciones (13.14) y (13.15), convirtiéndose en la ecuación original. Si en cualquier caso $R > D$, entonces el proceso no es planificable.

Cuando esta formulación del tiempo límite arbitrario se combina con el efecto de la fluctuación en la activación, se deben realizar dos modificaciones sobre el análisis anterior. Primero, como antes, se debe incrementar el factor de interferencia si cualquier proceso de prioridad mayor sufre una fluctuación en la activación:

$$w_i^{n+1}(q) = B_i + (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n(q) + J_j}{T_j} \right\rceil C_j \quad \textbf{(13.18)}$$

El otro cambio involucra al proceso en sí mismo. Si puede sufrir fluctuaciones en la activación, entonces dos ventanas consecutivas podrían solaparse si el tiempo de respuesta más la activación resultara mayor que el periodo. Para tener esto en cuenta debe modificarse la Ecuación (13.15):

$$R_i(q) = w_i^n(q) - qT_i + J_i \quad \textbf{(13.19)}$$

13.12.4 Tolerancia a fallos

La tolerancia a fallos mediante recuperación de errores, ya sea hacia adelante o hacia atrás, siempre implica cálculos extra. Éste podría ser un manejador de excepciones o un bloque de recupe-

ración. En sistemas de tiempo real tolerantes a fallos, los tiempos límite deberían cumplirse incluso ante ciertos niveles de fallo. Este nivel de tolerancia a fallos se conoce como *modelo de fallos*. Si C_i^f es el tiempo de computación extra que resulta de un error en el proceso i, entonces la ecuación del tiempo de respuesta se puede modificar fácilmente.

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in mpi(i)} C_k^f \tag{13.20}$$

donde $mpi(i)$ es el conjunto de procesos con mayor prioridad o igual que i.

Por ello, el modelo de fallos considera un máximo de un fallo, y existe la suposición de que un proceso ejecutará su acción de recuperación con la misma prioridad que sus tareas normales. Resulta fácil modificar la Ecuación (13.20) para incrementar el número de fallos permitidos (F):

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in mpi(i)} FC_k^f \tag{13.21}$$

En efecto, se puede analizar un sistema para incrementar los valores de F con el fin de ver el número de fallos (que llegan en una ráfaga) que se pueden tolerar. Alternativamente, el modelo de fallos puede indicar un intervalo de llegada mínimo para fallos, y en este caso la ecuación se convierte en:

$$R_i = B_i + C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in mpi(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right) \tag{13.22}$$

donde T_f es el tiempo mínimo entre llegadas para fallos.

En las Ecuaciones (13.21) y (13.22), la suposición se ha realizado en el peor caso, de forma que el fallo siempre se dará en el proceso que tiene el tiempo de recuperación más grande.

13.12.5 Introducción de desplazamientos

En el análisis de planificabilidad presentado al principio del este capítulo, se supone que todos los procesos comparten un tiempo de activación común. El instante crítico es aquél en el que se activan todos los procesos a la vez (esto suele ocurrir en el instante 0). Para la planificación de prioridades estáticas, esta suposición resulta segura; si todos los procesos cumplen sus requisitos de temporización cuando son activados conjuntamente, siempre serán planificables. Existen, sin embargo, conjuntos de procesos que se pueden beneficiar de la elección explícita de sus tiempos de activación de forma que no compartan un instante crítico. Se podría decir que un proceso tiene un **desplazamiento** con respecto a otros. Como ejemplo, considere los tres procesos definidos en la Tabla 13.13.

Si se supone un instante crítico, entonces el proceso a tiene un tiempo de respuesta de 4, y el proceso b tiene un tiempo de respuesta de 8; pero el tercer proceso tiene el peor caso de tiempo de respuesta (16), que está fuera de su tiempo límite. Para el proceso c, la interferencia del proceso b es suficiente para forzar una interferencia más de a, lo que resulta crucial. Sin embargo,

Tabla 13.13. Ejemplo de un conjunto de procesos.

Proceso	*T*	*D*	*C*
a	8	5	4
b	20	10	4
c	20	12	4

si el proceso *c* recibe un desplazamiento (*O*) de 10 (esto es, retiene el mismo periodo y tiempo límite relativos, pero su primera activación es en el instante 10), entonces nunca se ejecutará a la vez que *b*. El resultado es un conjunto planificable de conjunto de procesos (véase la Tabla 13.14).

Tabla 13.14. Análisis del tiempo de respuesta del conjunto de procesos.

Proceso	*T*	*D*	*C*	*O*	*R*
a	8	5	4	0	4
b	20	10	4	0	8
c	20	12	4	10	8

Sin embargo, los conjuntos de procesos con desplazamientos arbitrarios no son fáciles de analizar. Elegir los desplazamientos de forma que un conjunto sea óptimamente planificable es un problema NP-completo. En efecto, está lejos de resultar sencillo comprobar siquiera si un conjunto de procesos con desplazamientos comparten un instante crítico.[4]

A pesar de este resultado teórico, existen conjuntos de procesos que pueden ser analizados de una forma relativamente sencilla (aunque no necesariamente óptima). En la mayoría de los sistemas reales, los periodos de los procesos no son arbitrarios, sino que guardan cierta relación entre ellos. Como en el ejemplo anteriormente comentado, dos procesos tienen un periodo común. En estas situaciones resulta sencillo dar a uno un desplazamiento (de *T*/2) y analizar el sistema resultante utilizando una técnica de transformación que elimine el desplazamiento (y por lo tanto permita aplicar el análisis del instante crítico). En este ejemplo, los procesos *b* y *c* (*c* con un desplazamiento de 10) son remplazados por un proceso artificial único con un periodo 10, tiempo de ejecución 4, y tiempo límite 10 pero sin desplazamiento. Este proceso artificial tiene dos propiedades importantes.

- Si es planificable (cuando comparte un instante crítico con todos los demás procesos), los dos procesos reales cumplirán sus tiempos límite cuando a uno se le dé la mitad del periodo como desplazamiento.

[4] Un resultado interesante es que un conjunto de procesos con periodos primos entre sí siempre tendrá un instante crítico independientemente de los desplazamientos elegidos (Audsley y Burns, 1998).

- Si todos los procesos de prioridad menor son planificables cuando sufren la interferencia del proceso artificial (y todos los demás procesos de alta prioridad), seguirán siendo planificables cuando el proceso artificial sea reemplazado por los dos procesos reales (uno con el desplazamiento).

Estas propiedades se derivan del hecho de que el proceso artificial siempre utiliza más (o el mismo) tiempo de UCP que los dos procesos reales. La Tabla 13.15 muestra el análisis que podría aplicarse al conjunto de procesos transformado. En esta tabla, el proceso artificial recibe el nombre de *n*.

Tabla 13.15. Análisis del tiempo de respuesta del conjunto de procesos transformado.

Proceso	*T*	*D*	*C*	*O*	*R*
a	8	5	4	0	4
n	10	10	4	0	8

De forma más general, los parámetros del proceso artificial se calculan a partir de los procesos *a* y *b* como sigue:

$T_n = T_a/2$ (*o* $T_b/2$, ya que $T_a = T_b$)
$C_n = \text{Max}(C_a, C_b)$
$D_n = \text{Min}(D_a, D_b)$
$P_n = \text{Max}(P_a, P_b)$

donde *P* denota prioridad.

Claramente, lo que resulta posible para dos procesos también resultará aplicable para tres o más procesos. En Bate y Burns (1997) se da una descripción completa de estas técnicas. En resumen, aunque los desplazamientos arbitrarios son realmente imposibles de analizar, el uso juicioso de los desplazamientos y las técnicas de transformación puede devolver el problema de su análisis al de un conjunto simple de procesos que comparten un instante crítico. Por lo tanto, serán aplicables todos los análisis dados en las secciones anteriores de este capítulo.

13.12.6 Asignación de prioridades

La formulación dada para los tiempos límite arbitrarios tiene la propiedad de que ninguno de los algoritmos simples (tales como la tasa o el tiempo límite monotónico) dan la ordenación óptima de prioridad. En esta sección, se presentan un teorema y un algoritmo para la asignación de prioridades en situaciones arbitrarias. El teorema considera el comportamiento de los procesos de menor prioridad (Audsley et al., 1993b).

> **Teorema** *Si un proceso* p *tiene asignada la menor de las prioridades y es realizable, entonces, si existe una ordenación de prioridades realizable para el conjunto completo de procesos, ese orden asignará al proceso* p *la menor de las prioridades.*

La demostración de este teorema deriva de considerar las ecuaciones de planificabilidad, por ejemplo la Ecuación (13.14). Si un proceso tiene la menor prioridad, sufre la interferencia de todos los procesos de prioridad mayor. Esta interferencia no depende del orden actual de esas prioridades mayores. Por tanto, si cualquier proceso es planificable con el menor valor, se le puede asignar ese lugar, y todo lo que resta es asignar el orden de las $N-1$ prioridades. Afortunadamente, el teorema se puede volver a aplicar al conjunto reducido de procesos. Por ello, a través de sucesivas aplicaciones, se obtiene un orden completo de prioridades (si existe alguno).

El siguiente código Ada implementa el algoritmo de asignación de prioridades: `Conjunto` es un array de procesos que está ordenado por prioridad; `Conjunto(N)` es la prioridad más alta, y `Conjunto(1)` es la más baja. El procedimiento `Test_Proceso` prueba si un proceso `K` es realizable en ese lugar en el array. En el doble bucle, primero se intercambian los procesos en la última posición hasta que se encuentra un resultado factible, este proceso se fija en esa posición. Entonces, se considera la siguiente posición de prioridad. Si en cualquier instante el bucle interior falla al buscar un proceso relizable, se abandona todo el procedimiento. Hay que destacar que se puede conseguir un algoritmo conciso si se efectúa un intercambio extra.

```
procedure Asignar_Pri (Conjunto : in out Conjunto_Procesos; N : Natural;
                       Ok : out Boolean) is
begin
  for K in 1..N loop
    for Sgte in K..N loop
      Intercambia(Conjunto, K, Sgte);
      Test_Proceso(Conjunto, K, Ok);
      exit when Ok;
    end loop;
    exit when not Ok; -- fallo al buscar proceso planificable
  end loop;
end Asignar_Pri;
```

Si el test de factibilidad es exacto (necesario y suficiente), entonces el orden de prioridad es óptimo. Por ello, para tiempos límite arbitrarios (sin bloqueos) se encuentra un orden óptimo.

13.13 Sistemas dinámicos y análisis en línea

Al comienzo de este capítulo se comentó la gran variedad de esquemas de planificación desarrollados para diferentes requisitos de aplicaciones. Para los sistemas de tiempo real estrictos, es de desear un análisis aparte, o fuera de línea (off-line); de hecho, a menudo resulta obligatorio. Para realizar dicho análisis se requiere:

- Patrones de llegada del trabajo entrante conocidos y limitados (esto conduce a un conjunto fijo de procesos en el que se conocen o bien los periodos, o bien el peor caso de los intervalos de llegada).

- Tiempos de computación limitados.
- Un esquema de planificación que conduzca a la ejecución predecible de los procesos de la aplicación.

Este capítulo ha mostrado cómo la planificación de prioridad estática (y una cierta extensión, EDF) pueden proporcionar un entorno de ejecución predecible.

Al contrario que en el caso de los sistemas estrictos, hay aplicaciones de tiempo real dinámicas flexibles en las que *a priori* no se conocen ni los patrones de llegada ni los tiempos de computación. Aunque aún se puede realizar algún nivel de análisis, éste no puede ser completo, y por lo tanto se necesita algún tipo de análisis en línea.

La principal tarea de un esquema de planificación en línea es la gestión de cualquier sobrecarga que pueda ocurrir debido a la dinámica del entorno del sistema. Anteriormente se destaco que EDF es un esquema dinámico que proporciona una política de planificación óptima. Sin embargo, EDF también posee la propiedad de que se comporta muy mal durante las sobrecargas transitorias. Resulta posible obtener un efecto en cascada en el que cada proceso incumple su tiempo límite, pero utiliza suficientes recursos como para propiciar que el siguiente proceso también incumpla su tiempo límite.

Para contrarestar el deterioro producido por este efecto dominó, la mayoría de los esquemas en línea tienen dos mecanismos:

(1) Un módulo de control de admisión, que limita el número de procesos que pueden competir por los procesadores.

(2) Una rutina de distribución EDF para los procesos admitidos.

Un algoritmo de admisión ideal previene la sobrecarga de los procesadores, de modo que la rutina EDF funcione perfectamente.

Si hay que admitir ciertos procesos y rechazar otros, es preciso conocer la importancia relativa de cada uno. Esto se suele hacer asignando un valor. Los valores se pueden clasificar como sigue.

- Estático: el proceso siempre tiene el mismo valor independientemente de cuándo se active.
- Dinámico: el valor del proceso sólo se puede calcular en el instante en el que se activa (debido a que depende de factores ambientales o del estado actual del sistema).
- Adaptativo: la naturaleza dinámica del sistema es tal, que el valor del proceso cambiará durante su ejecución.

Para asignar valores estáticos (o para construir el algoritmo y definir los parámetros de entrada para los esquemas dinámicos o adaptativos), se requieren especialistas en el dominio de aplicación para articular su conocimiento sobre el comportamiento deseable del sistema. Como en otras áreas de la computación, la elicitación del conocimiento no se consigue sin problemas. Éstos no serán considerados aquí –véase Burns et al. (2000)–.

Uno de los problemas fundamentales del análisis fuera de línea es la relación que hay que establecer entre la calidad de las decisiones de planificación y los recursos y el tiempo necesarios para tomar esas decisiones. En un extremo, cada vez que llegue un proceso nuevo, el conjunto completo de procesos podría ser objeto de un test exacto fuera de línea como los descritos en este capítulo. Si el conjunto de procesos no es planificable, se elimina el proceso de menor valor y se repite el test (hasta que se obtiene un conjunto planificable). Este enfoque, conocido como *el mejor posible* (best-effort), es óptimo para la asignación de valores estáticos o dinámicos, aunque sólo si se ignoran las sobrecargas de los test. Una vez parametrizadas las sobrecargas, la efectividad del enfoque se ve seriamente comprometida. En general, habrá que utilizar heurísticas en la planificación en línea, y es improbable que cada aproximación funcione para todas las aplicaciones. Ésta es un área de investigación aún activa. Resulta evidente, sin embargo, que lo que se necesita no es una única política definida en un lenguaje o en un sistema operativo estándar, sino mecanismos a partir de los cuales las aplicaciones puedan programar sus propios esquemas para cumplir con sus requisitos concretos.

El punto final a tratar en esta sección son los sistemas híbridos que contienen tanto componentes estrictos como dinámicos. Es probable que se conviertan en la norma en muchas áreas de aplicación. Incluso en bastantes sistemas estáticos, los cálculos de valor añadido, bajo la forma de procesos flexibles o firmes que mejoran la calidad de los procesos estrictos, son una forma atractiva de estructurar los sistemas. En estas circunstancias, según se dijo en la Sección 13.8.1, se debe proteger a los procesos estrictos de cualquier sobrecarga inducida por el comportamiento de los procesos no estrictos. Un modo de conseguir esto es utilizar una planificación de prioridades estática para los procesos estrictos, y servidores para el resto del trabajo. Los servidores pueden incorporar cualquier política de admisión deseable, y servir el trabajo dinámico entrante utilizando EDF.

13.14 Programación de sistemas basados en prioridad

Sólo unos pocos lenguajes de programación definen explícitamente las prioridades como parte de sus capacidades de concurrencia. Aquéllos que lo hacen, a menudo sólo proporcionan un modelo rudimentario. Por ejemplo, occam2 tiene una variación del constructor `PAR` que indica que se han de asignar prioridades estáticas a los procesos designados:

```
PRI PAR
  P1
  P2
  PAR
    P3
    P4
  P5
```

Aquí, se han utilizado prioridades relativas, dando sentido al orden textual de los procesos en el `PRI PAR`. Por ello, `P3` y `P4` comparten el siguiente nivel de prioridad, y `P5` tiene la menor prioridad. No existe un rango mínimo de prioridades que deban soportar las distintas implementaciones, ni soporte de herencia de prioridad.

El único lenguaje que intenta aportar una visión más completa es Ada, por lo que será discutido en detalle en la siguiente subsección. Tradicionalmente, la planificación basada en prioridad ha sido un asunto más propio de los sistemas operativos que de los lenguajes. Por lo tanto, después de la discusión sobre Ada, se revisarán las capacidades de POSIX. Ada y POSIX suponen que cualquier análisis de planificabilidad puede realizarse fuera de línea. Más recientemente, Java para tiempo real ha intentado proporcionar las mismas capacidades que Ada y POSIX, pero con la opción de soportar, en línea, el análisis de realizabilidad. Esto se considerará en la Sección 13.14.3.

Quizás habría que destacar en este punto que, aunque la mayoría de los sistemas operativos de propósito general proporcionan la noción de prioridad de procesos o hilos, sus funcionalidades suelen ser inadecuadas para la programación de tiempo real estricto.

13.14.1 Ada

Según se dijo en el Prefacio, Ada se definió como un núcleo de lenguaje más un número de anexos para dominios de aplicación especializados. Estos anexos no contienen ninguna característica nueva del lenguaje (en forma de sintaxis nueva), pero definen pragmas y bibliotecas de paquetes que deben ser soportados si se va adherir un anexo particular. Esta sección considera alguna de las cosas que ofrece Anexo de Sistemas de Tiempo Real. En concreto, aquéllas que permiten que se asignen prioridades a tareas (y a objetos protegidos).[5] En el paquete `System`, se encuentran las siguientes declaraciones:

```
subtype Any_Priority is Integer range definido-en-la-implementación;
subtype Priority is Any_Priority range
                    Any_Priority'First .. definido-en-la-implementación;
subtype Interrupt_Priority is Any_Priority range
                  Priority'Last+1 .. Any_Priority'Last;
Default_Priority : constant Priority :=
                    (Priority'First + Priority'Last)/2;
```

Se divide un rango de tipo entero entre las prioridades estándares y el rango (mayor) de prioridades de interrupción. Cada implementación debe soportar un rango para `System.Priority` de al menos 30 valores, y al menos un valor distinto para `System.Interrupt_Priority`.

La prioridad inicial de una tarea se establece incluyendo un pragma en su especificación:

[5] Las prioridades también se pueden asignar a las colas de entrada y a la operación de la sentencia de selección. Esta sección se centrará, sin embargo, en las prioridades de las tareas y en las cotas de prioridad de los objetos protegidos.

```
task Controlador is
  pragma Priority(10);
end Controlador;
```

Si la definición de tipo de tarea contiene un pragma para fijar la prioridad, entonces todas las tareas de ese tipo tendrán la misma prioridad, a menos que se utilice un discriminante:

```
task type Servidores(Prioridad_Tarea : System.Priority) is
  entry Servicio1(...);
  entry Servicio2(...);
  pragma Priority(Prioridad_Tarea);
end Servidores;
```

Para aquellas entidades que actúan como manejadores de interrupciones, se define un pragma especial:

```
pragma Interrupt_Priority(Expresión);
```

o, simplemente,

```
pragma Interrupt_Priority;
```

La definición y uso de un pragma diferente para los niveles de interrupción mejora la legibilidad de los programas y ayuda a eliminar errores que se pueden producir si se confunden los niveles de prioridad de interrupciones y tareas. Sin embargo, la expresión utilizada en `Interrupt_Priority` se evalúa a la baja como una `Any_Priority`, y por lo tanto es posible asignar una prioridad relativamente baja a un manejador de interrupciones. Si se omite la expresión, se asigna el valor más alto posible.

Una prioridad asignada utilizando uno de esos pragmas se denomina **prioridad base**. Una tarea también podrá tener una **prioridad activa** que será mayor que ella (esto será explicado a su debido tiempo).

El programa principal, que se ejecutará bajo la noción de un entorno de tarea, puede fijar su prioridad colocando el pragma `Priority` en el subprograma principal. Si no se hace esto, se utiliza el valor por defecto definido en `System`. Cualquier otra tarea que omita la utilización del pragma tiene una prioridad base por defecto igual a la de la tarea que la creó.

Para utilizar ICPP, un programa Ada debe incluir el siguiente pragma:

```
pragma Locking_Policy(Ceiling_Locking);
```

Una implementación puede definir otras políticas de bloqueo; en el anexo de sistemas de tiempo real sólo se exige `Ceiling_Locking`. La política por defecto, si se omitiera el pragma, dependerá de la implementación. Para especificar la cota de prioridad de cada objeto protegido, se utilizan los pragmas `Priority` e `Interrupt_Priority`, definidos en las secciones previas. Si no existiera un pragma, se tomaría como cota `System.Priority'Last`.

Si una tarea invoca a un objeto con una prioridad mayor que la cota definida, se genera la excepción `Program_Error` . Si se permitiera tal invocación, podría darse una violación de la exclusión mutua sobre el objeto. Si es un manejador de interrupciones el que lo invoca con una prioridad inadecuada, entonces el programa se considera erróneo. Eso debe ser eventualmente prevenido mediante los test adecuados y el correcto análisis estático y dinámico del programa.

Con `Ceiling_Locking`, una implementación efectiva utilizará el hilo de la tarea invocadora para ejecutar no sólo el código de la llamada protegida, sino también el código de cualquier otra tarea que sea activada por las acciones de la llamada original. Por ejemplo, considérese el siguiente objeto protegido simple:

```
protected Control_Puerta is
  pragma Priority(28);
  entry Para_Y_Cierra;
  procedure Abre;
private
  Puerta: Boolean := False;
end Control_Puerta;

protected body Control_Puerta is
  entry Para_Y_Cierra when Puerta is
  begin
    Puerta := False;
  end Para_Y_Cierra;

  procedure Abre is
  begin
    Puerta := True;
  end Abre;
end Control_Puerta;
```

Suponga que una tarea `T`, con prioridad 20, invoca `Para_Y_Cierra` y se bloquea. Después, un tarea `S` (prioridad 27) invoca `Abre`. El hilo que implementa `S` tendrá que emprender las siguientes acciones:

(1) Ejecutar el código de `Abre` para `S`.

(2) Evaluar la barrera a la entrada y darse cuenta de que `T` está en proceso.

(3) Ejecutar el código de `Para_Y_Cierra` para `T`.

(4) Evaluar la barrera otra vez.

(5) Continuar con la ejecución de `S` después de invocar al objeto protegido.

En todo ello, no se ha producido un solo cambio de contexto. La alternativa para `S` es hacer que `T` sea ejecutable en el punto (2); `T` tendrá ahora una prioridad mayor (28) que la de `S` (27), y por

lo tanto el cambio de contexto sería para T, que completaría su ejecución dentro de Control_Puerta. Cuando salga T, se necesitará otro cambio de contexto para volver a colocar a S. Esto resulta mucho más caro.

Cuando entra una tarea en un objeto protegido, su prioridad se puede incrementar por encima del nivel de prioridad base definido por los pragmas Priority o Interrupt_Priority. La prioridad utilizada para determinar el orden de despacho es la **prioridad activa** de una tarea. Esta prioridad activa es el máximo de la prioridad base de la tarea y cualquier otra prioridad que hubiera heredado.

El uso de objetos protegidos es una forma mediante la cual una tarea puede heredar una prioridad activa más alta. Existen otras, como por ejemplo:

- Durante la activación: una tarea heredará la prioridad activa de la tarea padre que la crea; recuerde que la tarea padre está bloqueada hasta que sus tareas hijas acaban, lo cual podría producir una inversión de prioridad si no existiera esta regla de herencia.
- Durante una cita: la tarea que ejecuta la sentencia de aceptación heredará la prioridad activa de la tarea que realiza la llamada a la entrada (si es mayor que su propia prioridad).

En el último caso, no es necesario eliminar todas las posibles inversiones de la prioridad. Considere una tarea servidor S, con una entrada E y una prioridad base L (baja). Una tarea de alta prioridad realiza una llamada sobre E. Una vez que la cita ha comenzado, S se ejecutará con la prioridad más alta. Pero antes de que S realice la sentencia de aceptación para E, se ejecutará con una prioridad baja L (incluso aunque la tarea de alta prioridad esté bloqueada). Cualquier implementación puede soportar ésta y otras formas candidatas de herencia de prioridad. La implementación debe proporcionar, sin embargo, un pragma que el usuario pueda emplear para seleccionar las condiciones adicionales de forma explícita.

El Anexo de Sistemas de Tiempo Real pretende proporcionar características flexibles y extensibles. Es evidente que esto no resulta fácil. Ada 83 padeció el hecho de ser demasiado preceptivo. Sin embargo, la pérdida de una política de distribución definida podría ser desafortunada si no ayuda al desarrollo del software o a la portabilidad. Si se han definido prioridades base, entonces se supone que se va a emplear una planificación apropiativa basada en prioridades. En un sistema multiprocesador, esta implementación depende de si se considera cada procesador por separado o el conjunto de procesadores.

Para aportar extensibilidad, se puede seleccionar la política de despacho utilizando el siguiente pragma:

```
pragma Task_Dispatching_Policy(Identificador_Política);
```

El anexo de Sistemas de Tiempo Real define una posible política, FIFO_Within_Priority, donde las tareas que comparten la misma prioridad son encoladas en un orden FIFO. Por lo tanto, según llegan tareas ejecutables se colocan al final de una cola de ejecución conceptual para el nivel de prioridad dado. Una excepción a esto se produce cuando se desaloja una tarea; en este caso, la tarea se coloca al frente de la cola de ejecución para ese nivel de prioridad.

Si un programa especifica la opción `Fifo_Within_Priority`, también debe elegir la política `Ceiling_Locking` definida anteriormente. Juntas, representan un modelo consistente y utilizable para construir, implementar y analizar programas de tiempo real.

Otras posibilidades de Ada

Ada también proporciona otras capacidades que resultan útiles para programar una amplia variedad de sistemas. Por ejemplo, prioridades dinámicas, colas de entrada priorizadas, atributos de tareas, capacidades de control de tareas asíncronas, y otras. El lector debe consultar los «Anexos de Programación de Sistemas y Tiempo Real» del *Manual de Referencia Ada*, o Burns y Wellings (1998), para más detalles.

Actualmente, no existe soporte en Ada para servidores esporádicos. Sin embargo, véase Harbour et al. (1998) para una discusión sobre cómo conseguir una aproximación a éstos.

13.14.2 POSIX

POSIX soporta planificación basada en prioridades, y tiene opciones para soportar herencia de prioridad y protocolos de acotación. Las prioridades se pueden establecer dinámicamente. Dentro de las capacidades basadas en prioridad, existen cuanto políticas:

- FIFO: un proceso/hilo se ejecuta hasta que acaba o es bloqueado; si un proceso/hilo es expropiado por un proceso/hilo de mayor prioridad, entonces se coloca en la cabeza de la cola de ejecución para su prioridad.
- *Round-Robin* (turno rotatorio): un proceso/hilo se ejecuta hasta que acaba o es bloqueado, o hasta que su quantum de tiempo ha expirado; si un proceso/hilo es expropiado por un proceso de mayor prioridad, es colocado a la cabeza de la cola de ejecución para esa prioridad; sin embargo, si su quantum expira, será colocado al final.
- *Sporadic Server* (servidor esporádico): un proceso/hilo se ejecuta como un servidor esporádico (véase a continuación).
- OTHER (otra): política definida en la implementación (debe estar documentada).

Por cada política de planificación, existe un rango mínimo de prioridades que deben ser soportadas; para FIFO y *round-robin*, éste debe ser de al menos 32. La política de planificación se puede establecer por proceso y por hilo.

Se pueden crear hilos con la opción *system contention* (competición de sistema); en este caso, los hilos compiten con otros del sistema según su política y prioridad. Alternativamente, los hilos se pueden crear con la opción *process contention* (competición de proceso); en este caso, los hilos deben competir con los otros hilos (creados con la opción de competición de proceso) en el proceso padre. No está especificado el modo en que se planifican los hilos en relación con los hilos de otros procesos o con los hilos con competición global. La implementación debe decidir si soporta competición de sistema, competición de proceso, o ambos tipos.

Según se discutió en la Sección 13.8.2, un servidor esporádico asigna una cantidad limitada de la capacidad de la CPU para manejar eventos, tiene un periodo de reposición, cierta capacidad y dos prioridades. El servidor se ejecuta con la prioridad alta cuando le queda alguna capacidad y con la baja cuando se le ha agotado. Cuando un servidor se ejecuta con la prioridad alta, la cantidad de tiempo de ejecución que consume es substraída de su capacidad. La cantidad consumida se repone en el instante en el que el servidor fue activado más el periodo de reposición. Cuando la capacidad alcanza cero, la prioridad del servidor es puesta al valor bajo.

El Programa 13.1 ilustra una interfaz C para las capacidades de planificación POSIX. Las funciones están divididas entre aquéllas que manipulan la política de planificación y los parámetros de un proceso, y aquéllas que manipulan la política de planificación y los parámetros de un hilo. Si un hilo modifica la política y los parámetros de su propio proceso, el efecto en el hilo dependerá de su ámbito de competición (o nivel). Si está compitiendo a nivel del sistema, el cambio no afectará al hilo. Sin embargo, si está compitiendo en el nivel de proceso, existirá cierto impacto en el hilo.

Para prevenir la inversión de prioridad, POSIX permite que se puedan asociar protocolos de herencia en la variables mutuamente excluyentes. También se soporta el protocolo inmediato de acotación de la prioridad (llamado en POSIX protocolo de protección de prioridad).

Otras utilidades de POSIX

POSIX proporciona otras funcionalidades que resultan útiles en los sistemas de tiempo real. Por ejemplo, permite:

- Colas de mensajes ordenados por prioridad.
- Funciones para conseguir y establecer dinámicamente la prioridad de un hilo.
- Que los hilos indiquen cuándo sus atributos deberían ser heredados por cualquier hilo que ellos creen.

13.14.3 Java para tiempo real

Java para tiempo real posee la noción de objeto planificable: cualquier objeto que soporte la interfaz `Schedulable` dada en el Programa 13.2. Las clases `RealtimeThread`, `NoHeapRealTimeThread` y `AsyncEventHandler` soportan esta interfaz. Los objetos de estas clases tienen parámetros de planificación (véase el Programa 13.3). Se exige que las implementaciones Java para tiempo real soporten al menos 28 niveles de prioridad de tiempo real. De la misma manera que sucedía en Ada y en POSIX, cuanto mayor sea el valor entero, mayor será la prioridad (y, por lo tanto, mayor la posibilidad de ser elegido para ejecutar). Los hilos no de tiempo real reciben una prioridad por debajo de la prioridad de tiempo real mínima. Hay que destacar que los parámetros de planificación se ligan a los hilos en el momento de la creación de los mismos (véase el Programa 12.11). Si se modifican los objetos parámetro, esto afecta inmediatamente a los hilos asociados.

Programa 13.1. Una interfaz C típica para alguna de las funcionalidades de planificación POSIX.

```
#define SCHED_FIFO ...       /* planificación apropiativa con prioridades */
#define SCHED_RR ...         /* apropiativa con prioridades y quantum */
#define SCHED_SPORADIC ...   /* servidor esporádico */
#define SCHED_OTHER ...      /* planificador definido en la implementación */
#define PTHREAD_SCOPE_SYSTEM ... /* competición de sistema */
#define PTHREAD_SCOPE_PROCESS ... /* competición local */
#define PTHREAD_PRIO_NONE ... /* sin herencia de prioridad */
#define PTHREAD_PRIO_INHERIT ... /* herencia de prioridad básica */
#define PTHREAD_PRIO_PROTECT ... /* ICPP */

typedef ... pid_t;
struct sched_param {
   ...
   int sched_priority; /* usada en SCHED_FIFO y SCHED_RR */
   int sched_ss_low_priority
   timespec sched_ss_repl_period
   timespec sched_ss_init_budget
   int sched_ss_max_repl
   ... };

int sched_setparam(pid_t pid, const struct sched_param *param);
/* fija los parámetros de planificación de un proceso por su pid */

int sched_getparam(pid_t pid, struct sched_param *param);
/* ve los parámetros de planificación de un proceso por su pid */

int sched_setscheduler(pid_t pid, int politica,
                       const struct sched_param *param);
/* fija la política de planificación y parámetros de un proceso por pid */

int sched_getscheduler(pid_t pid);
/* da la política de planificación de un proceso por su pid */

int sched_yield(void);
/* produce que el proceso/hilo actual sea colocado al final */
/* de la cola de ejecución */

int sched_get_priority_max(int politica);
/* devuelve la máxima prioridad para la política dada */

int sched_get_priority_min(int politica);
/* devuelve la mínima prioridad para la política dada */

int sched_rr_get_interval(pid_t pid, struct timespec *t);
/* si pid /= 0, el quantum de tiempo para el proceso/hilo invocador *
 * es establecido en la estructura referenciada por t               *
```

```
 * si pid = 0, el quantum de tiempo para el proceso/hilo invocador  *
 * está en la estructura apuntada por t                             */

int pthread_attr_setscope(pthread_attr_t *attr,
                          int ambito_de_competicion);
/* fija el atributo ámbito de competición para un objeto atributo */
/* de un hilo */

int pthread_attr_getscope(const pthread_attr_t *attr,
                                  int *ambito_de_competicion);
/* obtiene el atributo ámbito de competición para un objeto atributo */
/* de un hilo */

int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                  int politica);
/* fija la política de planificación en un objeto atributo de un hilo */

int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                                  int *politica);
/* da la política de planificación para un objeto atributo de un hilo */

int pthread_attr_setschedparam(pthread_attr_t *attr,
                                  const struct sched_param *param);
/* fija la política de planificación para un objeto atributo de un hilo */

int pthread_attr_getschedparam(const pthread_attr_t *attr,
                                  struct sched_param *param);
/* da la política de planificación para un objeto atributo de un hilo */

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                    int protocolo);
/* fija el protocolo de herencia de prioridad */

int pthread_mutexattr_getprotocol(pthread_mutexattr_t *attr,
                                    int *protocolo);
/* da el protocolo de herencia de prioridad */

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                       int cota_prioridad);
/* fija la cota de la prioridad */

int pthread_mutexattr_getprioceiling(pthread_mutexattr_t *attr,
                                       int *cota_prioridad);
/* da la cota de la prioridad */

/* Las funciones anteriores devuelven 0 si tienen éxito excepto */
/* sched_get_priority_max y sched_get_priority_min */
```

Programa 13.2. La interfaz de Java para tiempo real `Schedulable`.

```
public interface Schedulable extends java.lang.Runnable
{
   public void addToFeasibility();
   public void removeFromFeasibility();

   public MemoryParameters getMemoryParameters();
   public void setMemoryParameters(MemoryParameters memoria);

   public ReleaseParameters getReleaseParameters();
   public void setReleaseParameters(ReleaseParameters activación);

   public SchedulingParameters getSchedulingParameters();
   public void setSchedulingParameters(SchedulingParameters planificación);

   public Scheduler getScheduler();
   public void setScheduler(Scheduler planificador);
}
```

Programa 13.3. La clase de Java para tiempo real `SchedulingParameters` y sus subclases.

```
public abstract class SchedulingParameters
{
   public SchedulingParameters();
}

public class PriorityParameters extends SchedulingParameters
{
   public PriorityParameters(int prioridad);

   public int getPriority();
   public void setPriority(int prioridad) throws
                           IllegalArgumentException;
   ...
}

public class ImportanceParameters extends PriorityParameters
{
   public ImportanceParameters(int prioridad, int importancia);
   public int getImportance();
   public void setImportance(int importancia);
   ...
}
```

De la misma manera que Ada y POSIX para tiempo real, Java para tiempo real soporta una política de despacho apropiativa basada en prioridades. Sin embargo, al contrario que Ada y POSIX para tiempo real, Java para tiempo real no necesita que el hilo expropiado sea colocado al comienzo de la cola de ejecución asociada con su nivel de prioridad. Java para tiempo real también soporta un planificador de alto nivel, cuyos propósitos son:

- Decidir cuándo admitir nuevos objetos planificables de acuerdo con los recursos disponibles y el algoritmo de factibilidad.
- Establecer la prioridad de los objetos planificables de acuerdo con el algoritmo de asignación de prioridades asociado con el algoritmo de realizabilidad.

Por ello, mientras que Ada y POSIX para tiempo real se centran en el análisis de planificabilidad fuera de línea, Java para tiempo real se enfrenta a sistemas más dinámicos con el potencial del análisis en línea.

En el Programa 13.4 se muestra la clase abstracta `Scheduler`. El método `isFeasible` considera sólo el conjunto de objetos planificables que se han añadido a la lista de factibilidad (utilizando los métodos `addToFeasibility` y `removeFromFeasibility`). El método `changeIfFeasible` comprueba si el conjunto de objetos continúa siendo realizable y si el objeto dado modificó sus parámetros de activación y de memoria. Si es así, se modifican los parámetros. Los métodos estáticos permiten interrogar o modificar el planificador por defecto.

Una vez definidas las subclases de la clase `Scheduler`, es la clase `PriorityScheduler` (definida en el Programa 13.5) la que implementa la planificación apropiativa basada en prioridad estándar.

Programa 13.4. La clase de Java para tiempo real `Scheduler`.

```
public abstract class Scheduler
{
   public Scheduler();
   protected abstract void addToFeasibility(Schedulable planificable);
   protected abstract void removeFromFeasibility(Schedulable planificable);

   public abstract boolean isFeasible();
   // comprueba el conjunto actual de objetos planificables

   public boolean changeIfFeasible(Schedulable planificable,
                  ReleaseParameters activación, MemoryParameters memoria);
   public static Scheduler getDefaultScheduler();
   public static void setDefaultScheduler(Scheduler planificador);

   public abstract java.lang.String getPolicyName();
}
```

Programa 13.5. La clase de Java para tiempo real `PriorityScheduler`.

```
class PriorityScheduler extends Scheduler
{
   public PriorityScheduler()

   protected void addToFeasibility(Schedulable s);
   protected void removeFromFeasibility(Schedulable s);

   public boolean isFeasible();
   // comprueba el conjunto actual de objetos planificables

   public boolean changeIfFeasible(Schedulable planificable,
                  ReleaseParameters activación, MemoryParameters memoria);

   protected void addToFeasibility(Schedulable s);
   protected void removeFromFeasibility(Schedulable s);

   public abstract boolean isFeasible();
   // checks the current set of schedulable objects

   public void fireSchedulable(Schedulable planificable);

   public int getMaxPriority();
   public int getMinPriority();
   public int getNormPriority();
   public java.lang.String getPolicyName();

   public static PriorityScheduler instance();

   ...
}
```

De nuevo, debe recalcarse que Java para tiempo real no requiere que cada implementación proporcione el algoritmo de factibilidad en línea. Simplemente, devolvería false a los métodos que comprueben la factibilidad.

Soporte de la herencia de prioridad

Java para tiempo real permite utilizar algoritmos de herencia de prioridad cuando se accede a clases sincronizadas. Para conseguirlo, se definen tres clases (véase el Programa 13.6). Considérese la clase siguiente:

Programa 13.6. Clases de Java para tiempo real que soportan herencia de prioridad.

```
public abstract class MonitorControl
{
   public MonitorControl();

   public static void setMonitorControl(MonitorControl política);
   // establece el defecto

   public static void setMonitorControl(java.lang.Object monitor,
                                        MonitorControl política);
   // establece una política para objetos individuales
}

public class PriorityCeilingEmulation extends MonitorControl
{
   public PriorityCeilingEmulation(int cota);

   public int getDefaultCeiling();
   // consigue la cota para este objeto
}

public class PriorityInheritance extends MonitorControl
{
   public PriorityInheritance();

   public static PriorityInheritance instance();
}
```

```
public class SynchronizeClass
{
  public void Método1() { ... };
  public void Método2() { ... };
}
```

Con las siguientes líneas de código, una instancia de esta clase puede fijar su protocolo de control como herencia de prioridad techo inmediata (llamada emulación de prioridad techo en Java para tiempo real) con una prioridad de 10:

```
SynchronizeClass SC = new SynchronizeClass();
PriorityCeilingEmulation PCI = new PriorityCeilingEmulation(10);
...
MonitorControl.setMonitorControl(SC, PCI);
```

13.14.4 Otras posibilidades de Java para tiempo real

Es importante destacar que todas las colas en Java para tiempo real están ordenadas por prioridad.

Otra característica que merece la pena mencionar, es que Java para tiempo real proporciona soporte para hilos aperiódicos bajo la forma de grupos de procesos. Se puede ligar un grupo de hilos aperiódicos y asignarle ciertas características que ayuden al análisis de la factibilidad. Se puede definir que los hilos no consuman más que cierto `coste` de tiempo de CPU en un periodo dado. En el Apéndice se muestra la descripción completa de la clase que soporta grupos de procesos.

Resumen

Un esquema de planificación tiene dos facetas: define un algoritmo para la compartición de recursos, y es un medio de predicción del peor caso del comportamiento de una aplicación que está utilizando esa forma de compartición de recursos.

La mayoría de los sistemas periódicos de tiempo real actuales se implementan utilizando un ejecutivo cíclico. Con este enfoque, el código de aplicación debe empaquetarse en un número fijo de «ciclos secundarios», de forma que la ejecución de la secuencia de ciclos secundarios (llamada «ciclo principal») permita que se cumplan todos los tiempos límite del sistema. Aunque es una estrategia de implementación efectiva para sistemas pequeños, este enfoque cíclico tiene una serie de inconvenientes.

- El empaquetado de los ciclos secundarios incrementa su dificultad según crece el sistema.
- Las actividades esporádicas son difícilmente acomodables.
- Los procesos con periodos grandes (mayores que el ciclo principal) son soportados de forma ineficiente.
- Los procesos con grandes tiempos de computación deben ser divididos, de forma que puedan ser empaquetados en series de ciclos secundarios.
- Es muy difícil alterar la estructura del ejecutivo cíclico para acomodar los cambios en los requisitos.

Debido a estas dificultades, este capítulo se ha centrado en el uso de esquemas de planificación basados en prioridades. Se han obtenido formas de calcular el tiempo de respuesta para un modelo flexible a partir de la descripción de un test simple basado en la utilización (el cual sólo es aplicable a un modelo de procesos restringido). Este modelo puede acomodar a procesos esporádicos, interacciones entre procesos, secciones no expropiables, fluctuaciones en la activación,

servidores aperiódicos, sistemas tolerantes a fallos, y relaciones arbitrarias entre los tiempos límite de los procesos (D) y su intervalo mínimo de llegadas (T).

La sincronización de procesos, tales como el acceso mutuamente excluyente a datos compartidos, puede producir inversión de la prioridad, a menos que se considere alguna forma de herencia de la prioridad. En este capítulo se han descrito dos protocolos con detalle: «herencia de cota de prioridad original» y «herencia de cota de prioridad inmediata».

En la planificación basada en prioridades, es importante que la asignación de prioridades refleje las características temporales de la carga de los procesos. En este capítulo se han descrito tres algoritmos:

- Tasa monotónica: para $D = T$.
- Tiempo límite monotónico: para $D < T$.
- Arbitrario: para $D > T$.

El capítulo concluía con ejemplos de cómo se puede llevar a cabo la planificación con prioridades estáticas en Ada, Java para tiempo real y POSIX para tiempo real.

Lecturas complementarias

Audsley, N. C., Burns, A., Davis, R., Tindell, K., y Wellings, A. J. (1995), «Fixed Priority Preemptive Scheduling: An Historical Perspective», *Real-Time Systems*, 8(3), 173-198.

Buttazzo, G. C. (1997), *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, New York: Kluwer Academic.

Burns, A., y Wellings, A. J. (1995), *Concurrency in Ada*, 2ª ed., Cambridge: Cambridge University Press.

Gallmeister, B. O. (1995), *Programming for the Real World POSIX.4*, Sebastopol, CA: O'Reilly.

Halbwachs, N. (1993), *Synchronous Programming of Reactive Systems*, New York: Kluwer Academic.

Klein, M. H. et al. (1993), *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*, New York: Kluwer Academic.

Joseph, M. (ed.) (1996), *Real-time Systems: Specification, Verification and Analysis*, Englewood Cliffs, NJ: Prentice Hall.

Natarajan, S. (ed.) (1995), *Imprecise and Approximate Computation*, New York: Kluwer Academic.

Rajkumar, R. (1993), *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, New York: Kluwer Academic.

Stankovic, J. A. et al. (1998), *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*, New York: Kluwer Academic.

Ejercicios

13.1 Tres procesos lógicos (*P*, *Q* y *S*) tienen las siguientes características. *P*: periodo 3; tiempo de ejecución necesario 1. *Q*: periodo 6; tiempo de ejecución necesario 2. *S*: periodo 18; tiempo de ejecución necesario 5.

Muestre cómo pueden planificarse estos procesos utilizando el algoritmo de planificación de tasa monotónica.

Muestre cómo podría construirse un ejecutivo cíclico para implementar los tres procesos lógicos.

13.2 Considere tres procesos *P*, *Q* y *S*. *P* tiene un periodo de 100 milisegundos, en el cual necesita 30 milisegundos de procesamiento. Los valores correspondientes para *Q* y *S* son 5,1 y 25,5 respectivamente. Suponga que *P* es el proceso más importante en el sistema, seguido de *S* y de *Q*.

(1) ¿Cuál es el comportamiento del planificador si la prioridad se basa en la importancia?

(2) ¿Cuál es la utilización del procesador de *P*, *Q* y *S*?

(3) ¿Cómo deberían planificarse los procesos de forma que se cumplan todos los tiempos límite?

(4) Ilustre uno de los esquemas que permite que estos procesos sean planificados.

13.3 Al conjunto anterior de procesos se le añade un cuarto proceso (*R*). El fallo de este proceso producirá que la seguridad se vea comprometida. *R* tiene un periodo de 50 milisegundos, pero su procesamiento depende de los datos, y varía entre 5 y 25 milisegundos. Discuta la forma en que debiera integrarse este proceso junto a *P*, *Q* y *S*.

13.4 La Figura 13.11 muestra el comportamiento de cuatro procesos periódicos: *w*, *x*, *y* y *z*. Estos procesos tienen unas prioridades determinadas por el esquema de tasa monotónico, con el resultado de que la prioridad(*w*) > prioridad(*x*) > prioridad(*y*) > prioridad(*z*).

El periodo de cada proceso comienza en el instante *S* y termina en el *T*. Los cuatro procesos comparten 2 recursos que están protegidos por los semáforos binarios *A* y *B*. En el diagrama, la etiqueta *A* (y *B*) implica una «operación wait sobre el semáforo»; la etiqueta *A'* (y *B'*) implica una «operación signal sobre el semáforo». La Tabla 13.16 resume los requisitos de los procesos.

El diagrama muestra las evolución de la ejecución de los cuatros procesos utilizando prioridades estáticas. Por ejemplo, *x* comienza en el instante 2, ejecuta con éxito una operación wait sobre *B* en el instante 3, y tiene que esperar en *A* en el instante 4 (*z* ya había

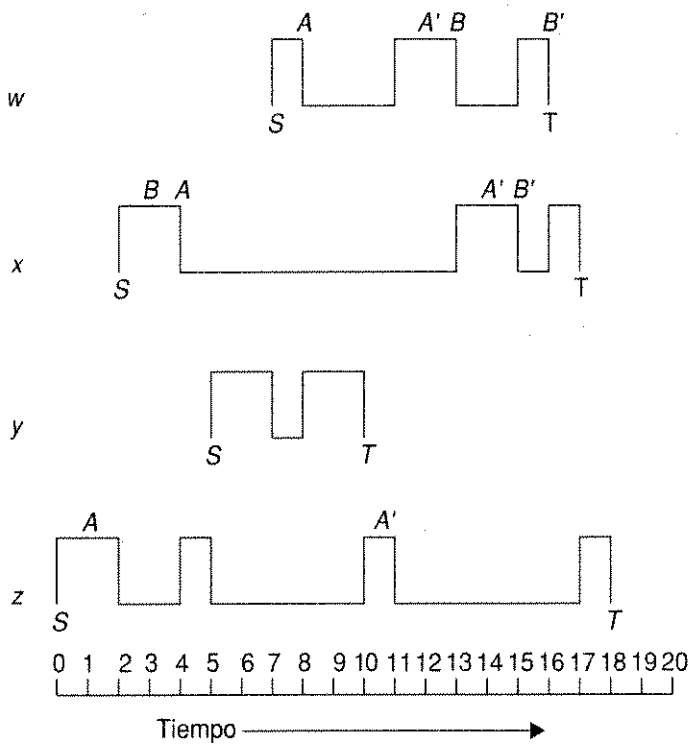


Figura 13.11. Comportamiento de los cuatro procesos periódicos del Ejercicio 13.4.

Tabla 13.16. Resumen de los requisitos de los procesos del Ejercicio 13.4.

Proceso	Prioridad	Tiempo inicio	Tiempo procesador	Semáforos usados
w	10	7	4	*A,B*
x	8	2	5	*A,B*
y	6	5	4	–
z	4	0	5	*A*

bloqueado *A*). En el instante 13 se ejecuta de nuevo (tiene bloqueado a *A*), activa *A* en el instante 14 y *B* en el 15. Entonces, es desalojado por *w*, pero se ejecuta otra vez en el instante 16. Finalmente termina en el instante 17.

Reelabore el diagrama dado en el Ejercicio 4 para ilustrar el comportamiento de estos procesos si se utiliza herencia de prioridad.

13.5 Reelabore el anterior diagrama del Ejercicio 4 para mostrar el comportamiento de estos procesos si se empleara herencia inmediata de cota de prioridad.

13.6 Con el protocolo de acotación de la prioridad es posible calcular el tiempo máximo que cualquier proceso puede ser bloqueado por la ejecución de un proceso de menor prioridad. ¿Cuál es la regla para calcular este bloqueo? Ilustre la respuesta calculando el máximo tiempo de bloqueo para cada proceso en el siguiente ejemplo. Un programa está compuesto de cinco procesos, *a*, *b*, *c*, *d* y *e* (listados en orden de prioridad, siendo *a* el de mayor prioridad), y de seis recursos, *R*1, ..., *R*6 (protegidos por semáforos que implementan

el protocolo de acotación de la prioridad). Los accesos a los recursos tienen un tiempo de ejecución en el peor caso que se muestra en la Tabla 13.17.

Tabla 13.17. Resumen de los requisitos de ejecución de los procesos del Ejercicio 13.6.

R1	*R2*	*R3*	*R4*	*R5*	*R6*
50 ms	150 ms	75 ms	300 ms	250 ms	175 ms

Los recursos son utilizados por los procesos de acuerdo a lo dispuesto en la Tabla 13.18.

Tabla 13.18. Resumen de los recursos requeridos por los procesos del Ejercicio 13.6.

Proceso	Usa
a	*R3*
b	*R1*, *R2*
c	*R3*, *R4*, *R5*
d	*R1*, *R5*, *R6*
e	*R2*, *R6*

13.7 ¿Es planificable el conjunto de procesos mostrado en la Tabla 13.19 utilizando el test simple basado en la utilización dado en la Ecuación (13.1)? ¿Es planificable el conjunto de procesos utilizando el análisis del tiempo de respuesta?

Tabla 13.19. Resumen de los atributos de los procesos del Ejercicio 13.7.

Proceso	Periodo	Tiempo ejecución
a	50	10
b	40	10
c	30	9

13.8 El conjunto de procesos mostrados en la Tabla 13.20 no es planificable utilizando la Ecuación (13.1), ya que al proceso *a* se le debe dar la prioridad tope debido a que es crítico. ¿Cómo se puede transformar para que sea planificable? Hay que tener en cuenta que los cálculos representados por *a* deben seguir recibiendo la prioridad máxima.

Table 13.20. Resumen de los atributos de los procesos del Ejercicio 13.8.

Proceso	Periodo	Tiempo ejecución	Crítico
a	60	10	ALTO
b	10	3	BAJO
c	8	2	BAJO

13.9 Los procesos mostrados en la Tabla 13.21 no son planificables utilizando la Ecuación (13.1), pero todos cumplen sus tiempos límite utilizando prioridades estáticas. Explique por qué.

Tabla 13.21. Resumen de los atributos de los procesos del Ejercicio 13.9.

Proceso	Periodo	Tiempo ejecución
a	75	35
b	40	10
c	20	5

13.10 En la Sección 13.8, se definió un proceso esporádico como aquél que tiene un intervalo mínimo entre llegadas. A menudo los procesos llegan en ráfagas. Actualice la Ecuación (13.4), para considerar ráfagas de tareas esporádicas en las que pueden darse N invocaciones arbitrariamente cercanas en un periodo T.

13.11 Amplíe la respuesta dada antes para considerar actividades esporádicas que llegan en ráfagas en las que puede haber N invocaciones en un periodo de T y cada invocación debe estar separada por al menos M unidades de tiempo.

13.12 ¿Hasta qué punto se pueden aplicar las ecuaciones de tiempo de respuesta dadas en este capítulo a recursos que no sean la CPU? Por ejemplo, ¿se pueden utilizar las ecuaciones para planificar el acceso a disco?

13.13 En un sistema de tiempo real de seguridad crítica, podrá utilizarse cierto conjunto de procesos para monitorizar eventos clave del entorno. Normalmente, habrá un tiempo límite definido entre la ocurrencia del evento y la producción de alguna salida (en respuesta a ese evento). Describa cómo se pueden utilizar procesos periódicos para monitorizar tales eventos.

13.14 Considere la lista de eventos (mostrados en la Tabla 13.22) junto con los costes de cómputo en respuesta a cada evento. Si se utiliza un proceso separado para cada evento (y estos procesos se implementan basándose en una planificación apropiativa basada en prioridades), describa cómo se puede aplicar el análisis de tasa monotónica para asegurar el cumplimiento de los tiempos límite.

Tabla 13.22. Resumen de los eventos del Ejercicio 13.14.

EVENTO	Tiempo límite	Tiempo computación
Evento *A*	36	2
Evento *B*	24	1
Evento *C*	10	1
Evento *D*	48	4
Evento *E*	12	1

13.15 ¿Cómo se puede planificar óptimamente el conjunto de procesos de la Tabla 13.23 (utilizando planificación de prioridades estáticas)? ¿Es planificable este conjunto de tareas?

Tabla 13.23. Resumen de las tareas del Ejercicio 13.15.

Proceso	*T*	*C*	*B*	*D*
a	8	4	2	8
b	10	2	2	5
c	30	5	2	30

13.16 Un diseñador de sistemas de tiempo real desearía ejecutar sobre el mismo procesador cierta mezcla de tareas Ada periódicas y esporádicas, siendo algunas críticas respecto a la seguridad, otras con respecto a la misión, y el resto no críticas. Él o ella está utilizando una planificación apropiativa basada en prioridades, y ha utilizado la ecuación de análisis del tiempo de respuesta para predecir que todas ellas cumplirán sus tiempos límite. Aporte razones sobre por qué, sin embargo, el sistema puede fallar en cuanto al cumplimiento de sus tiempos límite en tiempo de ejecución. ¿Qué mejoras podrían proporcionarse a los sistemas de soporte de ejecución de Ada para ayudar a eliminar estos problemas?

13.17 Muestre cómo se puede implementar en Ada la planificación del método de «primero el tiempo límite más temprano».

13.18 Describa, a grandes rasgos (con código), la forma en que Ada soporta las tareas esporádicas. ¿Cómo puede una tarea protegerse a sí misma de ejecutarse más frecuentemente que su intervalo mínimo ente llegadas?

13.19 ¿Hasta qué punto pueden ser implementados los servidores esporádicos en Ada?

13.20 Ada permite establecer dinámicamente la prioridad base de una tarea mediante el siguiente paquete.

```
with Ada.Task_Identification;
with System;
package Ada.Dynamic_Priorities is

   procedure Set_Priority(Prioridad : System.Any_Priority;
                          Tarea : Ada.Task_Identification.Task_Id :=
                          Ada.Task_Identification.Current_Task);
      -- genera un Program_Error si Tarea es la Null_Task_Id
      -- no tiene efecto si la tarea ha terminado

   function Get_Priority(Tarea : Ada.Task_Identification.Task_Id :=
            Ada.Task_Identification.Current_Task)
            return System.Any_Priority;
      -- genera un Tasking_Error si la tarea ha terminado
      -- o un Program_Error si Tarea es la Null_Task_Id
```

```
private
   ... -- no necesario en este caso
end Ada.Dynamic_Priorities;
```

Utilizando este paquete, muestre cómo implementar un protocolo de cambio de modo en el que se deben modificar las propiedades de un grupo de tareas en una operación atómica.

13.21 POSIX para tiempo real soporta cotas de prioridad dinámicas, mientras que Ada no lo hace. Explique los pros y los contras de soportar cotas de prioridad dinámicas.

13.22 ¿Hasta qué punto se puede utilizar `ProcessingGroupParameters` en un planificador Java para tiempo real para soportar servidores esporádicos?

13.23 Muestre cómo se puede implementar una planificación de «primero el tiempo límite más temprano», en Java para tiempo real, con un test de realizabilidad de utilización total del procesador menor del 100 por ciento.

Capítulo 14

Sistemas distribuidos

En los últimos treinta años, el coste de los microprocesadores y de la tecnología de comunicaciones ha seguido una trayectoria descendente en términos reales. Esto ha convertido a los sistemas distribuidos en una alternativa viable a los sistemas centralizados y monoprocesador en muchas áreas de aplicación de los sistemas embebidos. Las potenciales ventajas de la distribución incluyen:

- Mejores prestaciones para explotar el paralelismo.
- Mejor disponibilidad y fiabilidad para explotar la redundancia.
- Dispersión del poder de cómputo hacia donde se precisa.
- Posibilidad de un crecimiento incremental agregando o mejorando los procesadores y los enlaces de comunicación.

En este capítulo, se discuten algunos de los problemas que se presentan cuando se implementa un sistema de tiempo real sobre más de un procesador.

14.1 Definición de sistema distribuido

En lo que respecta a este capítulo, se define **sistema computacional distribuido** como un sistema de varios elementos de procesamiento autónomos que cooperan en un objetivo común o para lograr una meta común. Esta definición es suficientemente general como para satisfacer la mayoría de los conceptos intuitivos, sin descender a los detalles de dispersión física, medios de comunicación y demás. La definición excluye procesadores en disposición de tubería (pipeline) o en cadena (array), cuyos elementos no son autónomos; además, excluye aquellas redes de computadores (por ejemplo Internet) donde los nodos no trabajan hacia un objetivo común.[1] Dentro

[1] Sin embargo, según vaya mejorando la tecnología de comunicación, cada vez más sistemas interconectados en red se ajustarán a esta definición de sistema distribuido.

de esta definición entran la mayoría de las aplicaciones que uno pudiera, sensatamente, embeber sobre arquitecturas multiprocesador (por ejemplo operación y control, banca y otras aplicaciones aplicaciones comerciales orientadas a transacciones, y adquisición de datos). En la Figura 14.1 se muestra un sistema de fabricación basado en un esquema distribuido.

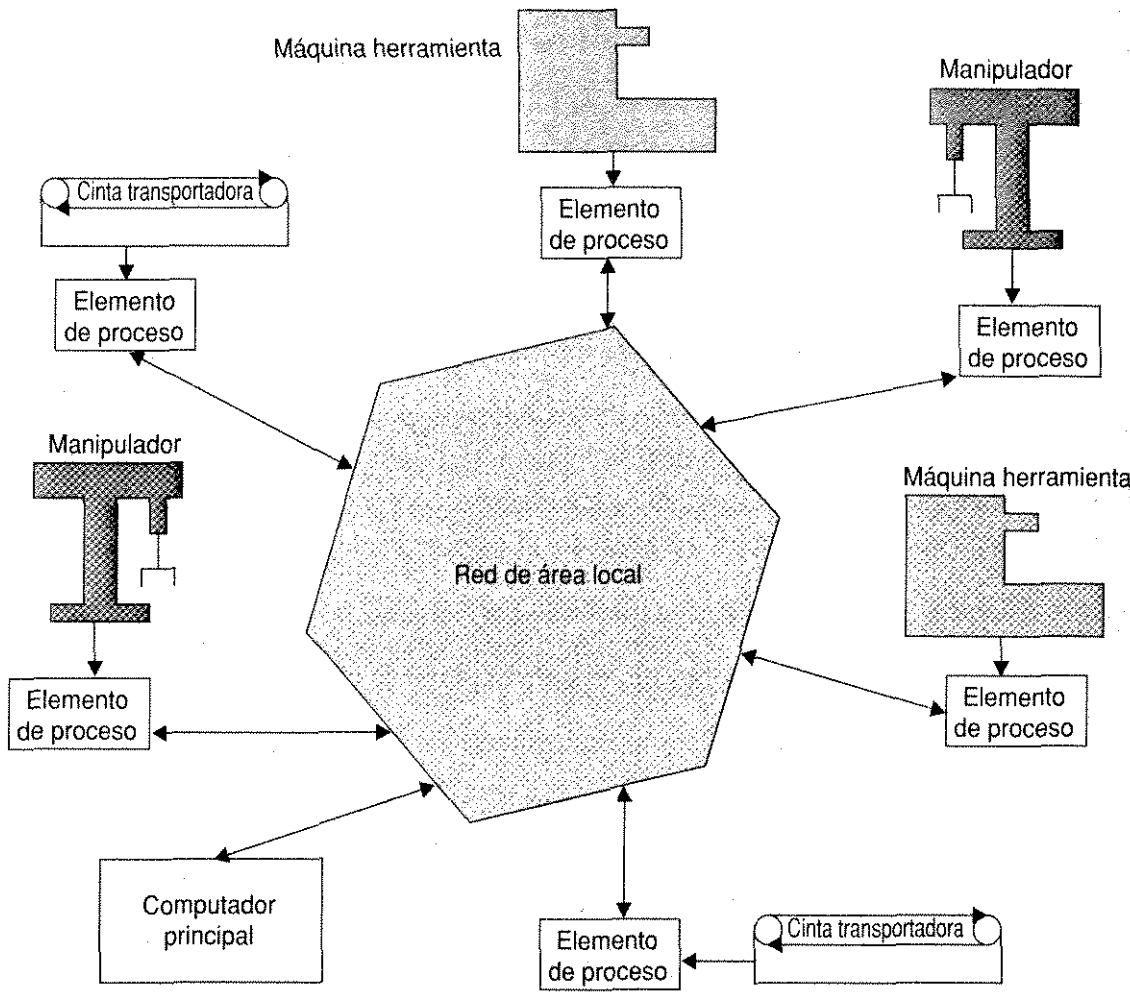


Figura 14.1. Un sistema distribuido embebido de fabricación.

Incluso los diseños de aviones modernos (tanto civiles como militares) contienen sistemas distribuidos embebidos. Por ejemplo, *Integrated Modular Avionics* (aviónica modular integrada) (AEEC, 1991) permite interconectar más de un módulo de procesamiento mediante un bus ARIND 629, como se muestra en la Figura 14.2.

Resulta útil clasificar los sistemas distribuidos en **fuertemente acoplados**, en los que los elementos de proceso, o nodos, tienen acceso a una memoria común, y **débilmente acoplados**, que carecen de ella. La importancia de esta clasificación radica en que la sincronización y la comunicación en un sistema fuertemente acoplado puede efectuarse mediante técnicas basadas en el empleo de variables compartidas, mientras que en un sistema débilmente acoplado se requiere, en último término, paso de mensajes.

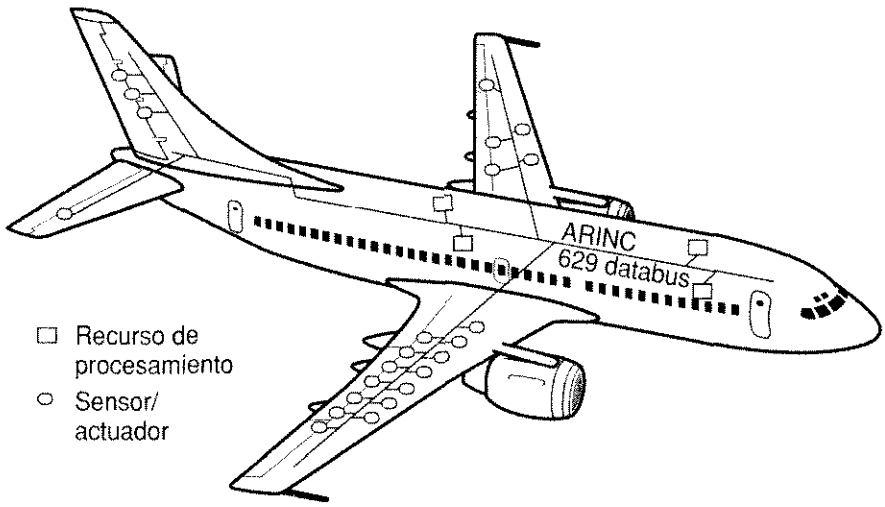


Figura 14.2. Un sistema distribuido embebido de aviónica civil.

En este capítulo se empleará la expresión «sistema distribuido» para las arquitecturas débilmente acopladas. Además, se supondrá en general una conectividad completa; las cuestiones relativas al rutado de mensajes y demás no se tendrán en cuenta. Véase Tanenbaum (1998) para una discusión completa de estos temas. Además, se supondrá que cada procesador sólo tiene acceso a su propio reloj, y que estos relojes se encuentran débilmente sincronizados (esto es, que pueden divergir en cierta medida).

Partiendo de la variedad de procesadores del sistema, podemos establecer otra clasificación. Un sistema **homogéneo** es aquél en el que todos los procesadores son del mismo tipo; un sistema **heterogéneo** contiene procesadores de tipos diferentes. Los sistemas heterogéneos presentan problemas por su diferente representación de programas y datos; estos problemas, aunque importantes, no se considerarán aquí. Este capítulo parte de la hipótesis de que todos los procesadores son homogéneos.

14.2 Panorámica de las cuestiones importantes

Hasta este punto del libro, la expresión programación concurrente se ha utilizado para discutir sobre la comunicación, sincronización y fiabilidad, sin involucrarse demasiado en cómo se implementan los procesos. Sin embargo, algunos de los temas que aparecen cuando se consideran aplicaciones distribuidas suscitan cuestiones fundamentales que van más allá de meros detalles de implementación. El fin de este capítulo es considerar estas cuestiones y sus implicaciones en las aplicaciones de tiempo real. Son las siguientes:

- **Soporte del lenguaje**. El proceso de escribir un programa distribuido se facilita en gran medida si el lenguaje y su entorno de programación soportan el particionado, la configuración, asignación y reconfiguración de la aplicación distribuida, junto a un acceso independiente de la ubicación de los recursos remotos.

- **Fiabilidad**. Disponer de varios procesadores permite que la aplicación sea tolerante a fallos; si bien, la aplicación deberá ser capaz de explotar esta redundancia. Aunque disponer de varios procesadores permite que la aplicación sea tolerante al fallo de un procesador, también introduce la posibilidad de que aparezcan otros fallos distintos de los que aparecen en un sistema centralizado monoprocesador. Estos fallos se asocian con fallos *parciales* del sistema, y el programa de la aplicación debe defenderse de ellos, o ser capaz de tolerarlos.

- **Algoritmos de control distribuidos**. La presencia de paralelismo real en la aplicación, procesadores físicamente distribuidos, y la posibilidad de que fallen los procesadores y los elementos de proceso, implica que se requieren nuevos algoritmos para el control de los recursos. Por ejemplo, se puede precisar acceder a archivos y datos almacenados en otras máquinas; en cierta medida, un fallo en una máquina o en la red no debe comprometer la disponibilidad o la consistencia de estos archivos o datos. Además, dado que no existe una referencia de tiempo común en un sistema distribuido, y teniendo cada nodo su propia noción local de tiempo, es muy difícil obtener una visión consistente del sistema completo. Esto puede ocasionar problemas al tratar de obtener exclusión mutua sobre datos distribuidos.

- **Planificación con tiempos límite**. En el Capítulo 13, se discutieron los problemas de planificación de procesos para lograr tiempos límite con sólo un procesador. Cuando los procesos son distribuidos, los algoritmos óptimos para un procesador dejan de serlo. Se precisan nuevos algoritmos.

Estas cuestiones se discutirán en detalle. Sin embargo, no es fácil hacer justicia en un único capítulo a todas las nuevas actividades en estas áreas.

14.3 Soporte del lenguaje

El producir un sistema software distribuido que se ejecute sobre un sistema hardware distribuido implica diversas etapas que son innecesarias al trabajar con un único procesador:

- El **particionado** es el proceso de dividir el sistema en partes (unidades de distribución) adecuadas para ser situadas sobre elementos de proceso del sistema en cuestión.

- La **configuración** tendrá lugar cuando las partes en las que está particionado el programa se encuentran ya asociadas con elementos de proceso concretos del sistema en cuestión.

- La **asignación** cubre el proceso real de convertir el sistema configurado en un conjunto de módulos ejecutables, y descargar éstos sobre los elementos de procesamiento del sistema en cuestión.

- La **ejecución transparente** es la ejecución del software distribuido de modo que sea posible acceder a los recursos remotos independientemente de su ubicación.

- La **reconfiguración** es el cambio dinámico de ubicación de un componente o recurso software.

La mayoría de los lenguajes concebidos explícitamente para abordar la programación distribuida proporcionan soporte lingüístico, al menos para la etapa de particionado, en el desarrollo del sistema. Para ello, se han propuesto procesos, objetos, particiones, agentes y vigilantes como unidades de distribución. Todas estas construcciones proporcionan interfaces bien definidas que les permiten encapsular los recursos locales y proporcionar acceso remoto. Algunas aproximaciones permitirán incluir información de configuración remota en el programa fuente, mientras que otras proveerán, aparte, un lenguaje de **configuración**.

La asignación y reconfiguración requieren el apoyo del entorno de programación y del sistema operativo.

Quizás sea en el área de la ejecución transparente donde se han volcado más esfuerzos para lograr un nivel de estandarización entre las diversas aproximaciones. La meta es conseguir que la comunicación entre los procesos distribuidos sea tan fácil y robusta como sea posible. En la realidad, la comunicación suele tener lugar entre procesadores heterogéneos sobre una red poco fiable, y en la práctica se requieren complejos protocolos de comunicación (véase la Sección 14.5). Lo que se precisa son mecanismos que permitan que:

- Los procesos no tengan que tratar con la forma de bajo nivel de los mensajes. Por ejemplo, que no necesiten traducir los datos desde las series de bits apropiadas para la transmisión, o que no tengan que romper el mensaje en paquetes.

- Pueda presuponerse que todos los mensajes recibidos por los procesos usuario se encuentran intactos y en buenas condiciones. Por ejemplo, si se dividen los mensajes en paquetes, el sistema de ejecución sólo entregará el mensaje cuando hayan llegado todos los paquetes al nodo receptor y puedan ser ensamblados correctamente. Además, si los bits del mensaje han llegado en desorden, o bien no se entregará el mensaje, o bien se reconstruirá antes de su entrega; obviamente, se precisa algo de información redundante para la comprobación y corrección de los errores.

- Los mensajes que reciba cada proceso sean de la clase que éste espera. El proceso no tendrá por qué efectuar comprobaciones en ejecución.

- No haya restricciones para la comunicación entre procesos en relación con tipos predefinidos en el sistema. En vez de esto, los procesos podrán comunicarse en relación con los valores que sean de interés para la aplicación. Idealmente, si se define la aplicación usando tipos abstractos de datos, deberá ser posible comunicar los valores correspondientes en los mensajes.

Es posible señalar tres importantes estándares *de facto* para la comunicación de programas distribuidos:

- Mediante una interfaz de programación de aplicaciones (API; application programming interface), como *sockets* (conectores), para los protocolos de transporte de red.

- Mediante el paradigma de llamada a un procedimiento remoto (RPC; remote procedure call).

- Mediante el paradigma de objetos distribuidos.

En la Sección 14.5 se abordará el tema de los protocolos de red, y en la Sección 14.4.3 se discutirá brevemente un interfaz Java para *sockets*.

14.3.1 Llamada a procedimientos remotos

En último término, la meta del paradigma de llamada a procedimientos remotos es hacer la comunicación distribuida tan simple como sea posible. Usualmente, se emplea RPC para comunicar programas escritos en el mismo lenguaje (por ejemplo Ada o Java). Entre los procedimientos, uno de ellos (servidor) puede ser llamado remotamente. A partir de la especificación del servidor, es posible generar automáticamente dos procedimientos más: un **resguardo de cliente** (client stub) y un **resguardo de servidor** (server stub). El resguardo de cliente se sitúa en vez del servidor en el lugar donde se origina la llamada remota. El resguardo de servidor se ubica en el mismo lugar que el procedimiento de servicio. El cometido de estos dos procedimientos es proveer un enlace transparente entre entre el cliente y el servidor (y en consecuencia dar cuenta de todos los requisitos planteados en la sección anterior). La Figura 14.3 muestra la sucesión de eventos en un RPC entre un cliente y un servidor mediante los dos procedimientos de resguardo. A menudo, los resguardos se consideran **middleware**, dado que se asientan entre la aplicación y el sistema operativo.

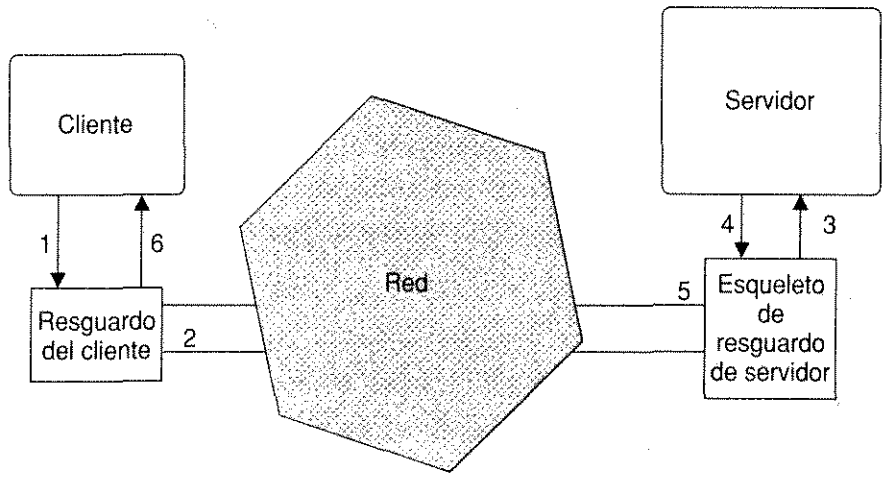


Figura 14.3. La relación entre el cliente y el servidor en un RPC.

El papel del resguardo de cliente es:

- Identificar la dirección del procedimiento (resguardo) del servidor.
- Convertir los parámetros de la llamada remota en un bloque de bytes apropiado para su transmisión a través de la red; esta actividad se suele llamar **empaquetado de parámetros** (marshalling).
- Enviar la petición de ejecución de procedimiento al servidor (resguardo).
- Esperar la respuesta del servidor (resguardo) y desempaquetar los parámetros de cualquier excepción propagada.

- Devolver el control al procedimiento cliente junto con los parámetros devueltos, o generar una excepción en el procedimiento cliente.

La tarea del resguardo de servidor es:

- Recibir peticiones de los procedimientos (resguardo) cliente.
- Desempaquetar los parámetros.
- Invocar al servidor.
- Atrapar cada excepción generada por el servidor.
- Empaquetar los parámetros devueltos (o excepciones) de forma apropiada para su transmisión sobre la red.
- Enviar la respuesta al cliente (resguardo).

Cuando los procedimientos cliente y servidor se encuentran codificados en diferentes lenguajes, o si se encuentran en máquinas de arquitecturas diferentes, los mecanismos de empaquetado y desempaquetado de parámetros convertirán los datos a un formato independiente del lenguaje y de la máquina (véase la Sección 14.4.4).

14.3.2 El modelo de objetos distribuido

El término **objetos distribuidos** (u **objetos remotos**) ha sido empleado durando los últimos años en variedad de contextos. En su sentido más amplio, el modelo de objetos distribuido permite:

- La creación dinámica de un objeto (en cualquier lenguaje) sobre una máquina remota.
- La identificación de un objeto por determinar y alojado en cualquier máquina.
- La invocación transparente de un método remoto sobre un objeto como si fuera un método local, sin importar el lenguaje en el que esté codificado el objeto.
- El reparto de ejecución (dispatch) de la llamada a un método a través de la red.

No todos los sistemas que permiten objetos distribuidos proporcionan mecanismos que aporten toda esta funcionalidad. Como se podrá ver en las siguientes subsecciones:

Ada permite la reserva estática de objetos y la identificación de objetos remotos Ada, facilita la ejecución transparente de métodos remotos, y soporta el despacho de ejecución distribuida de métodos remotos.

Java permite enviar el código de un objeto Java a través de la red y la creación de instancias remotamente, nombrar un objeto Java remoto, invocar sus métodos de modo transparente, y el despacho de ejecución distribuida.

CORBA permite crear objetos en diferentes lenguajes sobre máquinas diferentes, facilita la ejecución transparente de métodos remotos, y soporta el despacho de ejecución distribuida de métodos remotos.

14.4 Sistemas y entornos de programación distribuida

El ámbito de las aplicaciones distribuidas es enorme: desde un simple sistema de control embebido a una compleja plataforma genérica de procesamiento de información multilenguaje. Queda fuera del alcance de este libro el discutir exhaustivamente cómo se pueden diseñar e implementar estos sistemas. Sin embargo, se tendrán en cuenta cuatro aproximaciones:

(1) occam2: para sistemas simples de control embebido.

(2) Ada: para aplicaciones distribuidas de tiempo real más complejas.

(3) Java: para aplicaciones tipo Internet de un único lenguaje.

(4) CORBA: para aplicaciones multiplataforma y multilenguaje.

14.4.1 Occam2

Occam2 ha sido diseñado específicamente para que sus programas puedan ser ejecutados en un entorno distribuido, como el de una red de varios transputers. En general, los procesos de occam2 no comparten variables, de modo que la unidad de particionado es el mismo proceso. La configuración se obtiene mediante la construcción `PLACED PAR`. Un programa construido como un `PAR` en su nivel más alto, tal como:

```
PAR
  p1
  p2
  p3
  p4
  p5
```

podría distribuirse como sigue:

```
PLACED PAR
  PROCESSOR 1
    p1
  PROCESSOR 2
    PAR
      p2
```

```
      p3
  PROCESSOR 3
    PAR
      p4
      p5
```

Es importante darse cuenta de que la transformación del programa de tener un simple PAR a emplear PLACED PAR no invalidará el programa. Sin embargo, occam2 permite la lectura de variables por más de un proceso del mismo procesador. De modo que la transformación no será posible si el programa ha utilizado esta posibilidad.

En los transputers, también es preciso asociar cada canal externo con un enlace (link) de transputer apropiado. Esto se hace mediante la construcción PLACE AT. Por ejemplo, considere el ejemplo de más adelante con los siguientes canales enteros mostrados en la Figura 14.4.

El programa para su ejecución en un único transputer es:

```
CHAN OF INT c1, c2, c3, c4, c5:
PAR
  p1
  p2
  p3
  p4
  p5
```

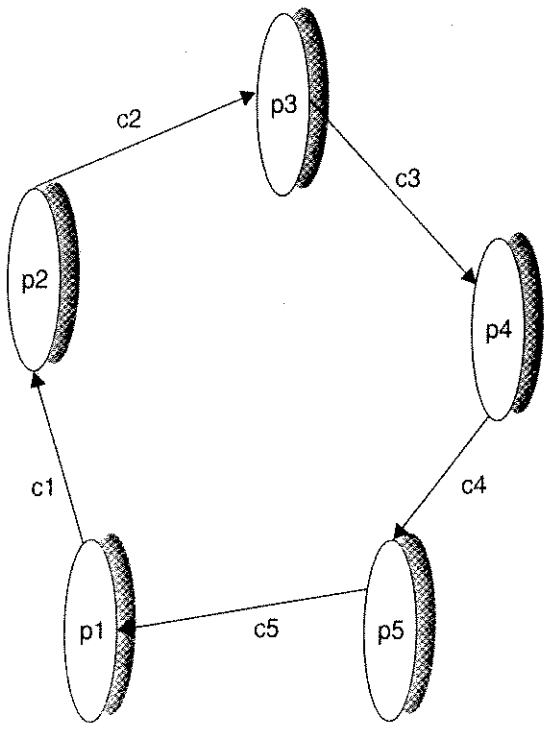


Figura 14.4. Cinco procesos occam2 conectados por cinco canales.

Si se reconfigura el programa para tres transputers como se muestra en la Figura 14.5, el programa occam2 resultaría:

```
CHAN OF INT c1, c3, c5:
PLACED PAR
  PROCESSOR 1
    PLACE c1 at 0:
    PLACE c5 at 1:
    p1
  PROCESSOR
    PLACE c1 at 2:
    PLACE c3 at 1:
    CHAN OF INT c2:
    PAR
      p2
      p3
  PROCESSOR 3
    PLACE c3 at 0:
    PLACE c5 at 2:
    CHAN OF INT c4:
    PAR
      p4
      p5
```

La facilidad con la que pueden configurarse los programas occam2 para su ejecución en un sistema distribuido es uno de sus principales atractivos.

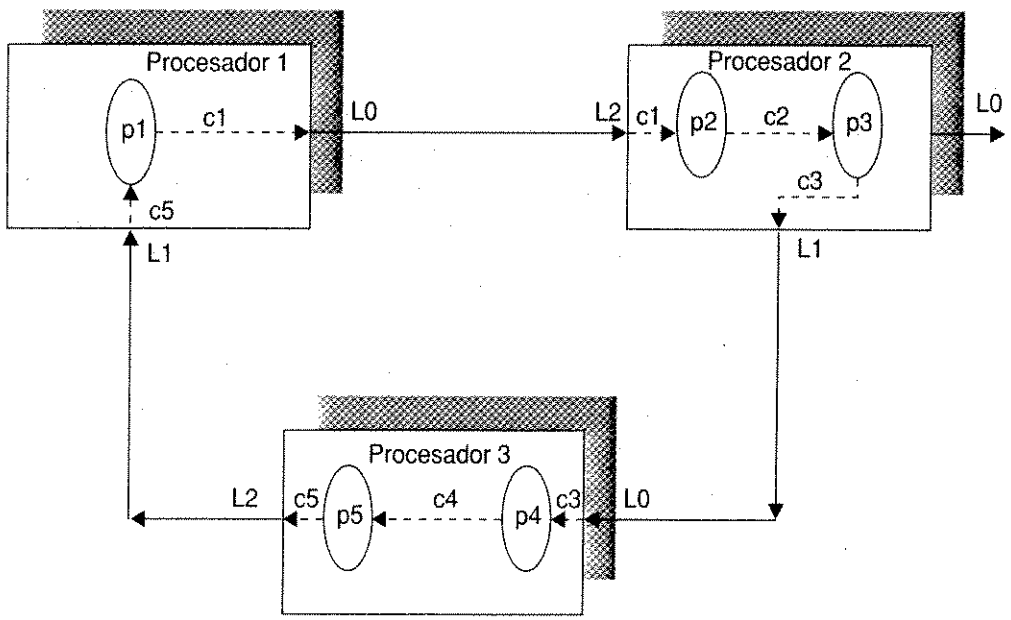


Figura 14.5. Cinco procesos occam2 configurados para tres transputers.

La asignación no se define en el lenguaje occam2, y tampoco se prestan posibilidades para la reconfiguración. Además, el acceso a los recursos no es transparente.

Perspectiva de tiempo real

El soporte de occam2 para tiempo real es limitado. Sin embargo, dentro de sus limitaciones, el modelo de sistema distribuido es consistente.

14.4.2 Ada

Ada define un sistema distribuido como:

> ... una interconexión de uno o más nodos de procesamiento (recurso del sistema que tiene capacidad de cómputo y de almacenamiento), y cero o más nodos de almacenamiento (recurso del sistema que sólo tiene capacidad de almacenamiento, y cuyo almacén es direccionable por más de un nodo de procesamiento).

El modelo para la programación de sistemas distribuidos de Ada especifica la **partición** como la unidad de distribución. Las particiones comprenden agregaciones de unidades de biblioteca (library units) (paquetes de biblioteca o subprogramas compilados separadamente), que pueden ser ejecutadas colectivamente en un entorno de ejecución distribuido objetivo. La configuración de la unidades de biblioteca en particiones no está definida en el lenguaje; se presupone que cada implementación la proporcionará, además de mecanismos para la asignación y, si es necesario, la reconfiguración.

Cada partición reside en un único lugar de ejecución, donde todas sus unidades de biblioteca ocupan el mismo espacio lógico de direcciones. Sin embargo, en el mismo lugar de ejecución puede convivir más de una partición. La Figura 14.6 muestra una posible estructura de partición. Las flechas representan las dependencias entre las unidades de biblioteca. La principal interfaz entre las particiones consiste en una o más especificaciones de paquete (cada una etiquetada como «unidad de biblioteca de interfaz de partición» en la Figura 14.6).

Las particiones pueden ser **activas** o **pasivas**. Las unidades de biblioteca que comprenden una partición activa residen y se ejecutan sobre el mismo elemento de procesamiento. Por contra, las unidades de biblioteca que constan de una partición pasiva residen en un elemento de almacenamiento que será accesible directamente para los nodos de las diferentes particiones activas que las referencian. Este modelo asegura que las particiones activas no puedan acceder directamente a variables de otras particiones activas. Las variables sólo pueden compartirse directamente entre particiones activas si se encapsulan en una partición pasiva. La comunicación entre particiones activas se define en el lenguaje para que se realice mediante llamadas a subprogramas remotos (aunque cada implementación puede proporcionar otros mecanismos de comunicación).

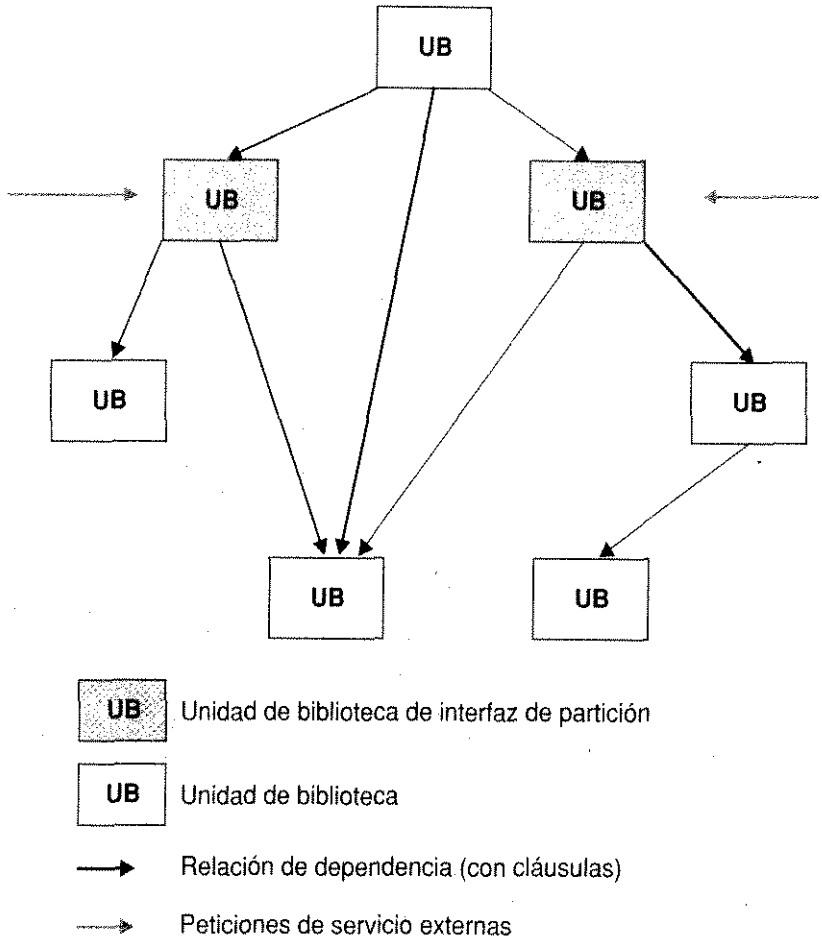


Figura 14.6. La estructura de una partición.

Pragmas de categorización

Para ayudar a la construcción de programas distribuidos, Ada distingue entre diferentes categorías de unidades de biblioteca, e impone restricciones sobre estas categorías para mantener la consistencia de tipo a través del programa distribuido. Las categorías (algunas de las cuales son útiles por derecho propio, sin importar si el programa va a ser distribuido) vienen designadas por los siguientes pragmas:

- `Preelaborate`

  Una unidad de librería preelaborada (`Preelaborate`) es aquélla que no puede ser elaborada sin la ejecución de código en tiempo de ejecución.

- `Pure`

  Los paquetes puros (`Pure`) son paquetes preelaborados con restricciones adicionales que les permiten ser replicados libremente en diferentes particiones activas o pasivas sin introducir ninguna inconsistencia de tipo. Estas restricciones conciernen a la declaración de objetos y tipos; en concreto, no se permiten variables y tipos de acceso por nombre, a menos que sean internos a un subprograma, unidad de tarea o unidad protegida.

- `Remote_Types`

  Un paquete de tipos remotos (`Remote_Types`) es un paquete preelaborado que no debe contener declaración de variable alguna en su parte visible.

- `Shared_Passive`

  Las unidades de biblioteca compartidas pasivas (`Shared_Passive`) se emplean para gestionar los datos globales compartidos entre las particiones activas. Se configuran en el sistema distribuido, además, sobre nodos de almacenamiento.

- `Remote_Call_Interface`

  Un paquete de interfaz de llamada remota (`Remote_Call_Interface`) define la interfaz entre particiones activas. Su cuerpo existe dentro de una sola partición. Todas sus otras apariciones vendrán dadas por resguardos.

  La especificación de un paquete `Remote_Call_Interface` debe ser preelaborada; además posee otras restricciones; por ejemplo, no debe contener definiciones de variables (para asegurar que no se produce ningún acceso remoto al dato).

Un paquete no categorizado por ningún pragma de categorización se dice que es un paquete de biblioteca *normal*. Si se encuentra incluido en más de una partición, se encontrará replicado, y todos los tipos y objetos se consideran distintos. Por ejemplo, el paquete `Calendar` es, según esto, normal.

Los pragmas anteriores facilitan la distribución de un programa Ada y garantizan la fácil identificación de las particiones ilegales (aquéllas que permiten al acceso remoto directo a variables entre particiones).

Comunicación remota

La única forma predefinida en que pueden comunicarse directamente dos particiones activas es mediante llamadas a subprogramas remotos. También podrán comunicarse indirectamente mediante estructuras de datos en particiones pasivas.

Existen tres formas en que una partición invocadora puede emitir una llamada a un subprograma remoto:

- Llamando a un subprograma que haya sido declarado directamente en un paquete de interfaz de llamada remota de otra partición.
- «Desreferenciando» un puntero a un subprograma remoto.
- Despachando, en tiempo de ejecución, un método de un objeto remoto.

Es importante observar que en el primer tipo de comunicación, las particiones invocadora e invocada se encuentran vinculadas estáticamente en la compilación. Sin embargo, en las dos últimas, las particiones se vinculan dinámicamente en tiempo de ejecución. Por ello, Ada puede soportar un acceso transparente a los recursos.

Programa 14.1. El paquete Ada `System.RPC`.

```
with Ada.Streams;
package System.RPC is

   type Partition_ID is range 0 .. implementation defined;

   Communication_Error : exception;

   type Params_Stream_Type ...

   -- Invocación síncrona
   procedure Do_RPC(
           Particion : in Partition_ID;
           Params    : access Params_Stream_Type;
           Resultado : access Params_Stream_Type);

   -- Invocación asíncrona
   procedure Do_APC(
           Particion : in Partition_ID;
           Params    : access Params_Stream_Type);

   -- El manejador para los RPC entrantes
   type RPC_Receiver is access procedure(
           Params    : access Params_Stream_Type;
           Resultado : access Params_Stream_Type);

   procedure Establish_RPC_Receiver(Partition : Partition_ID;
           Receptor  : in RPC_Receiver);

private
   ...
end System.RPC;
```

Muchas llamadas remotas contienen sólo parámetros «in» (entrada) o «access» (acceso), esto es, datos que se pasan en la misma dirección que la invocación; un invocador puede querer continuar su ejecución tan pronto como sea posible. En estas situaciones suele ser apropiado designar la llamada como *asíncrona*. Tanto si un procedimiento se invoca síncrona como asíncronamente, Ada lo considera como una propiedad del procedimiento, y no de la invocación. Esto se indica mediante el pragma `Asynchronous` al declarar el procedimiento.

Ada define cómo pueden particionarse los programas distribuidos y qué formas de comunicación deben soportarse. Sin embargo, los diseñadores del lenguaje tuvieron cuidado de no sobreespecificar el lenguaje y de no prescribir un sistema de soporte de ejecución para los programas Ada. Se trataba de permitir que los implementadores proporcionaran sus propios protocolos de

comunicación en red y, cuando fuera apropiado, la incorporación de otros estándares ISO –por ejemplo el estándar ISO de llamada a procedimiento remoto (ISO Remote Procedure Call)–. Para cumplir estos objetivos, el lenguaje Ada asume la existencia de un subsistema estándar proporcionado por la implementación para manejar toda la comunicación remota: el subsistema de comunicación de partición (Partition Communication Subsystem; PCS). Esto permite que los compiladores generen llamadas a una interfaz estándar sin preocuparse de la implementación subyacente.

El paquete definido en el Programa 14.1 muestra la interfaz al sistema de soporte de llamada a procedimiento (subprograma) remoto (RPC) que forma parte del PCS.

El tipo `Partition_Id` se emplea para identificar las particiones. Para cualquier declaración del nivel de biblioteca (`D`), `D'Partition_Id` aloja el identificador de la partición en la que fue elaborada la declaración. La excepción `Communication_Error` se genera cuando se detecta un error desde `System.RPC` durante una llamada a un procedimiento remoto. Para empaquetar (traducir datos en una forma apropiada orientada a stream) y desempaquetar los parámetros o los resultados de una llamada a un subprograma remoto, con el fin de enviarlos entre las particiones, se emplea un objeto de tipo stream: `Params_Stream_Type`. El objeto también se usa para identificar el subprograma concreto en la partición invocada.

El procedimiento `Do_RPC` se invoca desde el resguardo de llamada después de que los parámetros hayan sido embutidos en el mensaje. Tras enviar el mensaje a la partición remota, se suspende la tarea invocadora hasta que llega una respuesta. El procedimiento `Do_APC` actúa como `Do_RPC`, excepto que vuelve inmediatamente después de enviar el mensaje a la partición remota. Se utiliza siempre que el procedimiento llamado remotamente haya sido especificado con el pragma `Asynchronous`.

Inmediatamente después de elaborar una partición activa, se invoca a `Establish_RPC_Receiver`, aunque antes de invocar al subprograma principal. El parámetro `Receptor` designa un procedimiento proporcionado por la implementación, que recibe un mensaje e invoca al subprograma del paquete de interfaz apropiado.

Perspectiva de tiempo real

Aunque Ada define un modelo de tiempo real coherente para sistemas mono y multiprocesador, dispone de un soporte muy limitado para sistemas *distribuidos de tiempo real*. No hay integración alguna entre los anexos para sistemas distribuidos y para sistemas de tiempo real (*Distributed Systems Annex* y *Real-Time Annex*). Posiblemente, la tecnología de soporte para el lenguaje en esta área no se encuentra lo suficientemente aceptada como para merecer una estandarización.

14.4.3 Java

Esencialmente, hay dos formas de construir aplicaciones distribuidas en Java:

(1) Ejecutando programas Java en máquinas separadas y empleando mecanismos de comunicación en red.

(2) Empleando objetos remotos.

Comunicación en red con Java

En la Sección 14.5, se presentaron dos protocolos de comunicación: UDP y TCP. Éstos son los principales protocolos de comunicación empleados hoy en día, y el entorno Java proporciona clases (en el paquete `java.net`) que facilitan el acceso a ellos. El API de estos protocolos se efectúa mediante la clase `Socket` (para el protocolo TCP fiable) y la clase `DatagramSocket` (para el protocolo UDP). Se encuentra fuera del alcance de este libro el considerar esta aproximación con detalle, y se remite al lector a la sección de «Lecturas complementarias» del final de este capítulo para otras fuentes de información.

Objetos remotos

Aunque Java proporciona un acceso conveniente a los protocolos de red, estos protocolos son aún complejos, y tienden a disuadir de la programación de aplicaciones distribuidas. Consecuentemente, Java soporta el modelo de comunicación de objetos distribuidos mediante la noción de **objetos remotos**.

El modelo Java se centra en el uso del paquete `java.rmi`, que opera sobre el protocolo TCP. En este paquete se encuentra la interfaz `Remote`:

```
public interface Remote { };
```

Éste es el punto de arranque para la escritura de aplicaciones Java distribuidas. Extendiendo esta interfaz se logra la integración entre clientes y servidores. Considere, por ejemplo, un servidor que quiera devolver detalles de la predicción meteorológica de su localidad. Una interfaz apropiada podría ser:

```
public interface PrediccionMeteorologica extends java.rmi.Remote
  // compartida entre los clientes y el servidor
{
  public Prediccion damePrediccion() throws RemoteException;
}
```

El método `damePrediccion` debe contemplar la clase `RemoteException` en su lista de objetos lanzables para que la implementación subyacente pueda indicar si ha fallado la llamada remota.

`Prediccion` es un objeto que contiene detalles de la predicción para hoy. Como el objeto puede ser copiado a través de la red, debe implementar la interfaz `Serializable`:[2]

[2] Igual que la interfaz `Remote`, la interfaz `Serializable` está vacía. En ambos casos actúa como un etiquetado que presenta información al compilador. Para la interfaz `Serializable`, indica que el objeto puede ser convertido en un flujo de bytes adecuado para E/S.

```
public class Prediccion implements java.io.Serializable
{
  public String Hoy() {
     String hoy = "Húmedo";
     return hoy ;
  }
}
```

Una vez definida una interfaz remota apropiada, podrá declararse una clase servidora. De nuevo, es norma indicar que los objetos de la clase pueden ser llamados remotamente mediante la extensión de una de las clases predefinidas del paquete `java.rmi.server`. Actualmente, hay dos clases: `RemoteServer` (que es una clase abstracta derivada de la clase `Remote`), y `UnicastRemoteObject`, que es una extensión concreta de `RemoteServer`. La segunda proporciona la clase para servidores no replicados, y dispone de una conexión punto a punto con cada cliente mediante el protocolo TCP. Se prevé que futuras extensiones de Java proporcionen otras clases, como un servidor replicado mediante un protocolo de comunicación por multidifusión (multicast).

Los siguientes ejemplos muestran una clase servidora que proporciona la predicción meteorológica para el condado de Yorkshire, en el Reino Unido.

```
public class PrediccionMeteorologicaYorkshire extends UnicastRemoteObject
                    implements PrediccionMeteorologica
{
  public PrediccionMeteorologicaYorkshire() throws RemoteException
  {
     super(); // llamada al constructor padre
  }

  public Prediccion damePrediccion() throws RemoteException
  {
     ...
  }
}
```

Una vez que ha sido escrita la clase del servicio, es preciso generar los resguardos del servidor y del cliente para cada uno de los métodos que puedan ser llamados remotamente. Obsérvese que Java utiliza el término «esqueleto» (**skeleton**) para el resguardo del servidor. El entorno de programación Java proporciona una herramienta llamada «rmic», que toma una clase de servicio y genera automáticamente el resguardo de cliente y el esqueleto del servidor.

Todo lo que se pide ahora es que el cliente pueda adquirir un resguardo de cliente que acceda al objeto de servicio. Esto se obtiene mediante un registro. El registro es un programa Java aparte que se ejecuta en cada máquina huésped que aloja objetos de servicio; escucha en un puerto TCP estándar, y proporciona un objeto de la clase `Naming`. En el Programa 14.2 se incluye un extracto de esta clase.

Programa 14.2. Un extracto de la clase `Naming` de Java.

```
public final class Naming
{
  public static void bind(String nombre, Remote obj)
     throws AlreadyBoundException, java.net.MalformedURLException,
            UnknownHostException, RemoteException;
  // vincula el nombre al objeto
  // el nombre toma la forma de un URL tal como
  //            rmi://hostRemoto:puerto/nombreObjeto

  public static Remote lookup(String nombre)
     throws NotBoundException, java.net.MalformedURLException,
            UnknownHostException, RemoteException;
  // busca el nombre en el registro y devuelve un objeto remoto
  ...
}
```

Cada objeto de servicio puede usar la clase `Naming` para vincular su objeto remoto con un nombre. Los clientes pueden acceder entonces a un registro remoto para adquirir una referencia al objeto remoto. Obviamente, ésta es una referencia a un resguardo de cliente del objeto del servidor. El resguardo de cliente se encuentra alojado en la máquina del cliente. Una vez obtenida la referencia, podrán invocarse los métodos del servidor.

Perspectiva de tiempo real

Las posibilidades descritas son las de Java estándar, que no está diseñado para operar dentro de las restricciones de tiempo real. Java para tiempo real no dice nada, actualmente, sobre programación de sistemas de tiempo real distribuidos. Sin embargo, éste es un tema que será abordado dentro de los próximos años.

14.4.4 CORBA

La arquitectura común de intermediación de peticiones de objetos (CORBA; Common Object Request Broker Architecture) proporciona el modelo de objetos distribuido más general. Su objetivo es facilitar la interoperabilidad entre aplicaciones escritas en diferentes lenguajes, para diferentes plataformas mediante middleware de diferentes fabricantes. Fue diseñado por el Object Management Group (OMG), un consorcio de fabricantes de software, desarrolladores y usuarios finales, según el modelo arquitectónico de gestión de objetos (OMA; Object Management Architectural Model), el cual se muestra en la Figura 14.7.

En el núcleo de la arquitectura se encuentra el **intermediario de peticiones de objetos** (ORB; Object Request Broker). Éste es un bus software de comunicación que proporciona la infraes-

tructura principal que facilita la interoperabilidad entre aplicaciones heterogéneas (suele usarse el término CORBA para referirse al ORB). Los otros componentes de la arquitectura son:

Servicios de objetos (Object Services): un conjunto de servicios básicos que dan apoyo al ORB; por ejemplo, soporte para la creación de objetos, la nominación y el control de acceso, y el seguimiento de objetos reubicados.

Facilidades comunes (Common Facilities): un conjunto de funciones de uso común a lo largo de un amplio rango de dominios de aplicación; por ejemplo, interfaces de usuario, gestión de documentos y bases de datos.

Interfaces de dominio (Domain Interfaces): un grupo de interfaces que dan apoyo a dominios de aplicación concretos, como banca y finanzas o telecomunicaciones.

Interfaces de aplicaciones (Application Interfaces): las interfaces específicas de las aplicaciones de los usuarios finales.

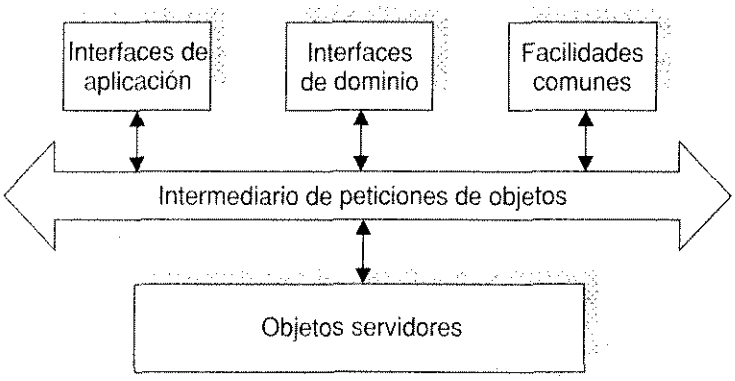


Figura 14.7. El modelo arquitectónico de gestión de objetos OMA.

Para garantizar la interoperatividad entre los ORB de los diferentes fabricantes, CORBA define un protocolo general inter-ORB (GIOP; General Inter-Orb Protocol) que se asienta sobre TCP/IP (véase la Sección 14.5.2).

El lenguaje de definición de interfaces (IDL; Interface Definition Language) es fundamental a la hora de escribir cualquier aplicación CORBA. Una interfaz CORBA es similar en concepto a la interfaz remota Java, que ya se discutió en la sección anterior. El IDL (que se parece a C++) se emplea para describir los mecanismos proveídos por un objeto de aplicación, los parámetros que se le pasan a cada método y sus valores devueltos, así como cualquier atributo de un objeto. A continuación se muestra una interfaz para la aplicación de predicción meteorológica dada en la anterior sección.

```
interface PrediccionMeteorologica {
  void DamePrediccion(out Prediccion hoy);
}
```

Una vez empleado el IDL, se requieren ciertas herramientas para «compilarlo». El compilador de IDL genera varios archivos nuevos en alguno de los lenguajes de programación existentes, como Java o Ada. Estos archivos incluyen:

- Resguardos de cliente, que proporcionan un canal de comunicación entre el cliente y el ORB.
- Un esqueleto de servidor, que permite al ORB llamar a las funciones del servidor.

Ahora es cuando se puede escribir el código de los clientes y los servidores. El servidor consta de dos partes: el código para el objeto de la aplicación en sí, y el código para el proceso servidor. El objeto de la aplicación debe enlazarse con el esqueleto del servidor, y la forma de lograr esto depende del lenguaje elegido. En Java, por ejemplo, se hace produciendo una clase que es una subclase del esqueleto generado para el servidor. Entonces se completan los métodos para los objetos de la aplicación. El proceso servidor en sí puede escribirse como un programa principal que crea el objeto de aplicación e inicializa el ORB, indicándole que el objeto en cuestión está listo para recibir peticiones del cliente. La estructura del cliente es similar.

En realidad, CORBA proporciona muchas más posibilidades de las descritas. Así como la asociación estática entre el cliente y el servidor, los clientes CORBA pueden descubrir dinámicamente una interfaz IDL de un servidor sin conocer previamente los detalles del servidor. Además, los servicios que dan apoyo al ORB permiten un amplio rango de funciones, como servicio de eventos, transacciones y control de concurrencia, objetos persistentes y servicios de intercambio. Las aplicaciones acceden a estos servicios mediante el adaptador de objetos portables (POA; Portable Object Adapter). Se trata de una biblioteca que provee el entorno de ejecución para un objeto servidor.

Perspectiva de tiempo real

Aunque, en general, CORBA proporciona un soporte completo para los objetos distribuidos, desde la perspectiva de este libro una de las principales limitaciones de CORBA es su carencia de soporte a las aplicaciones de tiempo real. Los ORB no fueron diseñados para operar dentro de restricciones de tiempo, y el resultado es que muchos de ellos son inapropiados para sistemas de tiempo real flexibles, por no hablar de los sistemas que operan con restricciones de tiempo rígidas. Además, los modelos básicos de comunicación proporcionados por CORBA son: un RPC síncrono, donde el cliente debe esperar la respuesta del servidor, y un RPC síncrono diferido, donde el hilo del cliente continúa y encuesta posteriormente por la respuesta. Según se mostró enel Capítulo 13, los modelos de comunicación síncronos son más complicados de analizar por sus propiedades de temporización que los modelos asíncronos, y los resultados son más pesimistas.

Por estas razones, y por el creciente uso de CORBA en las aplicaciones de tiempo real, OMG ha tomado últimamente la decisión de abordar los problemas de prestaciones asociados con el principal estándar CORBA. Se han acometido los problemas desde tres perspectivas:

(1) CORBA mínimo (Minimum CORBA).

(2) Mensajería CORBA (CORBA Messaging).

(3) CORBA de tiempo real (Real-Time CORBA).

Minimum CORBA es un subconjunto de la especificación CORBA 2.2 que omite muchos servicios, incluyendo aquéllos asociados con la configuración dinámica entre objetos clientes y servidores. El subconjunto está enfocado a los sistemas embebidos con recursos limitados.

CORBA Messaging es una extensión al estándard CORBA para dar soporte a el paso asíncrono de mensajes y parámetros de calidad de servicio (QoS).

La especificación Real-Time CORBA (RT CORBA) (Object Management Group, 1999) define mecanismos que soportan la predecibilidad de las aplicaciones CORBA distribuidas. Se supone que el comportamiento de tiempo real se obtiene empleando un planificador basado en prioridades fijas, compatible con la mayoría de los sistemas operativos de tiempo real, especialmente con aquéllos que soportan las interfaces de tiempo real de POSIX. El aspecto clave de RT CORBA es que permite que las aplicaciones configuren y controlen tanto los recursos de procesamiento como los de de comunicación. Estas posibilidades se describen brevemente a continuación.

Gestión de los recursos de procesamiento

La especificación RT CORBA permite a las aplicaciones cliente y servidor gestionar las siguientes propiedades:

- **La prioridad con que los servidores procesan las peticiones de los clientes**. RT CORBA especifica prioridades CORBA y mecanismos globales para relacionar dichas prioridades con el rango de prioridades de cada sistema operativo de tiempo real concreto que aloja el RT ORB. Al usar esta correspondencia, se proporcionan tres modelos de prioridad: (1) un modelo declarado por el servidor, en el que es éste quien especifica las prioridades para las peticiones a sus servicios; (2) un modelo propagado por el cliente, en el que éste especifica la prioridad, la cual es propagada al servidor; y (3) un modelo de transformación, en el que la prioridad de cada petición se basa en factores externos, tales como la carga actual del servidor o el estado del servicio de planificación global.

- **El grado de multienhebrado en el servidor**. RT CORBA controla el grado de multienhebrado mediante el concepto de depósitos de hilos (hebras). Usando un POA de tiempo real, las aplicaciones de servicio pueden especificar: el número de hilos por defecto que se crean inicialmente, el número máximo de hilos que pueden crearse dinámicamente, y la prioridad por defecto de todos los hilos. Estos hilos se asignan de un depósito de hilos, que puede ser o no compartido entre las aplicaciones. Se consigue más flexibilidad aún mediante el concepto de depósitos de hilos con carriles (lanes). Aquí, no sólo se puede controlar el grado de concurrencia global, sino también la cantidad de trabajo efectuado a cierta prioridad.

- **El impacto de la herencia de prioridad y los protocolos de acotación de la prioridad**. RT CORBA define mutexes al estilo POSIX para garantizar la consistencia de los protocolos de sincronización mediante el encapsulamiento de los recursos compartidos.

Gestión de recursos de red

Aunque la posibilidad de tener un acceso transparente a los recursos permite la escritura de aplicaciones distribuidas de propósito general, hace imposible realizar cualquier análisis de prestaciones de tiempo real realista. Por esta razón, RT CORBA permite establecer conexiones explícitas entre clientes y servidores, por ejemplo, en tiempo de configuración. También es posible controlar cómo se envían las peticiones de los clientes por estas conexiones. RT CORBA facilita múltiples conexiones entre clientes y servidores para reducir los problemas de inversión de prioridad debidos al uso de protocolos de transporte que no son de tiempo real. Además, se permiten conexiones de transporte privadas. Así, podrá hacerse una invocación de un cliente a un servidor sin miedo a que la llamada tenga que competir con otras invocaciones que pudieran multiplexarse sobre la misma conexión.

Incluso con las posibilidades anteriores, sigue siendo ventajoso el poder fijar cualquier parámetro de calidad de servicio de los protocolos de comunicación específicos que subyacen al protocolo de comunicación inter-ORB. RT CORBA proporciona interfaces para seleccionar y configurar estas propiedades por parte del cliente y del servidor.

El servicio de planificación

De la discusión anterior, debería haber quedado claro que RT CORBA proporciona muchos de los mecanismos que facilitan la construcción de aplicaciones distribuidas CORBA. Sin embargo, fijar todos los parámetros requeridos (para poder obtener una política de planificación consistente) puede ser difícil. Por esta razón, RT CORBA permite que una aplicación especifique sus requisitos de planificación en cuanto a características tales como su periodo, su tiempo de ejecución en el peor caso, su criticidad, etc. Esto se efectúa fuera de línea, y a cada entidad de aplicación planificada (denominada *actividad*) se le asigna un nombre textual. En el momento de la ejecución, se proporcionan interfaces que permiten a una aplicación planificar una actividad nombrada mediante un *servicio de planificación*. Este servicio establece todos los parámetros de prioridad necesarios para implementar una política de planificación específica, como la de tiempo límite o la de tasa monotónica.

14.5 Fiabilidad

Resulta paradójico que la distribución que proporciona los medios para construir sistemas más fiables introduzca, al mismo tiempo, más fallos potenciales en el sistema. Aunque la disponibilidad de varios procesadores permita que la aplicación sea más tolerante a fallos de procesador, también presenta la posibilidad de que aparezcan fallos inexistentes en un sistema centralizado monoprocesador. En concreto, con varios procesadores aparece el concepto de fallo parcial del sistema. En un sistema de un único procesador, si falla la memoria del procesador fallará el sistema completo (a veces el procesador puede ser capaz de continuar y recuperarse de un fallo parcial de memoria, pero por lo general el sistema caerá). Sin embargo, en un sistema distribuido, es posible que falle un procesador mientras el resto continúa operando. Además, el retraso en la propagación por la red de comunicación subyacente varía, y los mensajes pueden seguir caminos

diferentes. Esto, junto con un medio de transmisión no fiable, puede ocasionar la pérdida de mensajes, la corrupción, y la entrega en desorden. La creciente complejidad de un software preciso que tolere tales fallos, puede ser también una amenaza para la fiabilidad del sistema.

14.5.1 Sistemas de interconexión abiertos

Se ha invertido mucho esfuerzo en protocolos de comunicación para redes y sistemas distribuidos. Tratar de ellos con detalle queda fuera del alcance de este libro; remitimos al lector a la sección «Lecturas complementarias» del final de este capítulo. En general, los protocolos de comunicación están estratificados para facilitar su diseño e implementación. Sin embargo, existen muchos tipos de redes diferentes, cada una con su propio concepto de «arquitectura de red» y de protocolos de comunicación asociados. Consecuentemente, sin algún estándar internacional se hace extremadamente difícil el intentar interconectar sistemas de diferentes características. La organización ISO (International Organization for Standarization) ha definido varios estándares que abarcan al concepto de *interconexión de sistemas abiertos*, o extensibles (OSI; Open Systems Interconnections). El término «abierto» se emplea para indicar que, al seguir estos estándares, el sistema estará abierto a todos los otros sistemas del mundo que respeten los mismos estándares. Estos estándares son conocidos como el *modelo de referencia OSI* (OSI Reference Model). Hay que recalcar que este modelo no se refiere a aplicaciones específicas de las redes de computadores, sino a la *estructura* de los protocolos de comunicación requeridos para proporcionar un servidio de comunicación fiable e independiente del fabricante. El modelo de referencia OSI es un modelo estratificado. Sus niveles se muestran en la Figura 14.8.

La idea básica de la estratificación es que cada nivel se asiente sobre los servicios proporcionados por los niveles inferiores, para así presentar un servicio a los niveles superiores. Desde un nivel concreto, los niveles inferiores deben considerarse como una caja negra que implementa un servicio. El medio a través del cual un nivel accede a los servicios proporcionados por los niveles inferiores es la interfaz de esos niveles. Cada interfaz define las reglas y el formato de la información intercambiada entre los límites de los niveles adyacentes. Los módulos que implementan cada capa son conocidos habitualmente como **entidades**.

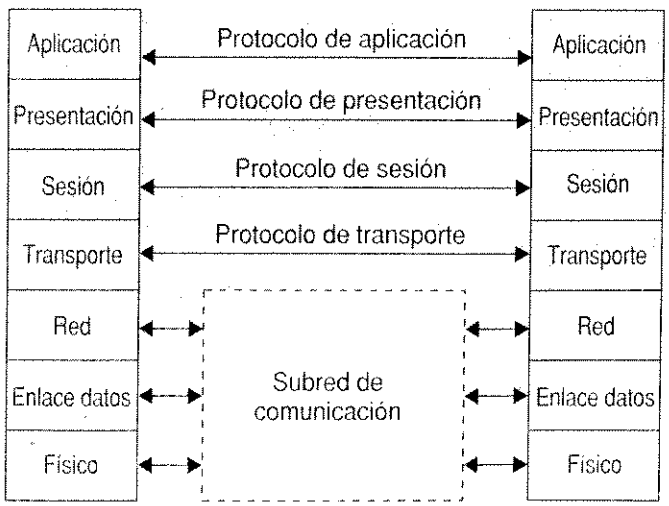


Figura 14.8. El modelo de referencia OSI.

En redes y sistemas distribuidos, cada capa puede estar distribuida en más de una máquina; para proporcionar su servicio, las entidades del mismo nivel de diferentes máquinas pueden necesitar intercambiar información. Tales entidades se conocen como **entidades pares**.Un **protocolo** es el conjunto de reglas que gobiernan la comunicación entre entidades pares.

El modelo OSI en sí mismo no define estándares de protocolos; mediante la partición de las funciones de la red en niveles, sugiere dónde deberían desarrollarse los protocolos estándar, aunque dichos estándares caen fuera del modelo en sí mismo. Como es lógico, han aparecido desarrollos basados en dichos estándares.

A continuación, se describe brevemente cada nivel.

(1) **El nivel físico**

El nivel físico tiene que ver con la transmisión de los datos en bruto por el canal de comunicación. Su cometido es asegurar que, en ausencia de errores, cuando un lado envía un bit 1, éste se recibe como un bit 1, y no como un bit 0.

(2) **El nivel de enlace de datos**

El nivel de enlace de datos convierte un canal de transmisión potencialmente no fiable en fiable para su uso por el nivel de red. También es responsable de la resolución de cualquier disputa por el acceso al canal de transmisión entre nodos conectados al canal.

Para que el nivel de enlace de datos pueda proporcionar un canal de comunicación fiable, debe ser capaz de corregir errores. Existen dos técnicas básicas y ya familiares: *control de errores hacia delante*, y *control de errores hacia atrás*. El control de errores hacia delante precisa suficiente información redundante en cada mensaje como para corregir cualquier error que apareciera en la transmisión. En general, la cantidad de redundancia requerida se incrementa rápidamente según se incrementa el número de bits de información. El control de errores hacia atrás sólo requiere la detección del error; una vez detectado, se puede emplear un esquema de retransmisión para obtener el mensaje correcto (ésta es la tarea del nivel de enlace de datos). En el mundo de las redes y los sistemas distribuidos, predomina el control de errores hacia atrás.

La mayoría de las técnicas de control de errores hacia atrás incorporan cierto elemento llamado *suma de comprobación* (checksum), que se envía junto al mensaje y resume el contenido del mismo. En la recepción, se recalcula la suma de comprobación y se compara con la recibida. Cualquier discordancia indica que se ha producido un error en la transmisión. En este punto, el nivel de enlace de datos puede ofrecer tres clases básicas de servicio: un servicio sin conexión y sin reconocimiento, un servicio sin conexión y con reconocimiento, o un servicio orientado a conexión. Con el primero, no se proveen más operaciones. Para el emisor pasa inadvertido el hecho de que el mensaje haya sido recibido intacto o no. Con el segundo, se informa al emisor cada vez que se recibe un mensaje correcto; la ausencia de reconocimiento de mensaje en un intervalo de tiempo indica la presencia de un error. El tercer tipo de servicio establece una conexión entre el emisor y el receptor, y garantiza que todos los mensajes se reciben correctamente y en orden.

(3) **El nivel de red**

El nivel de red (o nivel de comunicación de red) se ocupa de cómo encaminar (rutar) la información desde el nivel de transporte hacia su destino a través de la red de comunicación subyacente. Los mensajes se parten en paquetes, que pueden ser encaminados por diferentes vías; el nivel de red debe reensamblar los paquetes y habérselas con cualquier congestión que pudiera ocurrir. No hay un acuerdo claro sobre si el nivel de red debiera intentar proporcionar un canal de comunicación de red perfecto. Se pueden indentificar dos extremos en los servicios: *circuitos virtuales* (orientado a conexión) y *datagramas* (sin conexión). Con los circuitos virtuales, se establece un canal de comunicación perfecto. Todos los paquetes de mensajes llegan, y lo hacen en la secuencia apropiada. Con un servicio de datagramas, el nivel de red intenta entregar cada paquete de forma independiente del resto. Consecuentemente, los mensajes pueden llegar en desorden, o no llegar en absoluto.

El nivel físico, el de enlace y el de red son dependientes de la red, y su forma detallada de operar puede variar de un tipo de red a otro.

(4) **El nivel de transporte**

El nivel de transporte (o nivel de host a host) proporciona una comunicación de un host a otro para su uso en el nivel de sesión. Debe ocultar todos los detalles de la red de comunicación subyacente al nivel de sesión, de forma que se pueda substituir un tipo de red por otro. En realidad, el nivel de transporte protege la porción de niveles de red del cliente (niveles 5-7) de la parte portadora (niveles 1-3).

(5) **El nivel de sesión**

El papel del nivel de sesión es proporcionar una ruta de comunicación entre dos procesos del nivel de aplicación mediante las posibilidades del nivel de transporte. La conexión entre usuarios se denomina habitualmente sesión, y puede incluir una conexión remota o una transferencia de archivos. Las operaciones involucradas en el establecimiento de una sesión (enlace) son la autenticación y la contabilidad (accounting). Una vez que se ha iniciado la sesión, el nivel deberá controlar el intercambio de datos y la sincronización de las operaciones con datos entre los dos procesos.

(6) **El nivel de presentación**

El nivel de presentación suele efectuar transformaciones de utilidad sobre los datos para resolver cuestiones de heterogeneidad respecto a la presentación de los datos a las aplicaciones. Por ejemplo, permite que un programa interactivo dialogue mediante uno de los terminales de presentación de un conjunto incompatible de éstos. Puede encargarse de la compresión de texto y/o de la encriptación.

(7) **El nivel de aplicación**

El nivel de aplicación proporciona las funciones de alto nivel de la red, tales como el acceso a bases de datos, sistemas de correo y demás. La elección de la aplicación puede dictar la funcionalidad de los servicios proporcionados por los niveles inferiores. En

consecuencia, ciertas áreas de aplicación pueden especificar un conjunto de protocolos a lo largo de los siete niveles que se precisan para soportar la función de procesamiento distribuido requerida. Por ejemplo, una iniciativa de General Motors ha definido un conjunto de protocolos para lograr una interconexión abierta dentro de una planta de fabricación automatizada, los cuales se denominan *Protocolos de Automatización de Fabricación* (MAP; Manufacturing Automation Protocols).

14.5.2 Niveles TCP/IP

El modelo OSI se encuentra ya un poco obsoleto, y no aborda directamente el tema de Internet. El modelo de referencia TCP/IP tiene sólo cinco capas, y se muestra en la Figura 14.9.

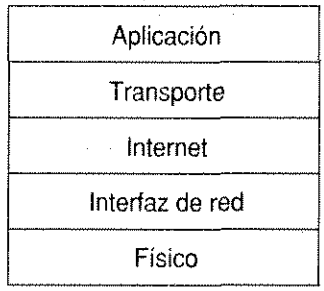


Figura 14.9. El modelo de referencia TCP/IP.

El nivel físico es equivalente al nivel físico de ISO. El nivel de interfaz efectúa las mismas funciones que el nivel de enlace de datos de ISO. El nivel de Internet especifica el formato de los paquetes que se enviarán a lo largo de la interred, y efectúa todas las funciones de encaminamiento asociadas al nivel de red de ISO. Según estos términos se definió el *Protocolo Internet* (IP; Internet Protocol).

El nivel de transporte es equivalente al nivel de transporte de ISO; proporciona dos protocolos: el Protocolo de Datos de Usuario (UDP; User Data Protocol), y el Protocolo de Control de Transmisión (TCP; Transmission Control Protocol). UDP proporciona un protocolo sin conexión no fiable que permite un acceso eficiente al protocolo IP del nivel inferior. El protocolo TCP proporciona un protocolo de flujo de datos (stream) fiable punto a punto.

14.5.3 Protocolos ligeros y redes de área local

Los modelos OSI y TCP/IP se desarrollaron con el fin de permitir el acceso extensible a las redes de área extensa; éstas se caracterizaban por un reducido ancho de banda y altas tasas de error. La mayoría de los sistemas embebidos emplean tecnología de redes de área local, y se encuentran aislados del mundo exterior; las redes de área local se caracterizan por tener un gran ancho de banda de comunicación con bajas tasas de error. Pueden emplear un variado número de tecnologías, tales como tecnología de difusión (como en Ethernet) o tecnología de conmutación punto a punto –por ejemplo ATM (Asynchronous Transfer Mode)–. Es por esto por lo que, aun siendo posible implementar comunicación entre procesos sobre un lenguaje de alto nivel usando

la aproximación OSI o TCP/IP (como hace, por ejemplo, Java RMI), en la práctica el coste suele ser prohibitivo. Así, muchos diseñadores unen los protocolos de comunicación a los requisitos sobre el lenguaje (la aplicación) y el medio de comunicación, resultando los denominados protocolos *ligeros*. Una cuestión clave de su diseño es el grado de tolerancia a los fallos de comunicación.

A primera vista, parece que la completa fiabilidad de la comunicación es un requisito esencial si se precisa escribir aplicaciones distribuidas fiables y eficientes. Sin embargo, no tiene por qué ser siempre así. Considere los tipos de error que pueden presentarse en una aplicación distribuida. Si dos procesos distribuidos comunican y sincronizan sus actividades para proporcionar un servicio, aparecen potenciales errores, como:

- Errores transitorios a causa de interferencias en el medio físico de comunicación.
- Errores de diseño en el software responsable de ocultar los errores transitorios de los subsistemas de comunicación.
- Errores de diseño en los protocolos entre el proceso servidor y cualquier otro servidor necesario para proveer el servicio.
- Errores de diseño en el protocolo entre dos procesos servidores.

Para protegerse contra el último error, es preciso que los procesos servidores proporcionen comprobaciones al nivel de aplicación (comprobación de extremo a extremo). El argumento de un diseño de sistema **de extremo a extremo** sostiene que (Saltzer et al., 1984) dada la necesidad de tales comprobaciones para proveer un servicio fiable, no es necesario repetir estas comprobaciones a niveles inferiores de la jerarquía del protocolo, particularmente cuando el medio de comunicación (por ejemplo, una red de área local, como Ethernet o Token Ring) proporciona un mecanismo de transmisión con una baja tasa de error (aunque no perfecta). En estos casos, es preferible tener un mecanismo de comunicación rápido y fiable, aunque no en un ciento por ciento, que un sistema fiable al ciento por ciento pero lento. Aquellas aplicaciones que requieran alta fiabilidad pueden intercambiar eficiencia por fiabilidad en el nivel de aplicación.

Existen estándares para los protocolos de comunicación de red de área local, particularmente en el nivel de enlace de datos, que se divide en dos subniveles: el de control de acceso al medio (MAC; Medium Access Control) y el de control enlace lógico (LLC; Logical Link Control). MAC se encarga de la interfaz con el medio físico de comunicación, y existen estándares para buses CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*; acceso múltiple con detección de portadora y detección de colisiones) (como por ejemplo, Ethernet), conmutación de paquetes (por ejemplo, ATM), buses, y anillos de paso de testigo. El nivel LLC se encarga de proporcionar un protocolo sin conexión u orientado a conexión.

Como ya se comentó antes, un protocolo habitual orientado al lenguaje es la llamada a procedimiento remoto (ISO/IEC JTC1/SC21/WG8, 1992). Ésta suele implementarse directamente sobre un mecanismo básico de comunicación provisto por la red de área local (por ejemplo, el nivel LLC). Con lenguajes como Ada y Java, en ausencia de fallos hardware, se considera que las llamadas a procedimiento remoto son fiables. Esto es, para cada llamada a procedimiento remo-

to, si se vuelve de la invocación significa que el procedimiento se ha ejecutado una vez y sólo una; a esto se le suele denominar semántica RPC de **exactamente una vez**. Sin embargo, en presencia de fallos en las máquinas, es difícil obtener esto, dado que un procedimiento puede verse ejecutado parcial o totalmente varias veces dependiendo de dónde se produjo la caída y de si el programa fue o no reiniciado. Ada asume que la llamada fue ejecutada *como máximo una vez*, porque no existe la noción de reinicio por parte de un programa Ada tras un fallo.

Para una red de área local de tiempo real y para sus protocolos asociados, es importante proveer retrasos acotados y conocidos en las transmisiones de los mensajes. Se volverá a este tema en la Sección 14.7.2.

14.5.4 Protocolos de comunicación de grupo

La llamada a un procedimiento (o método) remoto es una forma habitual de comunicación entre clientes y servidores en los sistemas distribuidos. Sin embargo, no se limita a la comunicación entre dos procesos. A menudo, cuando un grupo de procesos interacciona (por ejemplo, al realizar una acción atómica), se hace necesario enviar una comunicación a todo el grupo. Estos mecanismos vienen proporcionados por algún paradigma de comunicación por **multidifusión** (multicast). Algunas redes, por ejemplo Ethernet, proporcionan un mecanismo de multidifusión por hardware como parte de su nivel de enlace de datos. Si no es éste el caso, habrá que añadir más protocolos software.

Todas las redes y procesadores son no fiables en mayor o menor medida. Se hace preciso, por tanto, proporcionar un mecanismo de comunicación por multidifusión con garantías específicas:

- **Multidifusión no fiable**: no garantiza el reparto al grupo; el protocolo de multidifusión proporciona un equivalente al nivel de servicio de datagrama.
- **Multidifusión fiable**: el protocolo intenta entregar el mensaje al grupo lo mejor que puede, pero no ofrece ninguna garantía de entrega.
- **Multidifusión atómica**: el protocolo garantiza que si un proceso del grupo recibe el mensaje, lo habrán recibido también los demás miembros del grupo; esto es, el mensaje se entrega a cada miembro del grupo o a ninguno.
- **Multidifusión atómica ordenada**: además de garantizar la atomicidad de la multidifusión, el protocolo también garantiza que todos los miembros del grupo reciban los mensajes de diferentes emisores en el mismo orden de envío.

Cuantas más garantías dé un protocolo, mayor es su coste de implementación. Además, el coste variará en función del modelo de fallo empleado (véase la Sección 5.2).

Para la multidifusión atómica y para la atómica ordenada, es importante poder acotar el tiempo que se invierte en efectuar la operación. Sin esto, será imposible predecir sus prestaciones en un sistema de tiempo real estricto.

Considere una multidifusión atómica ordenada sencilla con este modelo de fallo:

- Los procesadores fallan silenciosamente.
- Todos los fallos de comunicación son fallos de omisión.
- No ocurren más de N fallos de omisión de la red consecutivos.
- La red está completamente conectada y no hay ningún particionado en la red.

Para lograr una transmisión de mensajes atómica, todo lo que se necesita es transmitir cada mensaje $N + 1$ veces; a esto se le denomina **difusión** del mensaje. Para lograr la propiedad de ordenamiento utilizando la difusión es preciso conocer el tiempo que dura la transmisión completa de cada mensaje. Suponga que el valor de la transmisión del peor caso posible para todos los mensajes es T_D, y que los relojes de la red se encuentran débilmente sincronizados con una diferencia máxima de C_Δ. Cada mensaje lleva un sello temporal del emisor con el valor de su reloj local (C_{emisor}). Cada receptor puede entregar el mensaje a su proceso cuando su reloj local es mayor de $C_{\text{emisor}} + T_D + C_\Delta$, ya que es en este momento cuando se garantiza que habrán recibido el mensaje todos los receptores, y cuando el mensaje es **válido**. Estos mensajes podrán ser ordenados según sus tiempos de validez. Si dos mensajes tienen tiempos de validez idénticos, puede imponerse un ordenamiento arbitrario (como usar la dirección de red del procesador). En la Figura 14.10 se muestra la aproximación para $N = 1$.

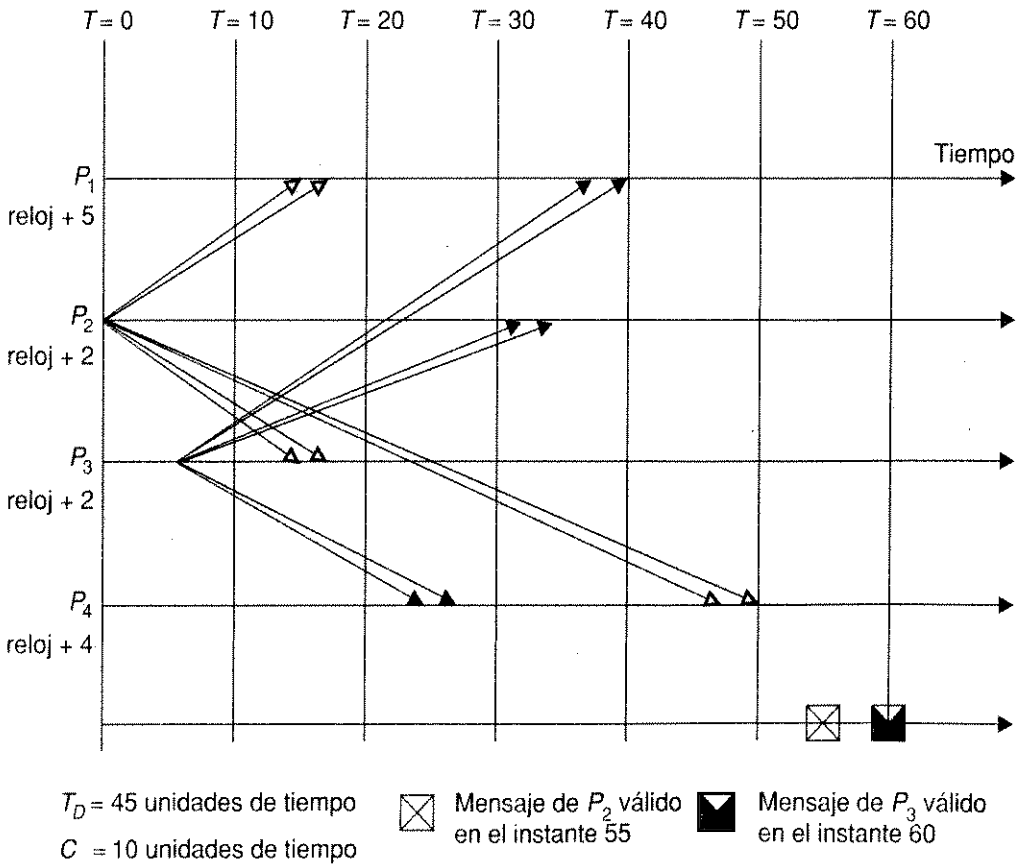


Figura 14.10. Una sencilla multidifusión atómica ordenada basada en difusión.

Observe que aunque el sencillo algoritmo anterior garantiza que todos los procesadores reciben y procesan los mensajes en el mismo orden, no garantiza que el orden sea el auténtico orden en el que se enviaron los mensajes. Esto es así, porque que el valor utilizado para determinar el orden se basa en los relojes locales de los procesadores, que podrán diferir en C_Δ.

14.5.5 Fallo del procesador

En el Capítulo 5 se identificaron dos aproximaciones generales para proporcionar tolerancia a fallos software y hardware; las estáticas (enmascarando) y las de redundancia dinámica. Aunque disponer de hardware tolerante a fallos juega un papel importante en la obtención de fiabilidad en presencia de fallos de comunicación y de procesador, se necesitan grandes cantidades de hardware para implementar la redundancia modular triple, y resulta muy caro (aunque veloz). Esta sección se centra en la forma de proveer tolerancia a fallos mediante métodos software.

Por el momento, se supondrá que todos los procesadores del sistema **fallan silenciosamente**. Esto quiere decir que si un procesador se estropea de cualquier forma, entonces sufrirá un fallo de omisión permanente. Si los procesadores no fallan silenciosamente será posible que envíen mensajes inválidos entre ellos. Esto introduciría un nivel extra de complejidad en la discusión siguiente.

Tolerancia de fallos de procesador mediante redundancia estática

En el Capítulo 5 se discutió la programación de *N*-versiones en el contexto de la tolerancia estática a los fallos de diseño. Obviamente, si cada una de las versiones de un programa de *N*-versiones reside en un procesador diferente, obtendremos también tolerancia a fallos de procesador. Sin embargo, incluso si no introducimos diversidad en el diseño, aún sigue siendo deseable obtener copias idénticas de algunos componentes del sistema para obtener la disponibilidad requerida. Esto se suele denominar **replicación activa**.

Suponga que se diseña una aplicación según el modelo de objetos distribuidos. Resulta posible replicar los objetos sobre procesadores diferentes, e incluso variar el grado de replicación según la importancia del objeto concreto. Para que la réplica de los objetos sea transparente al programador de la aplicación, deben poseer un comportamiento determinista. Esto significa que para una secuencia de peticiones dada sobre un objeto, el comportamiento del objeto es predecible. Si no es éste el caso, los estados de cada una de las réplicas del objeto replicado pueden acabar siendo diferentes. En consecuencia, cualquier petición efectuada a ese objeto podría producir resultados diversos. En definitiva, el conjunto de objetos debe mantenerse consistente: sus miembros no deben divergir.

Si hay que replicar los objetos, también es preciso replicar cada invocación a método remoto. Más aún, será necesario disponer de una semántica RPC exactamente una vez. Como se muestra en la Figura 14.11, cada objeto cliente ejecutará potencialmente una llamada uno a muchos al procedimiento, y cada procedimiento de servicio recibirá una petición muchos a uno. El sistema de ejecución será el responsable de la coordinación de estas llamadas y de garantizar la semántica requerida. Esto conlleva comprobaciones periódicas sobre los lugares de servicio con llama-

das extra para determinar su estatus; un fallo en la respuesta indicará una caída del procesador. En efecto, el sistema de ejecución deberá soportar alguna forma de protocolo de pertenencia al grupo, y un protocolo de comunicación de grupo por *multidifusión atómica ordenada*.

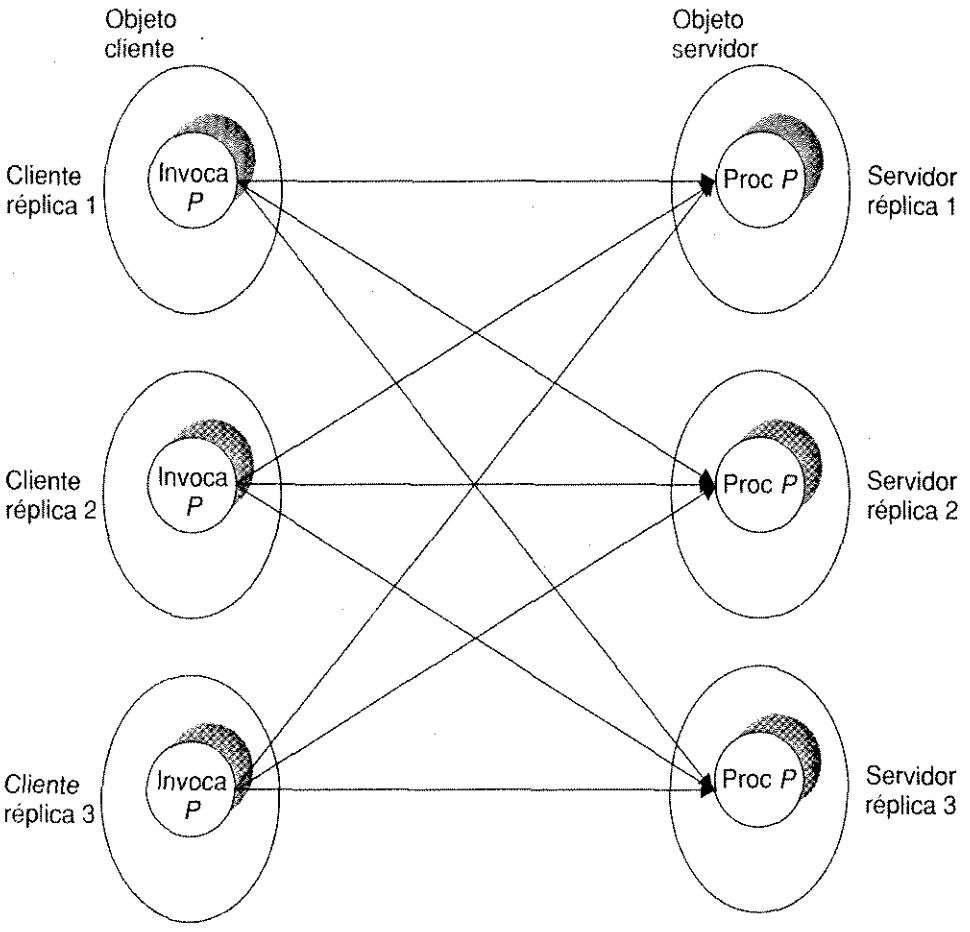


Figura 14.11. Llamadas a procedimiento remoto replicadas.

Un lenguaje que permite explícitamente la replicación (al nivel de proceso) es C concurrente tolerante a fallos (Fault-Tolerant Concurrent C) (Cmelik et al., 1988). El lenguaje supone que los procesadores tienen un modo de fallo del tipo fallo-parada, y proporciona un protocolo de acuerdo distribuido para asegurar que todas las réplicas se comportan de un modo determinista. C concurrente tolerante a fallos tiene un modelo de comunicación y sincronización muy similar a Ada, y así, deberá asegurar que si en una réplica se sigue una rama concreta de una sentencia select, se sigue la misma rama en todas las demás réplicas (incluso si las réplicas no se están ejecutando en un paso de bloqueo). Hay también intentos de obtener un Ada tolerante a fallos (Wellings y Burns, 1996; Wolf, 1998).

Aunque la replicación transparente de objetos es una aproximación atractiva al problema de la tolerancia a fallos de procesador, el coste de la multidifusión subyacente y de los protocolos de acuerdo puede ser prohibitivo para un sistema de tiempo real duro. En concreto, cuando los objetos se encuentran activos y contienen más de una tarea (con tareas potencialmente anidadas), es necesiario que el protocolo de acuerdo de ejecución (que asegura la consistencia entre las réplicas) se ejecute en cada decisión de planificación. Una solución más económica es disponer de

respaldos (standby) **en caliente** con un almacenamiento periódico del estado. Con esta aproximación, podría replicarse un objeto en más de un procesador. Sin embargo, a diferencia de lo que ocurre en la replicación completa, sólo existe una copia activa del objeto en cada momento. El estado de este objeto se almacena regularmente (habitualmente antes y después de la comunicación con otro objeto) en los procesadores donde residen las réplicas. Si el nodo primario falla, se puede arrancar un respaldo desde el último punto de comprobación.

Hay un claro compromiso entre la eficiencia en el uso de los recursos de procesador asignados a un objeto para que esté disponible, y el tiempo necesario para que se recupere de sus fallos. La replicación activa es costosa en recursos de procesamiento, pero requiere muy poco, si acaso tiempo de recuperación; por contra, la replicación en caliente (también llamada replicación **pasiva**) es más barata, pero su tiempo de recuperación es más lento. Esto ha conducido a un tipo de replicación de compromiso, denominado **líder seguidor** (leader follower) (Barrett et al., 1990), donde todas las réplicas se encuentran activas, pero una de ellas se considera como versión primaria y toma todas las decisiones que puedan ser potencialmente no deterministas. Estas decisiones son entonces comunicadas a los seguidores. Esta forma de comunicación es más económica de proveer que la replicación activa, y evita el largo tiempo de recuperación de la replicación pasiva.

Tolerancia a fallos de procesador mediante redundancia dinámica

Uno de los problemas a la hora de proporcionar tolerancia a fallos de modo transparente a la aplicación, es que es imposible para el programador especificar tanto una ejecución segura como degradada.

La alternativa a la redundancia y replicación estática es permitir que el programador de la aplicación maneje los fallos de procesador(es) dinámicamente. Obviamente, las técnicas discutidas hasta este punto, y que permiten que una aplicación tolere faltas en el diseño del software (por ejemplo acciones atómicas) proporcionarán cierta medida de tolerancia de fallos hardware. En el Capítulo 5, se describieron las cuatro fases de la tolerancia a fallos: detección de error, evaluación y confinamiento del daño, recuperación del error y tratamiento del fallo en un servicio continuado. En el contexto de fallo de un procesador, habrá que efectuar las siguientes acciones:

(1) Debe detectarse el fallo del procesador (o procesadores) y comunicarse a los procesadores restantes del sistema. La detección de fallos podría realizarse normalmente mediante el soporte de ejecución distribuido. Sin embargo, debe existir un mecanismo de comunicación apropiado con el software de aplicación que permita indicar qué procesadores han fallado. C concurrente tolerante a fallos proporciona un mecanismo de comunicación de ese tipo.

(2) Es preciso evaluar el daño producido en el fallo; esto requiere conocer qué procesos estaban en funcionamiento en los procesadores estropeados, y qué procesos y qué procesadores permanecen activos (y su estado). Para lograr esto, el programador de la aplicación deberá tener el control sobre el emplazamiento de los objetos en las máquinas. Además, debe estar claro cuál es el efecto del fallo del procesador sobre el pro-

ceso que se ejecuta en el procesador fallido, y sobre sus datos. También deberá definirse el efecto de cualquier interacción, tanto pendiente como futura, con estos procesos y sus datos.

(3) A partir de los resultados de la evaluación de daños, el software superviviente debe llegar a un acuerdo sobre la respuesta al fallo y efectuar las acciones precisas para llevar a cabo esa respuesta. Para obtener una máxima tolerancia, esta parte del procedimiento de recuperación deberá ser distribuida. Si sólo un procesador realizara las operaciones de recuperación, el fallo de este procesador sería catastrófico para la aplicación. La recuperación hará necesario cambiar las rutas de comunicación entre los procesos, de modo que puedan proporcionarse servicios alternativos. También, dado que la respuesta seleccionada dependerá del estado global de la aplicación, será necesario hacer disponibles ciertos datos en todas las máquinas, y que esto se realice conservando un estado consistente. Por ejemplo, las decisiones que habrá que tomar como consecuencia de un fallo en un procesador en un sistema de aviónica dependerán de la altitud de la aeronave. Si hay varios procesadores que presentan valores de altitud diferentes, los procedimientos de recuperación proporcionarán objetivos contrapuestos.

(4) Tan pronto como sea posible, el procesador estropeado y/o su software asociado deberán ser reparados, y el sistema devuelto a su estado normal libre de errores.

Ninguno de los lenguajes de programación de tiempo real considerados en este libro proporciona mecanismos apropiados para hacer frente a la reconfiguración dinámica tras el fallo de un procesador.

Ada y la tolerancia a fallos

El lenguaje Ada no presupone el modelo de fallo que subyace a la implementación de los programas. Todo lo que se indica es la generación de una excepción predefinida, `Communication_Error`, si una partición intenta comunicarse con otra y se detecta un error en el subsistema de comunicación.

Ada tampoco soporta ninguna aproximación concreta a la tolerancia a fallos, pero permite que la implementación proporcione los mecanismos apropiados. La siguiente discusión asume que cierta implementación distribuida de Ada se ejecuta sobre procesadores con modo de fallo del tipo fallo-parada.

La posibilidad de replicar particiones tampoco se establece explícitamente en Ada, aunque cada implementación es libre de hacerlo (como ejemplo véase Wolf, 1998). Sin embargo, puede extenderse el subsistema de comunicación de partición (PCS), de modo que sea posible proporcionar un mecanismo de llamada a procedimiento remoto replicada. Cada partición replicada tiene asociado un identificador de grupo que se puede utilizar por el sistema. Todas las llamadas a procedimiento remoto son llamadas replicadas en potencia. El cuerpo del paquete podría requerir acceder a un mecanismo de multidifusión ordenada. Observe, sin embargo, que esta aproximación no permite la replicación arbitraria de particiones, contando con que el sistema de

ejecución de Ada al completo está involucrado en cada decisión de planificación. En consecuencia, se requerirán mayores restricciones.

Se puede ofrecer notificación asíncrona mediante un objeto protegido en un PCS extendido. El sistema de ejecución podría entonces llamar a un procedimiento cuando detectara un fallo de procesador. Este puede abrir una entry, que podría generar una transferencia asíncrona del control en una o más tareas.

Alternativamente, puede haber una o varias tareas (o un grupo de tareas) esperando la aparición de un fallo. El siguiente paquete muestra esta aproximación.

```
package System.RPC.Reliable is

   type Id_Grupo is range 0 .. definido_implementacion;

   -- Invocación replicada síncrona
   procedure Do_Replicated_RPC(Grupo : in Id_Grupo;
             Parametros : access Params_Stream_Type;
             Resultados : out Param_Stream_Access);

   -- Invocación asíncrona
   procedure Do_Replicated_APC(Grupo : in Id_Grupo;
             Parametros : access Params_Stream_Type);

   type RRPC_Receiver is access procedure (Servicio : in Service_Id;
             Parametros : access Params_Stream_Type;
             Resultados : out Param_Stream_Access);

   procedure Establish_RRPC_Receiver(Particion : in Partition_Id;
             Receptor : in Receptor_RRPC);

   protected Notificacion_Fallo is
      entry Particion_Fallida(P : out Partition_Id);
   private
      procedure Senala_Fallo(P : Partition_Id);
      ...
   end Notificacion_Fallo;

private
      ...
end System.RPC.Reliable;
```

Una vez que se ha notificado a las tareas, éstas deberán evaluar el daño producido al sistema, lo que requiere conocimiento sobre la configuración actual. Es posible la reconfiguración mediante los mecanismos de enlace dinámico del lenguaje –véase Burns y Wellings (1998)–.

Obtención de ejecución fiable de programas occam2

Aunque occam2 se diseñó para su uso en un entorno distribuido, no posee semántica de fallo. Los procesos que fallan debido a un error interno (por ejemplo, un error sobre los límites de un array) son equivalentes al proceso STOP. Un proceso que estuviera esperando comunicar sobre un canal donde el otro proceso estuviera en un procesador estropeado esperaría para siempre, a menos que empleara un tiempo límite de espera. Sin embargo, es posible imaginarse que la semántica adecuada para ambos procesos en occam2 sería considerarlos STOP, y proporcionar un mecanismo donde pudieran ser informados los otros procesos.

Obtención de ejecución fiable de programas Java para tiempo real

Java precisa que cada método que pueda ser invocado remotamente declare `RemoteException` en su lista de objetos lanzables. Esto permite que la implementación subyacente pueda notificar cuándo falla una llamada remota. Además, Java deja abierta la posibilidad de definir servicios remotos que soportan replicación. Actualmente, Java para tiempo real no se pronuncia sobre su aplicación en un sistema distribuido.

14.6 Algoritmos distribuidos

Hasta aquí, este capítulo se ha centrado en la expresión de programas distribuidos en lenguajes como Ada, Java y occam2, así como en los problemas generales de tolerancia a fallos en procesadores y comunicaciones. Queda fuera del alcance de este libro el considerar algoritmos distribuidos específicos que pudieran necesitarse para el control y la coordinación del acceso a los recursos en un entorno distribuido. Sin embargo, es útil establecer ciertas propiedades que pudieran darse por supuestas en un entorno distribuido. En concreto, es necesario mostrar cómo pueden ordenarse los eventos, cómo puede organizarse el almacenamiento de modo que sus contenidos sobrevivan a un fallo de procesador, y cómo puede obtenerse el acuerdo en presencia de procesadores fallidos. Estos algoritmos suelen ser necesarios para implementar los protocolos de multidifusión atómica.

14.6.1 Ordenación de eventos en un entorno distribuido

En muchas aplicaciones, es preciso determinar el orden de los eventos que aparecen en el sistema. Esto no presenta dificultad alguna para los sistemas monoprocesador o para los sistemas fuertemente acoplados, que tienen una memoria común y un reloj común. Para los sistemas dis-

tribuidos, sin embargo, no hay un reloj común, y el retraso que aparece en el envío de los mensajes entre los procesadores indica que estos sistemas tienen dos importantes propiedades:

- Para cualquier secuencia de eventos dada, es imposible probar que dos procesos diferentes observarán, exactamente, la misma secuencia.
- Puesto que los cambios de estado pueden ser vistos como eventos, es imposible probar que dos procesadores cualesquiera tendrán la misma vista global de un subconjunto dado del estado del sistema. En el Capítulo 12, se introdujo la noción de ordenamiento causal de los eventos para ayudar a resolver este problema.

Si los procesos de una aplicación distribuida van a coordinar y sincronizar sus actividades en respuesta a los eventos según aparecen, será necesario disponer un orden causal sobre estos eventos. Por ejemplo, para detectar el interbloqueo entre procesos que comparten recursos, es importante conocer si un proceso liberó el recurso *A* antes de pedir el recurso *B*. El algoritmo que aquí se presenta permite ordenar los eventos de un sistema distribuido, y se debe a Lamport (1978).

Considere el proceso *P*, que provoca los eventos $p0$, $p1$, $p2$, $p3$, $p4$, ..., pn. Dado que es un proceso secuencial, el evento $p0$ deberá haber ocurrido antes que el evento $p1$, el cual deberá haber ocurrido antes que el evento $p2$, etc. Esto se escribe así: $p0 \rightarrow p1 \rightarrow p2 \rightarrow \ldots \rightarrow pn$. De modo análogo, para el proceso *Q*: $q0 \rightarrow q1 \rightarrow q2 \ldots \rightarrow qn$. Si estos dos procesos están distribuidos y no existe comunicación entre ellos, es imposible decir si $p0$ ocurrió antes o después que $q0$, $q1$, y los demás. Considérese ahora que el evento $p1$ está enviando un mensaje a *Q*, y que $q3$ es el evento donde se recibe el mensaje de *P*. En la Figura 14.12 se muestra esta interacción.

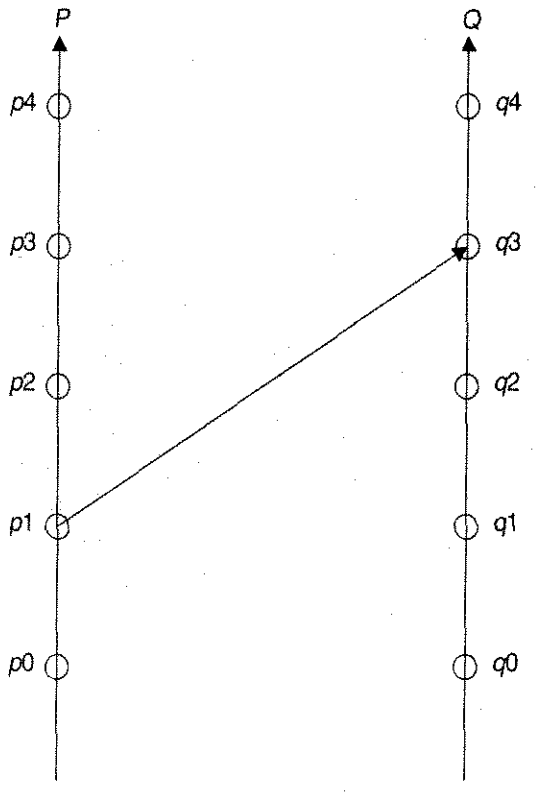


Figura 14.12. Dos procesos que interaccionan.

Como el acto de recibir un mensaje debe ocurrir después de que el mensaje haya sido enviado, entonces $p1 \rightarrow q3$, y como $q3 \rightarrow q4$, se sigue que $p1 \rightarrow q4$ (esto es, $p1$ pudiera tener un efecto causal sobre $q4$). Todavía no tenemos información sobre si ocurrió antes $p0$ o $q0$. Estos eventos se denominan **eventos concurrentes** (dado que no hay ordenamiento causal). Como ninguno de ellos puede afectar al otro, no es realmente importante considerar cuál de ellos ha ocurrido primero. Sin embargo, es importante que aquellos procesos que toman decisiones basadas en este orden asuman todos ellos el mismo ordenamiento.

Para ordenar totalmente todos los eventos de un sistema distribuido, es preciso asociar una *marca temporal* (time-stamp) a cada uno de ellos. Sin embargo, no se trata de una marca temporal física, sino más bien de una marca temporal lógica. Cada procesador del sistema mantiene un reloj lógico, que se incrementa cada vez que ocurre un evento en ese procesador. El evento $p1$ en el proceso P ocurrió antes que $p2$ en el mismo proceso si el valor del reloj lógico en $p1$ es menor que el valor del reloj lógico en $p2$. Obviamente, es posible utilizar este método para el reloj lógico asociado a cada proceso para obtener una sincronización. Por ejemplo, el reloj lógico de P en $p1$ podría ser mayor que el reloj lógico de Q en $q3$; sin embargo, $p1$ deberá haber ocurrido antes que $q3$, porque no se puede recibir un mensaje antes de que haya sido emitido. Para resolver este problema, cuando se comunican mensajes entre dos procesos, hay que enviar la marca temporal del proceso que envió el mensaje. Además, cada vez que un proceso reciba un mensaje debe cambiar su reloj lógico cuando sea mayor que el suyo propio, y será:

- la marca temporal encontrada en el mensaje más uno (al menos) para un paso de mensajes asíncrono; o

- la marca temporal para el paso de mensajes síncrono.

Esto asegura que no se ha recibido ningún mensaje antes de haber sido enviado. Usando el algoritmo anterior, todos los eventos del sistema tienen asociada una marca temporal y puede estar parcialmente ordenada. Si dos eventos tienen la misma marca temporal, es suficiente con emplear una condición arbitraria, tal como el valor numérico de los identificadores de proceso, para obtener un ordenamiento total artificial sobre los eventos.

14.6.2 Implementación del tiempo global

La aproximación al ordenamiento por tiempo global descrita anteriormente empleaba relojes lógicos. Un esquema alternativo es el empleo de un modelo de tiempo global basado en el tiempo físico. Esto puede lograrse si todos los nodos tienen acceso a una única fuente de tiempo; sin embargo, lo normal es que cada nodo tenga su propio reloj. Por ello es preciso coordinar los relojes locales. Existen muchos algoritmos para ello, y todos ellos implican son la compensación de la inevitable deriva de reloj que aparece aun cuando todos los relojes son idénticos por construcción.

En los relojes de cuarzo, la deriva entre dos de ellos podrá ser de un segundo en aproximadamente seis días. Si la temporización de los eventos es significativa a un nivel de milisegundos, la deriva se convierte en un problema tras solamente ocho minutos de funcionamiento.

Para limitar la deriva del reloj, es preciso manipular el tiempo base en cada nodo. El tiempo nunca deberá volver atrás; por esto, aunque sea posible adelantar un reloj lento cierto número de pulsos, a los relojes rápidos sólo es posible ralentizarlos, y no atrasarlos.

Cualquier algoritmo de coordinación de reloj deberá proporcionar un límite (Δ) sobre la máxima diferencia entre dos relojes cualesquiera (o así deberá percibirse desde un reloj de referencia externo). Deberá asumirse que el evento A precede al evento B si (y sólo si):

$$t(A) + \Delta < t(B)$$

donde $t(A)$ es el instante del suceso A.

Una forma de obtener la sincronización entre los nodos es que un proceso central servidor de tiempo (S) suministre tiempo según su propio reloj a cada petición. El servidor de tiempo podrá encontrarse ligado a una fuente de tiempo externa o disponer de un reloj hardware más preciso. Otra posibilidad es considerar que cierto nodo del sistema funcione como fuente de tiempo del sistema. En un sistema distribuido, las comunicaciones no son instantáneas, y por esto el tiempo proporcionado por S se encuentra sujeto a dos fuentes de error: la variabilidad en el tiempo que toma la respuesta a la petición desde S, y cierto no determinismo introducido por la capacidad de respuesta del cliente una vez que obtiene dicho mensaje. También hay que enfatizar que S no deberá ser interrumpido entre la lectura de su reloj y el envío del mensaje de respuesta.

Para reducir las fluctuaciones introducidas en la ida y vuelta del mensaje, S podrá informar de su «tiempo» periódicamente. Si además se emplea un mensaje de alta prioridad (y el mismo S posee prioridad alta), entonces podrán minimizarse las otras fuentes de variabilidad.

Una aproximación alternativa es que el cliente de S envíe la lectura de su propio reloj en el mensaje. Al recibir el mensaje, S observará su propio reloj y enviará un factor de corrección, δ:

$$\delta = t(\text{cliente}) + \text{min} - t(S)$$

donde min es el retraso mínimo para la comunicación del mensaje desde el cliente.

Entonces se transmitirá el factor de corrección al cliente. El tiempo de transmisión del segundo mensaje no es tan importante. Al recibir el mensaje de respuesta, el cliente deberá avanzar su propio reloj en la cantidad δ si su valor es negativo, o ralentizar su reloj en la cantidad δ si es positivo.

El empleo de una sola fuente de tiempo proporciona un esquema simple, pero existe la posibilidad de un fallo en un único punto, S, o en el nodo sobre el que se esté ejecutando. Otros algoritmos emplean una aproximación descentralizada, en la que todos los nodos difunden su «tiempo» y se toma un valor votado por consenso. En dicha votación pueden omitirse los valores desproporcionados. Estos algoritmos de sincronización de reloj deben satisfacer tanto la propiedad de **acuerdo** como la propiedad de **precisión**, donde:

- La condición de acuerdo se satisface si la tasa de deriva entre dos relojes no fallidos se encuentra limitada.

- La condición de aproximación se satisface sólo si los relojes no fallidos tienen una deriva limitada con respecto al tiempo real.

14.6.3 Implementación de un almacenamiento estable

En muchas circunstancias, es necesario disponer de un almacén cuyos contenidos sobrevivan a una caída en un procesador; esto se denomina **almacenamiento estable**. Dado que la memoria principal de cualquier procesador es volátil, es preciso usar un disco (o cualquier otra forma de almacenamiento no volátil) como dispositivo de almacenamiento estable. Desgraciadamente, las operaciones de escritura sobre disco no son atómicas, dado que la operación puede malograrse por el camino. Cuando ocurre esto, el gestor de recuperación no puede determinar si la operación se produjo o no. Para resolver este problema, cada bloque de datos se almacena por duplicado sobre áreas de disco separadas. Estas áreas se escogen de modo que un aterrizaje de cabezas durante una lectura sobre un área no destruya la otra; si es preciso, podrán encontrarse sobre discos separados físicamente. Se supone que la unidad de disco indicará si cada operación aislada de escritura se ha completado con éxito (mediante comprobaciones redundantes). La aproximación, en ausencia de fallo en el procesador, es escribir el bloque de datos sobre la primera área del disco (esta operación podrá repetirse hasta que tenga éxito); únicamente cuando esto se ha podido efectuar, se escribirá dicho bloque sobre el segundo área del disco.

Si se produce una caída mientras se está actualizando un almacenamiento estable, podrá ejecutarse la siguiente rutina de recuperación.

```
lee_bloque1;
lee_bloque2;
if ambos_son_legibles and bloque1 = bloque2 then
     -- no hacer nada, la caída no afectó al almacén estable
else
     if un_bloque_es_ilegible then
          -- copia el bloque bueno sobre el bloque malo
     else
          if ambos_son_legibles_pero_diferentes then
               -- copia el bloque1 sobre el bloque2 (o viceversa)
          else
               -- ha ocurrido un fallo catastrófico y ambos
               -- bloques se encuentran ilegibles
          end
     end;
end;
```

Este algoritmo funcionará incluso si ocurren nuevas caídas durante su ejecución.

14.6.4 Obtención de acuerdo en presencia de procesos fallidos

Al principio de este capítulo se tomó como premisa que si un procesador falla, lo hace de modo silencioso. Según esto, el procesador detiene efectivamente *toda* ejecución, y no toma parte en ninguna comunicación con cualquier otro procesador del sistema. De hecho, incluso el algoritmo de almacenamiento estable presentado arriba supone que la caída de un procesador implica una detención inmediata de su funcionamiento. Sin esta premisa, un procesador incorrecto podría efectuar transiciones de estado arbitrarias y enviar mensajes espúreos a los otros procesadores. En esta situación, ni siquiera un programa lógicamente correcto podría garantizar el resultado deseado. Esto haría casi imposible construir sistemas tolerantes a fallos. Aunque pudiera invertirse todo el esfuerzo posible en construir procesadores que operaran en presencia de fallos en componentes, sería *imposible* garantizar esto último empleando una cantidad *finita* de hardware. Consecuentemente, no queda más remedio que establecer un límite en el número de fallos. Esta sección considera el problema de cómo un grupo de procesos que se ejecutan sobre procesadores diferentes pueden lograr un consenso en presencia de cierto número de procesos fallidos dentro del grupo. Se parte de la base de que el sistema de comunicación es fiable (esto es, se encuentra lo suficientemente replicado como para garantizar un servicio fiable).

Problema de los generales bizantinos

El problema de acordar valores entre procesos que pueden residir en procesadores con fallos se suele denominar **problema de los generales bizantinos** (Lamport et al., 1982). Varias divisiones del ejército bizantino, cada una mandada por su propio general, cercan un acuartelamiento enemigo. Los generales, que pueden comunicarse mediante mensajeros, deben llegar a un acuerdo sobre el ataque al campamento. Para esto, cada uno observa el campamento enemigo y comunica sus observaciones al resto. Por desgracia, puede que uno o más de los generales sean traidores y capaces de comunicar información falsa. El problema para los generales leales es cómo obtener la misma información. En general, $3m + 1$ generales pueden contra m traidores si efectúan $m + 1$ rondas de intercambio de mensajes. Para simplificar, se muestra dicha aproximación del algoritmo con cuatro generales que pueden vérselas con un traidor, los cuales tomarán las siguientes premisas:

(1) Cada mensaje que se envía es entregado correctamente.

(2) El receptor de un mensaje sabe quién lo envió.

(3) Se puede detectar la ausencia de mensaje.

Se remite al lector a la literatura en la materia para soluciones más generales (Pease et al., 1980; Lamport et al., 1982).

Considere un general (Gi) y la información que ha observado (Oi). Cada general mantiene un vector V de información recibida de los otros generales. Inicialmente, el vector de Gi tan sólo contiene el valor Oi; esto es, $V\,i(j)$ = nulo (para $i <> j$) y $V\,i(j) = Oi$. Cada general envía un mensajero a cada uno de los otros generales indicando su observación; los generales leales siempre

enviarán la observación correcta; el general traidor podrá enviar una observación falsa, y puede que ésta sea diferente para cada uno de los generales. Al recibir las observaciones, cada general actualiza su vector y envía el valor de las otras tres observaciones a los otros generales. Obviamente, el traidor podrá enviar observaciones diferentes de las que hubiera recibido, o ninguna de ellas. En el último caso, los generales podrán elegir un valor arbitrario.

Tras este intercambio de mensajes, cada general leal podrá construir un vector con el valor mayoritario de aquellos tres valores que hubiera recibido de cada observación de cada general. Si no hubiera tal mayoría, supondrá que, en realidad, no se ha efectuado ninguna observación.

Suponga, por ejemplo, que las observaciones de un general le han conducido a una de estas tres conclusiones: ataque (*A*), retirada (*R*), o espera (*E*). Considere el caso de que *G*1 sea un traidor, *G*2 concluye ataque, *G*3 ataca y *G*4 espera. Inicialmente, el estado de cada vector es el indicado en la Figura 14.13.

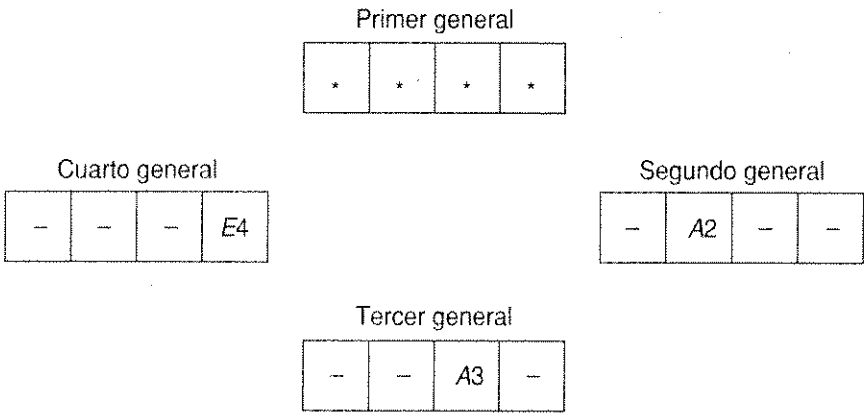


Figura 14.13. Generales bizantinos: estado inicial.

El índice sobre el vector (1..4) proporciona el número de general; los contenidos del elemento de cada índice dan la observación de ese general y quién informó de la observación. Inicialmente, cuando el cuarto general almacena «espera» en el cuarto elemento de su vector, indica que la observación vino de él mismo.

Por supuesto, en el caso general podría ser imposible autenticar quién informó de la observación. Pero, en este ejemplo, puede suponerse que el traidor no es tan enrevesado.

Tras el primer intercambio de mensajes, los vectores son (suponiendo que el traidor se da cuenta de que el campamento es vulnerable y envía arbitrariamente los valores retirada y espera a todos los generales) los que se muestran en la Figura 14.14. Tras el segundo intercambio de mensajes, la información de que dispone cada general es la que se muestra en la Figura 14.15 (suponiendo de nuevo que el traidor envía aleatoriamente los mensajes retirada y espera).

Para ver cómo se ha obtenido esta tabla, considere la última fila para el cuarto general. Esta información ha sido obtenida, supuestamente, del tercer general (como indica el 3), e incluye información sobre las decisiones del primero y segundo generales. Así pues, el *R*3 de la primera columna indica que el tercer general cree que el primer general desea la retirada.

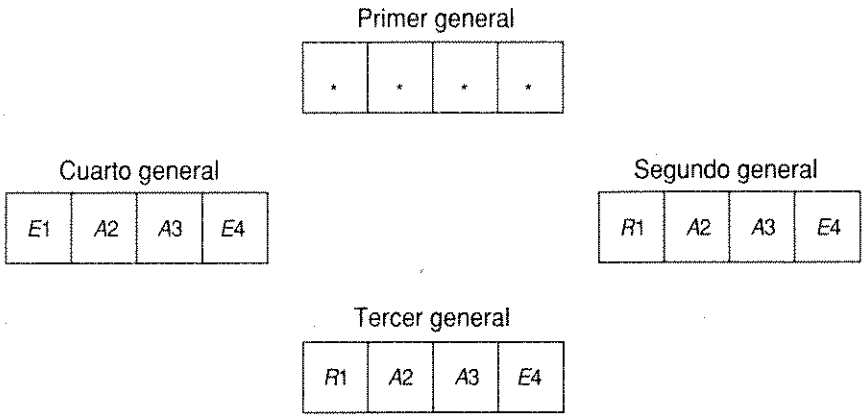


Figura 14.14. Generales bizantinos: estado tras el primer intercambio de mensajes.

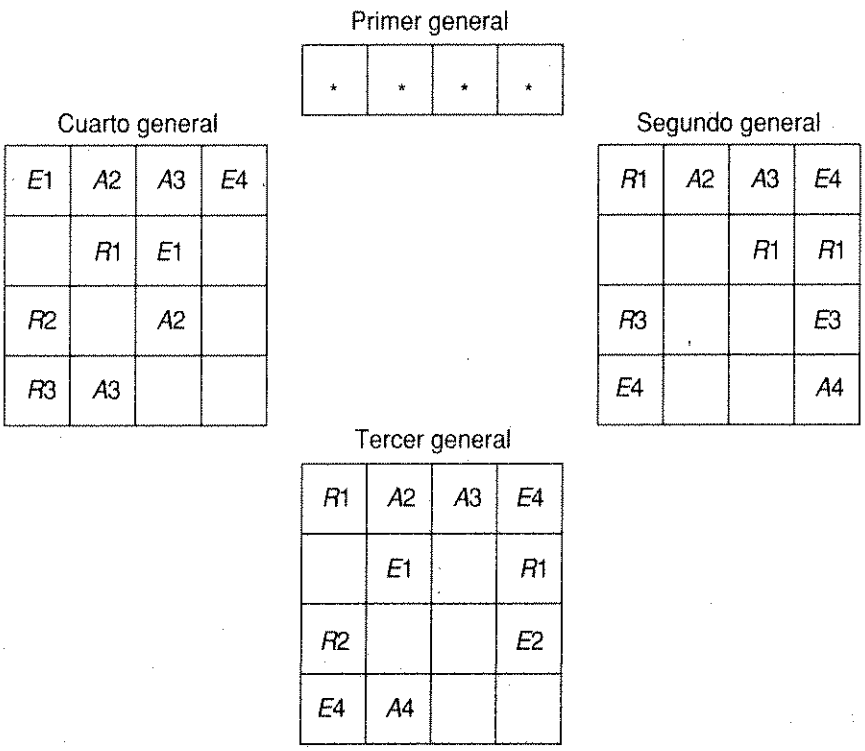


Figura 14.15. Generales bizantinos: segundo intercambio de mensajes.

La Figura 14.16 proporciona un vector final (por mayoría). Los generales leales tienen una visión consistente de las observaciones del resto, y en consecuencia pueden tomar una decisión uniforme.

Si fuera posible restringir las acciones de los traidores (es decir, un modelo de fallo más estricto), entonces podría reducirse el número de generales (procesadores) necesarios para tolerar m traidores (fallos). Por ejemplo, si un traidor fuera incapaz de modificar una observación de un general leal (digamos que el general debe pasar una copia firmada de la observación y que la firma no puede modificarse o falsificarse), entonces sólo se precisarían $2m + 1$ generales (procesadores).

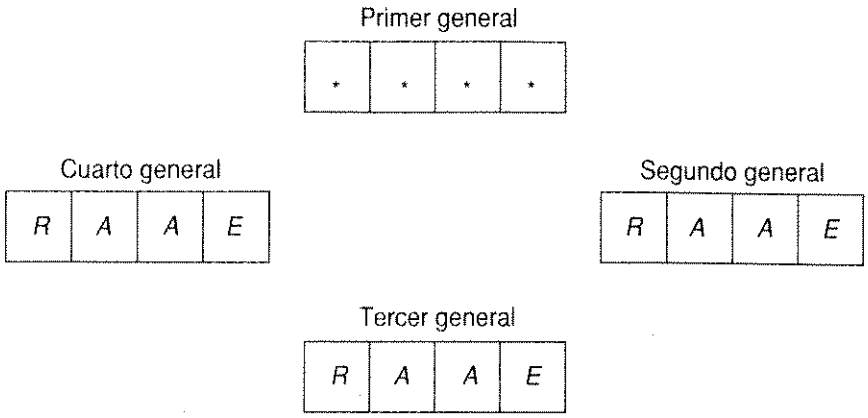


Figura 14.16. Generales bizantinos: estado final.

A partir de las soluciones al problema de los generales bizantinos, es posible construir un procesador con fallo silencioso disponiendo de procesadores replicados internamente que realizan un acuerdo bizantino (Schneider, 1984). Si los procesadores correctos detectan algún desacuerdo, podrán detener su ejecución.

14.7 Planificación con tiempo límite en un entorno distribuido

Habiendo considerado ya cierto número de arquitecturas, protocolos de comunicación y algoritmos distribuidos, es posible volver ahora a la cuestión central de comprender el comportamiento temporal de las aplicaciones construidas sobre un entorno distribuido. Aquí, las partes del sistema pueden estar progresando a diferentes ritmos, y los retrasos en las comunicaciones pueden resultar significativos. No solamente se encontrará ligado el tiempo de ejecución al de los computadores, sino que que los diversos procesadores/nodos deben disponer de alguna forma de vínculo en el tiempo. El término **síncrono** se emplea (en este contexto) para denominar a un sistema distribuido que presenta las siguientes propiedades:

- Existe una cota superior en los retrasos de los mensajes; ésta consta del tiempo que tarda el envío, el transporte, y la recepción de un mensaje en el enlace de comunicación.
- Cada procesador dispone de un tiempo local, y existe una cota en la tasa de deriva entre dos relojes cualesquiera.
- Los procesadores en sí mismos progresan, como mínimo, a cierta velocidad.

Observe que esto no implica necesariamente que no ocurran fallos. Más bien, la cota superior en el retraso de los mensajes subsistirá aun cuando los procesadores correctos se comuniquen sobre un enlace sin fallos. En realidad, la existencia de esta cota superior podrá utilizarse para realizar la detección de fallos.

Diremos que un sistema es **asíncrono** cuando no presenta ninguna de las tres propiedades anteriores.

En esta sección sólo se tendrán en cuenta los sistemas síncronos, y se indagará sobre dos cuestiones principales: el tema de la asignación de procesos a procesadores, y la planificación de las comunicaciones.

14.7.1 Asignación

El desarrollo de esquemas de planificación apropiados para sistemas distribuidos (y multiprocesador) es problemático. Graham (1969) mostró que los sistemas multiprocesador pueden comportarse de un modo bastante impredecible en cuanto al comportamiento de temporización que presentan. Él empleó asignación dinámica (esto es, procesos que se asignan a procesadores cuando pasan al estado de ejecutables), y fue capaz de apuntar las siguientes anomalías:

- Incrementar el tiempo de ejecución de un proceso P, podría conducir a que tuviera un tiempo de respuesta mayor.
- Incrementar la prioridad de un proceso P, podría conducir a que tuviera un tiempo de respuesta mayor.
- Incrementar el número de procesadores podría conducir a que P tuviera un tiempo de respuesta mayor.

Todos estos resultados van claramente contra la intuición.

Mok y Dertouzos (1978) mostraron que los algoritmos que son óptimos para sistemas monoprocesador no son óptimos para mayores números de procesadores. Considere, por ejemplo, tres procesos periódicos, $P1$, $P2$ y $P3$, que se ejecutarán sobre dos procesadores. Pongamos que $P1$ y $P2$ tienen los mismos requisitos de tiempo límite –en concreto un periodo y tiempo límite de 50 unidades de tiempo, y un requisito de ejecución (por ciclo) de 25 unidades–; supongamos que $P3$ tiene unos requisitos de 100 y 80. Si se emplea el algoritmo de tasa monotónica (discutido en el Capítulo 13), $P1$ y $P2$ tendrán la mayor prioridad, y se lanzarán en ambos procesadores (en paralelo) durante las indicadas 25 unidades. Esto dejará a $P3$ con tan sólo 75 unidades para completar una tarea de 80 unidades de ejecución. El hecho de que $P3$ disponga de dos procesadores es irrelevante (uno de ellos permanecerá ocioso). Como resultado de aplicar el algoritmo de tasa monotónica, $P3$ perderá su tiempo límite, incluso aunque la utilización media de los procesadores sea tan sólo del 65 por ciento. Sin embargo, una asignación de $P1$ y $P2$ a un procesador y de $P3$ al otro, satisfará fácilmente sus requisitos temporales.

Otros ejemplos podrían demostrar que la formulación «primero el tiempo límite más temprano» (EDF; earliest deadline first) es igualmente no óptima. Esta dificultad con los algoritmos óptimos monoprocesador no debe sorprender, dado que se sabe que la planificación óptima para sistemas multiprocesador es NP-duro (Graham et al., 1979). Es pues necesario buscar formas de simplificar el problema y proporcionar algoritmos que den resultados subóptimos adecuados.

Asignación para procesos periódicos

La discusión anterior mostró que una asignación juiciosa de procesos puede afectar significativamente a la planificabilidad. Considérese otro ejemplo; esta vez son cuatro procesos para ejecutarse sobre dos procesadores; pongamos que sus tiempos de ciclo son 10, 10, 14 y 14. Si los dos de 10 se asignan al mismo procesador (y en consecuencia los otros dos de 14 al otro), se obtiene una utilización de procesadores del 100 por ciento. El sistema es planificable incluso si los tiempos de ejecución para los cuatro procesos son (digamos) 5, 5, 10 y 4. Sin embargo, si se situaran juntos a uno de 10 y uno de 14 sobre el mismo procesador (como resultado de una asignación dinámica), entonces la utilización máxima caería al 83 por ciento.

Lo que parece demostrar este ejemplo es que es mejor asignar los procesos periódicos estáticamente que dejarlos migrar (y, como consecuencia, desbalancear la asignación y, potencialmente, degradar las prestaciones del sistema). Incluso en un sistema fuertemente acoplado que ejecuta un único distribuidor de ejecución, es mejor mantener los procesos sobre el mismo procesador que intentar utilizar un procesador ocioso (y arriesgarse a desbalancear la asignación).

Si se usa un despliegue estático, entonces el algoritmo de tiempo límite monotónico, u otros esquemas óptimos monoprocesador, pueden comprobar la planificabilidad sobre cada procesador. Al efectuar la asignación, los procesos relacionados armónicamente deberían desplegarse juntos (esto es, sobre el mismo procesador), puesto que esto ayudará a incrementar la utilización.

Asignación de procesos esporádicos y aperiódicos

De la misma forma que parece necesario asignar estáticamente los procesos periódicos, la misma aproximación podría parecer útil para un modelo de procesos esporádicos. Si todos los procesos se aplican estáticamente, entonces podrán usarse los algoritmos discutidos en el Capítulo 13 sobre cada procesador (esto es, cada procesador, en realidad, corre su propio planificador/distribuidor).

Calcular los tiempos de ejecución (en el caso medio y en el peor) requiere conocer los bloqueos potenciales. El bloqueo interno al procesador puede delimitarse mediante protocolos de herencia o de acotación (ceiling). Sin embargo, en un sistema multiprocesador existe otra forma de bloqueo: éste ocurre cuando un proceso es retrasado por otro proceso de otro procesador. Esto se denomina bloqueo remoto, y no se acota fácilmente. En un sistema distribuido, puede eliminarse el bloqueo remoto añadiendo procesos que gestionen la distribución de los datos. Por ejemplo, en lugar de estar bloqueado esperando a leer ciertos datos de un lugar remoto, podría añadirse un proceso extra a dicho lugar remoto cuyo papel fuera enviar los datos donde se precisen. Así, los datos estarían disponibles localmente. Este tipo de modificación sobre el diseño puede hacerse sistemáticamente, pero complica la aplicación (aunque conduce a un modelo de planificación más simple).

Una de las desventajas de una política de asignación puramente estática es que no se obtiene beneficio alguno de la capacidad de reserva de algún procesador cuando algún otro procesador experimenta una sobrecarga transitoria. Para los sistemas de tiempo real estrictos, es necesario que cada procesador sea capaz de tratar con los peores casos de tiempo de ejecución de sus procesos periódicos, y con los tiempos máximos de llegada y ejecución para su carga esporádica. Pa-

ra mejorar esta situación, Stankovic et al. (1985) y Ramamritham y Stankovic (1984) han propuesto algoritmos de planificación de tareas más flexibles.

En sus aproximaciones, todos los procesos periódicos críticos para la seguridad y los procesos esporádicos se asignan estáticamente, pero los procesos aperiódicos no críticos pueden migrar. Se emplea el siguiente protocolo:

- Cada proceso aperiódico llega a algún nodo de la red.
- El nodo donde llega el proceso aperiódico comprueba si este nuevo proceso puede planificarse junto con la carga existente. Si se puede, se dice que el proceso está garantizado por este nodo.
- Si el nodo no puede garantizar el nuevo proceso, busca nodos alternativos que puedan garantizarlo. Esto se hace usando conocimiento sobre el estado del conjunto de la red y pidiendo capacidad extra en los otros nodos.
- El proceso es, así pues, movido a un nuevo nodo donde haya una alta probabilidad de que pueda ser planificado. Sin embargo, a causa de condiciones de carrera, puede que el nodo no sea capaz de planificarlo una vez que ha llegado. Entonces se pasa el test de garantía localmente; si el proceso falla, el test deberá moverse de nuevo.
- De esta forma, un proceso aperiódico o bien es planificado (garantizado), o bien falla en la consecución de su tiempo límite.

La utilidad de su aproximación viene mejorada por el uso de un algoritmo heurístico lineal para determinar dónde debería moverse un proceso no garantizado. Esta heurística es poco costosa computacionalmente (a diferencia del algoritmo óptimo, que es NP-duro), pero proporciona un alto grado de éxito; esto es, existe una alta probabilidad de que mediante la heurística se llegue a la planificación de un proceso aperiódico (si es planificable de alguna forma).

El coste de la ejecución del algoritmo heurístico y el movimiento de los procesos aperiódicos es tenido en cuenta por la rutina de garantía. No hay que decir que el esquema es aplicable sólo si es posible mover los procesos aperiódicos y si este movimiento es eficiente. Algunos procesos aperiódicos pudieran estar fuertemente acoplados a un tipo de hardware concreto sobre un nodo, y tendrán al menos algún componente que deberá ejecutarse localmente.

14.7.2 Planificación del acceso a los enlaces de comunicación

La comunicación entre procesos que están en diferentes máquinas en un sistema distribuido requiere la transmisión y la recepción de los mensajes a través del subsistema de comunicación subyacente. En general, estos mensajes deberán competir unos con otros para ganarse el acceso al medio de comunicación (por ejemplo, conmutadores, buses o anillos). Para que los procesos de tiempo real cumplan sus tiempos límites, deberá ser necesario planificar el acceso al subsistema de comunicación de forma que sea consistente con la planificación de los procesos de cada pro-

cesador. Si éste no es el caso, entonces puede aparecer una inversión de la prioridad cuando un proceso de alta prioridad intenta acceder al enlace de comunicación. Los protocolos estándar como aquéllos asociados con Ethernet no soportan el trafico con restricciones de tiempo estrictas, ya que tienden a incluir los mensajes en colas FIFO o a utilizar algoritmos de retractación (back-off) no predecibles cuando se detecta una colisión de mensajes.

Aunque el enlace de comunicación es sólo otro recurso, existen al menos cuatro cuestiones que diferencian el problema de la planificación de enlaces del de la planificación de procesadores.

- A diferencia de un procesador, que tiene un único punto de acceso, un canal de comunicación tiene muchos puntos de acceso; uno por cada nodo físicamente conectado.
- Mientras que los algoritmos apropiativos son adecuados para la planificación de procesos sobre un único procesador, la apropiación durante la transmisión del mensaje hace que necesite retransmitirse el mensaje completo.
- Además de los tiempos límite impuestos por los procesos de la aplicación, habría también que imponer tiempos límite sobre la disponibilidad del búfer; los contenidos de cada búfer deberán ser retransmitidos antes de que puedan situarse nuevos datos en él.

Aunque se han empleado muchas aproximaciones *ad hoc*, en los sistemas distribuidos hay al menos tres esquemas que permiten un comportamiento predecible.

TDMA

La extensión natural al empleo de un ejecutor cíclico para la planificación monoprocesador, es emplear una aproximación cíclica para las comunicaciones. Si todos los procesos de la aplicación son periódicos, es posible producir un protocolo de comunicación por franjas de tiempo. Tales protocolos se denominan TDMA (*Time Division Multiple Access;* acceso múltiple por división del tiempo). Cada nodo dispone de un reloj sincronizado con el resto de relojes de los nodos. A cada nodo se le asignan franjas de tiempo en las que puede comunicarse durante cada ciclo de comunicación. Éstas se encuentran sincronizadas con las franjas de ejecución del distribuidor cíclico de cada tiempo. No deberían aparecer colisiones, dado que cada nodo sabe cuándo puede escribir, y además sabe cuándo hay un mensaje disponible que necesita leer.

La dificultad con la aproximación TDMA está en construir la planificación; además, esta dificultad se incrementa exponencialmente con el número de nodos del sistema. Una arquitectura que ha mostrado considerable éxito en el empleo de TDMA es TTA (*Time Triggered Architecture*; arquitectura disparada por tiempo) (Kopetz, 1997), que emplea una heurística de reducción de un grafo complejo para construir los planes. La otra desventaja de TDMA es que es difícil planificar cuándo pueden comunicarse los mensajes esporádicos.

Esquemas de paso de testigo temporizado

Una aproximación más general a la basada puramente en franjas de tiempo es el empleo de un esquema de paso de testigo: cierto mensaje especial (**testigo**) es pasado de un nodo a otro. Los

nodos pueden enviar mensajes sólo cuando poseen el testigo, y dado que sólo hay un testigo, no podrá haber colisiones entre mensajes. Cada nodo posee el testigo sólo durante un tiempo máximo, y por esto el **tiempo de rotación del testigo** se encuentra limitado.

Cierto número de protocolos emplean esta aproximación, siendo un ejemplo de ello el protocolo de fibra óptica FDDI (*Fiber Distributed Data Interface*; interfaz de datos distribuido de fibra). Aquí los mensajes se agrupan en dos clases, denominándose, aunque de forma confusa, síncronos y asíncronos. Los mensajes síncronos tienen fuertes restricciones de tiempo, y se emplean para definir el tiempo de posesión del testigo en cada nodo y, por tanto, el *tiempo de rotación del testigo destino*. Los mensajes asíncronos no se limitan con restricciones fuertes; pueden ser comunicados por un nodo tanto si no tienen mensajes síncronos que enviar como si el testigo ha llegado tempranamente (porque los otros nodos no tengan nada que transmitir). El comportamiento de este protocolo en el peor caso ocurre cuando llega un mensaje mensaje a un nodo justo cuando está siendo pasado el testigo; hasta ese momento ningún nodo ha transmitido mensajes, y por ello el testigo está siendo entregado demasiado pronto. Tras pasar el testigo desde un nodo con el nuevo mensaje síncrono, el resto de los nodos tendrán ahora montones de mensajes que enviar. El primer nodo envía su carga síncrona y, como el testigo ha llegado demasiado pronto, envía un montón de mensajes asíncronos; todos los nodos subsiguientes envían su carga síncrona. En el instante en que el testigo vuelve al nodo de interés, habrá pasado un intervalo de tiempo igual al doble del *tiempo de rotación del testigo destino*. Éste es precisamente el tiempo de entrega acotado.

Protocolos basados en prioridad

Dados los beneficios derivados del uso de la planificación basada en prioridad en los procesadores, es razonable suponer que será útil aplicar este tipo de mecanismo a la planificación de los mensajes. Tales protocolos suelen tener dos fases: en la primera fase, cada nodo indica la prioridad del mensaje que desea transmitir. Ésta será, obviamente, la prioridad máxima de todo el conjunto de mensajes que va a emitir. Al final de esta fase, uno de los nodos habrá ganado el derecho a transmitir su mensaje. En algunos protocolos, podrán solaparse las dos fases (esto es, mientras cierto mensaje está siendo difundido, se modificarán partes del mensaje de modo que se pueda determinar la prioridad del próximo mensaje).

Si bien durante cierto tiempo se han venido definiendo protocolos basados en prioridad, éstos solían tener la desventaja de que los protocolos de comunicación sólo soportaban un pequeño rango de prioridades. Se idearon para distinguir entre amplias clases de mensajes, más que con el fin de planificar dichos mensajes. Como se indicó en el Capítulo 13, los mejores resultados para la planificación basada en prioridad se obtienen cuando cada proceso (o mensaje, en esta discusión) tiene una prioridad diferente. Hoy en día, afortunadamente, existen protocolos que proporcionan un amplio rango de prioridades; un ejemplo es CAN (*Controller Area Network*; red de área de controladores) (Tindell et al., 1995).

En CAN 2.0A, cada mensaje de 8 bytes viene precedido por un identificador de 11 bits que sirve de prioridad. Al comienzo de la secuencia de transmisión, cada nodo escribe (simultáneamente) al bus de difusión el primer bit de su identificador de mensaje de máxima prioridad. El

protocolo CAN actúa como una gran puerta AND; si cualquier nodo escribe un 0, todos los nodos leen un 0. El bit 0 se considera *dominante*. El protocolo procede como sigue (para cada bit del identificador):

- Si un nodo transmite un cero, continuará con el próximo bit.
- Si un nodo transmite un uno y lee un uno, continuará con el próximo bit.
- Si un nodo transmite un uno y lee un cero, se retrae, y no toma parte en esta ronda de transmisión.

Cuanto menor sea el valor del identificador, mayor prioridad tendrá. Como los identificadores son únicos, el protocolo fuerza a que sólo un nodo acabe dominando tras 11 rondas bit a bit.

El valor de CAN es que es un genuino protocolo basado en prioridad, y por ello podrá aplicarse todo el análisis presentado en el Capítulo 13. La desventaja del tipo de protocolo empleado por CAN es que restringe la velocidad de comunicación. Para que sea posible que todos los nodos escriban sus bit de identificación «a la vez», y para que todos lean los valores AND subsiguientes (y actúen conforme a este valor antes de enviar, o no, el siguiente bit), hay que imponer severos límites en la velocidad de transmisión. Estos límites son, en realidad, función de la longitud del cable empleado para transmitir los bits, y por ello CAN no es adecuado para entornos geográficamente dispersos. En realidad, CAN fue diseñado para la informática del interior de los automóviles modernos, donde ha tenido un considerable éxito.

ATM

ATM puede usarse tanto en redes de área extensa como en redes de área local. El objetivo es dar soporte a un amplio rango de requisitos de comunicación, como los que se requieren para voz y vídeo, o como los de transmisión de datos. ATM permite la comunicación punto a punto mediante uno o más conmutadores. Habitualmente, se conecta cada computador de la red a un conmutador mediante dos fibras ópticas: una para el tráfico hacia el conmutador, y la otra para el tráfico en sentido contrario.

Cualquier dato transmitido se divide en paquetes de tamaño fijo llamados **células**, cada una de las cuales tiene una cabecera de 5 bytes y un campo de datos de 48 bytes. Las aplicaciones se comunican una con otra mediante **canales virtuales** (VC; virtual channels). Los datos que se transmiten en cada canal virtual tienen cierto comportamiento de temporización asociado, como cadencia de bit, periodo o tiempo límite. Un **nivel de adaptación** proporciona servicios específicos para soportar los datos de clase de usuario concretos; el comportamiento preciso de este nivel varía para adecuarse a las necesidades de transmisión de un sistema concreto. En el nivel de adaptación se efectúa la corrección de error y sincronización de principio a fin, y la segmentación y el reensamblado de los datos de usuario en células ATM.

Una red ATM corriente puede contener muchos conmutadores, cada uno de los cuales lleva asociado una política de entrada/salida de células; en la mayoría de los conmutadores comerciales se empleaba la política FIFO básica, aunque ahora algunos proporcionan una política basada en prioridad. Dentro de cada conmutador concreto, se puede establecer un conjunto de conexio-

nes entre ciertos puertos de entrada y de salida del conmutador según una tabla de conexión. La aplicación de la política de colas de células puede hacerse en los puertos de entrada y/o en los puertos de salida.

Una aproximación habitual tomada en una solución ATM de tiempo real puede ser:

- Predefinir todos los requisitos de canales virtuales (VC).
- Predefinir la ruta y, consecuentemente, los recursos de red asignados a cada VC.
- Controlar el ancho de banda total requerido por cada VC en cada recurso de red a través de los que se encamina.
- Calcular los retrasos resultantes y, por tanto, evaluar la realizabilidad de cierta asignación particular de VC sobre el hardware de red.

Es preciso controlar el uso del ancho de banda en cada VC para evitar la congestión de la red. El motivo de ello es proporcionar un comportamiento predecible para cada VC, y por tanto para toda la red en conjunto. Cuando se presenta cierta congestión, se corre el riesgo de perder células ATM (bajo las reglas habituales del nivel de protocolo ATM), y los retrasos para el peor caso se vuelven difíciles de predecir a menos que seamos excesivamente pesimista (permitiendo la detección de células perdidas y posibles retransmisiones).

14.7.3 Planificación holística

Cualquier sistema distribuido de tiempo real razonablemente grande puede contener decenas de procesadores y dos o tres canales de comunicación distintos. Podrán planificarse tanto los subsistemas de procesamiento como los de comunicación para hacer predecible el comportamiento temporal en el peor caso. La aproximación estática a la asignación facilita este cálculo. Tras analizar cada componente del sistema, será posible combinar las predicciones para comprobar el cumplimiento de los requisitos de temporización globales del sistema (Tindell y Clark, 1994; Palencia Gutierrez y Gonzalez Harbour, 1998; Palencia Gutierrez y Gonzalez Harbour, 1999). Para abordar el problema de la *planificación holística*, es preciso tener en cuenta dos factores importantes.

- ¿Afectará negativamente la variabilidad del comportamiento de un componente sobre el comportamiento de otra parte del sistema?
- ¿Conducirá la simple agregación del comportamiento del peor caso de cada componente a predicciones pesimistas?

La cantidad de variabilidad dependerá de la aproximación a la planificación. Si se emplea una aproximación disparada exclusivamente por tiempo (esto es, sistemas de ejecución cíclica enlazados mediante canales TDMA), no queda mucha holgura para desviaciones sobre el comportamiento repetitivo. Sin embargo, si se emplea una planificación basada en prioridad en los procesadores y en los canales, podrá haber una variación considerable. Considere, por ejemplo, un proceso esporádico generado por la ejecución de un proceso periódico de otro nodo. Por tér-

mino medio, el proceso esporádico se ejecutará al mismo ritmo que el proceso periódico (pongamos que cada 50 ms). Pero el proceso periódico (y el mensaje de comunicación subsiguiente que genera el esporádico) no ocurrirán exactamente en el mismo momento de cada periodo. Pudiera ocurrir que el proceso se ejecutara relativamente tarde en cierta invocación y antes en la próxima. En consecuencia, el esporádico podría haber sido generado a tan sólo 30 ms de su antecesor. Modelar el esporádico como un proceso periódico con un periodo de 50 ms sería incorrecto, y podría conducir a la conclusión falsa de que pueden satisfacerse todos los tiempos límite. Afortunadamente, la variabilidad en su tiempo de generación puede modelarse de modo preciso empleando el análisis de fluctuación (jitter) de generación presentado en la Sección 13.12.2.

Mientras que el análisis de tiempo de respuesta presentado para la planificación monoprocesador es necesario y suficiente (es decir, da una imagen precisa del auténtico comportamiento del procesador), la planificación holística puede pecar de pesimista. Esto ocurre cuando el comportamiento del peor caso de un subsistema implica que en algún otro componente pasará aún peor que su peor caso. A menudo, sólo los estudios de simulación permitirán determinar (estadísticamente) el nivel de pesimismo. Aún es preciso investigar más para determinar exactamente la efectividad de la planificación holística.

Una cuestión final importante sobre la planificación holística es que puede dar cierta ayuda en el problema de la asignación. La asignación estática suele ser la más apropiada. Pero obtener la mejor asignación estática es aún un problema NP-duro. Para este problema se han ideado muchas heurísticas. Aunque, recientemente, también se han aplicado técnicas de búsqueda, como la simulación de condensación (simulated annealing) y los algoritmos genéticos, que han resultado ser de bastante ayuda, y pueden extenderse con facilidad a sistemas replicados para tolerancia a fallos (donde se precisa jugar con asignaciones que asocian réplicas de diferentes componentes).

Resumen

En este capítulo, se ha definido el sistema distribuido como un conjunto de elementos de procesamiento autónomos que cooperan en un cometido común o por una meta común. Alguno de los temas que aparecen cuando se consideran aplicaciones distribuidas plantea cuestiones fundamentales que van más allá de simples aspectos de implementación. Éstas son: el soporte del lenguaje, la fiabilidad, los algoritmos de control distribuidos, y la planificación con tiempos límites.

Soporte del lenguaje

La producción de un sistema software distribuido para su ejecución sobre un sistema hardware distribuido implica: **particionado**: proceso de dividir el sistema en partes (unidades de distribución) adecuadas para su ubicación sobre los elementos de procesamiento del sistema en cuestión; **configuración**: asociar los componentes particionados con los elementos de procesamiento concretos del sistema en cuestión; **asignación**: proceso real de convertir el sistema configurado en un conjunto de módulos ejecutables y de cargar éstos sobre los elementos de proceso; **ejecución transparente**: ejecutar el software distribuido de modo que el acceso a los recursos remotos sea

independiente de su ubicación (habitualmente mediante una llamada a un procedimiento remoto); y **reconfiguración**: cambiar la situación de un componente software o de un recurso.

Aunque occam2 se diseñó para usarse en entornos distribuidos, es de bastante bajo nivel; la unidad de particionado es el proceso; la configuración se efectúa de modo explícito en el lenguaje, pero no se da soporte ni a la asignación ni a la reconfiguración.

Ada permite agrupar un conjunto de elementos de biblioteca en «particiones» que se comunican mediante llamadas a procedimiento remoto y a objetos remotos. Ni la configuración, ni la asignación ni la reconfiguración se soportan directamente desde el lenguaje.

Java permite la distribución de objetos que se comunican mediante llamadas a procedimientos remotos o mediante *sockets*. Ni la configuración, ni la asignación ni la reconfiguración vienen soportadas directamente por el lenguaje.

CORBA permite objetos distribuidos e interoperatividad entre aplicaciones escritas en diferentes lenguajes para diferentes plataformas mediante software middleware de diferentes fabricantes.

Fiabilidad

Aunque disponer de varios procesadores permite que la aplicación sea tolerante a fallos de procesador, también implica la posible aparición de nuevos tipos de fallos que no se presentan en un sistema monoprocesador centralizado. En concreto, con varios procesadores aparece el concepto de fallo parcial del sistema. Además, el medio de comunicación puede perder, corromper, o cambiar el orden de los mensajes.

La comunicación entre procesos fuera de los límites de una máquina precisa de niveles de protocolos que hagan que sea posible tolerar las condiciones de error transitorias. Se han definido estándares para dar apoyo a estos niveles de protocolos. El modelo de referencia OSI es un modelo estratificado que consta de los niveles de aplicación, presentación, sesión, transporte, red, enlace de datos y físico. Fue desarrollado, en principio, para facilitar la extensibilidad de las redes de área extensa; las redes de área extensa se caracterizaban por su bajo ancho de banda de comunicación con altas tasas de error. TCP/IP es otro modelo de referencia de protocolo dirigido en principio a las redes de área extensa. Sin embargo, la mayoría de los sistemas distribuidos embebidos emplean tecnología de área local y están cerrados al mundo exterior. Las redes de área local se caracterizan por un gran ancho de banda de comunicación con bajas tasas de error. En consecuencia, aunque es posible implementar comunicación entre procesos sobre el nivel del lenguaje empleando la aproximación OSI, el coste práctico suele ser excesivo. Por eso, muchos diseñadores unen los protocolos de comunicación a los requisitos del lenguaje (la aplicación) y el medio de comunicación; el resultado se denomina protocolos ligeros.

Cuando interactúan grupos de procesos, es preciso rutar la comunicación al grupo completo. Un paradigma de comunicación por **multidifusión** proporciona este mecanismo. Es posible diseñar una familia de protocolos de comunicación de grupo en la que cada uno proporcione un mecanismo de comunicación por multidifusión con garantías específicas: la **multidifusión no fiable**

no proporciona garantías sobre la entrega el grupo; en la **multidifusión fiable**, el protocolo intenta entregar los mensajes al grupo lo mejor que puede; en la **multidifusión atómica**, el protocolo garantiza que si un proceso del grupo recibe el mensaje también lo recibirá el resto de miembros del grupo; en la **multidifusión atómica ordenada**, el protocolo, además de garantizar la atomicidad de la multidifusión, también garantiza que todos los miembros del grupo reciben los mensajes de los diferentes emisores en el mismo orden.

Es posible tolerar el fallo de los procesadores mediante redundancia estática o dinámica. La redundancia estática implica la replicación de los componentes de la aplicación sobre diferentes procesadores. El grado de replicación puede variar según la importancia de cada componente concreto. Uno de los problemas de proporcionar tolerancia a fallos de modo transparente a la aplicación es que se vuelve imposible para el programador especificar la ejecución degradada o segura. La alternativa a la redundancia estática es permitir al programador el manejo dinámico de los fallos de los procesadores. Para esto se precisa que el fallo del procesador sea detectado y comunicado al resto de los procesos del sistema; que sea evaluado el daño ocurrido; que, a partir de estos resultados, el software restante pueda consensuar una respuesta y efectuar las acciones precisas para llevar a cabo la respuesta; y que, tan pronto como sea posible, sea reparado el procesador incorrecto y/o su software asociado y el sistema vuelva a su estado normal sin errores. Pocos de los lenguajes de programación de tiempo real proporcionan mecanismos apropiados para tratar con la reconfiguración dinámica tras un fallo de procesador.

Algoritmos de control distribuidos

La presencia de un auténtico paralelismo en una aplicación, junto a la distribución física de los procesadores y la posibilidad de que fallen los procesadores y los enlaces de comunicación, hacen preciso nuevos algoritmos para el control de los recursos. Se han considerado los siguientes algoritmos: ordenamiento de eventos, implementación de almacenamiento estable, y protocolos de «acuerdo bizantino». Muchos de los algoritmos distribuidos presuponen que los procesadores se adhieren al modelo de «fallo-parada»; esto quiere decir que o bien funcionan correctamente, o bien se paran inmediatamente después de fallar.

Planificación del tiempo límite

Por desgracia, el problema general de la asignación dinámica de procesos a procesadores (para obtener los tiempos límites de todo el sistema) es computacionalmente caro. Es pues necesario implementar algún tipo de esquema de asignación menos flexible. Una aproximación rígida es desplegar todos los procesos estáticamente o permitir la migración sólo de los no críticos.

Una vez asignados los procesadores, es necesario planificar el medio de comunicación. TDMA, el paso de testigo temporizado y la planificación basada en prioridad, se muestran apropiados para los protocolos de comunicación de tiempo real. Por último, es preciso verificar los requisitos de temporización de entre principio y fin mediante la consideración de la planificación holística del sistema completo.

Lecturas complementarias

Brown, C. (1994), *Distributed Programming with Unix*, Englewood Cliffs, NJ: Prentice Hall.

Comer, D. E. (1999), *Computer Networks and Internets*, New York: Prentice Hall.

Coulouris, G. F., Dollimore, J., y Kindberg, T. (2000), *Sistemas distribuidos, conceptos y diseño*, tercera edición. Addison-Wesley.

Farley, J. (1998), *Java Distributed Computing*, Sebastopol, CA: O'Reilly.

Harold, E. (1997), *Java Network Programming*, Sebastopol, CA: O'Reilly.

Halsall, F. (1995), *Comunicaciones de datos, redes de computadoras y sistemas abiertos*, segunda edición, Addison-Wesley.

Hoque, R. (1998), *CORBA* 3, Foster City, CA: IDG Books Worldwide.

Horstmann, C. S., y Cornell, G. (2000), *Core Java 2, Volume II – Advanced Features*, Sun Microsystems.

Kopetz, H. (1997), *Real-Time Systems: Design Principles for Distributed Embedded Applications*, New York: Kluwer International.

Lynch, N. (1996), *Distributed Algorithms*, San Mateo, CA: Morgan Kaufmann.

Mullender, S. (ed.) (1993), *Distributed Systems*, segunda edición, Reading, MA: Addison-Wesley.

Tanenbaum, A. S. (1994), *Sistemas operativos distribuidos*, Prentice Hall.

Tanenbaum, A. S. (1998), *Redes de Computadoras*, Prentice Hall.

Ejercicios

14.1 Hable sobre algunas de las *desventajas* de los sistemas distribuidos.

14.2 Ada provee un Anexo para Sistemas de Tiempo Real y un Anexo para Sistemas Distribuidos. Discuta hasta que punto estos dos anexos conforman un Anexo para Sistemas Distribuidos de Tiempo Real.

14.3 Desde el punto de vista de la abstracción de datos, discuta por qué las variables no deberían ser visibles en la interfaz de un objeto remoto.

14.4 Considere las implicaciones de tener llamadas de entrada temporizadas y condicionales en un entorno distribuido en Ada.

14.5 El siguiente proceso en occam2 tiene cinco canales de entrada y tres canales de salida. Todos los enteros recibidos por los canales de entrada son transmitidos a todos los canales de salida:

```
INT I,J,temp:
WHILE TRUE
   ALT I = 1 FOR 5
      in[I] ? temp
         PAR J = 1 FOR 3
            out[J] ! temp
```

Puesto que este proceso tiene una interfaz de cuatro canales, no puede implementarse sobre un único transputer, a menos que sus procesos clientes residan sobre el mismo transputer. Transforme el código de modo que pueda implementarse sobre tres transputers. (Suponga que un transputer sólo dispone de cuatro canales.)

14.6 Compare y contraste los modelos de objetos remotos de Ada, Java y CORBA.

14.7 ¿Hasta donde puede implementarse en Java el servicio de paso de mensajes de CORBA?

14.8 Bosqueje los niveles de comunicación implicados cuando un delegado francés del Consejo de Seguridad de las Naciones Unidas desea conversar con el delegado ruso. Suponga que se dispone de intérpretes que traducen a un idioma común (digamos el inglés), y que se comunican a través de operadores telefónicos. ¿Sigue este esquema de comunicación estratificado el modelo ISO OSI?

14.9 ¿Por qué puede ser diferente la semántica de una invocación a un procedimiento remoto de una llamada normal a un procedimiento?

14.10 Si se empleara el nivel de red OSI para implementar un mecanismo de RPC, ¿debería emplearse un servicio datagrama, o uno de circuito virtual?

14.11 Compare y contraste las aproximaciones de almacenamiento estable y datos replicados a la hora de obtener datos de sistema fiables que sobrevivan a un fallo de procesador.

14.12 Rehaga el «problema de los generales bizantinos» dado en la Sección 14.6.4 de modo que *G*1 sea el traidor, *G*2 concluya la retirada, *G*3 concluya la espera y *G*4 concluya el ataque.

14.13 Puesto que hemos de elegir entre Ethernet y paso de testigo para un subsistema de tiempo real, ¿cuál elegiremos si el objetivo principal es un acceso determinista bajo fuertes cargas?

14.14 Actualice el algoritmo dado en la Sección 10.8.2 para la recuperación de errores hacia delante en un sistema distribuido.

Capítulo 15

Programación de bajo nivel

Entre las principales características de un sistema embebido se encuentra la necesidad de interacción con dispositivos de entrada y salida (E/S) de propósito específico. Desgraciadamente, hay muchos tipos distintos de interfaces y de mecanismos de control de dispositivos. Esto es así, principalmente, porque: los diferentes computadores proporcionan diferentes métodos de interacción con los dispositivos; los diferentes dispositivos disponen de diferentes requisitos de control; y entre dispositivos similares de diferentes fabricantes aparecen interfaces y requisitos de control diferentes.

Para proporcionar un marco conceptual de los mecanismos de alto nivel necesarios para programar dispositivos de bajo nivel, es preciso haber comprendido las funciones básicas de E/S del hardware. Por eso, este capítulo aborda primero dichos mecanismos, para después tratar con las características de los lenguajes en general, y finalmente entrar en detalles para lenguajes concretos.

Los sistemas embebidos suelen disponer de una cantidad de memoria limitada. Por tanto, el programador debe cerciorarse de que el compilador asigna justo la memoria necesaria para la tarea que maneja. Además, habrá que controlar cuidadosamente la asignación dinámica de la memoria para garantizar que el código de tiempo crítico no se vea demorado por la inoportuna ejecución de algunas funciones del sistema, tales como el recolector de basura.

15.1 Mecanismos hardware de entrada/salida

En lo concerniente a la entrada y salida de dispositivos, hay dos clases generales de arquitecturas: la primera dispone de buses lógicos diferentes para la memoria y para E/S; en la segunda, la memoria y los dispositivos de E/S se asientan sobre el mismo bus lógico. Ambas se representan de forma esquemática en las Figuras 15.1 y 15.2.

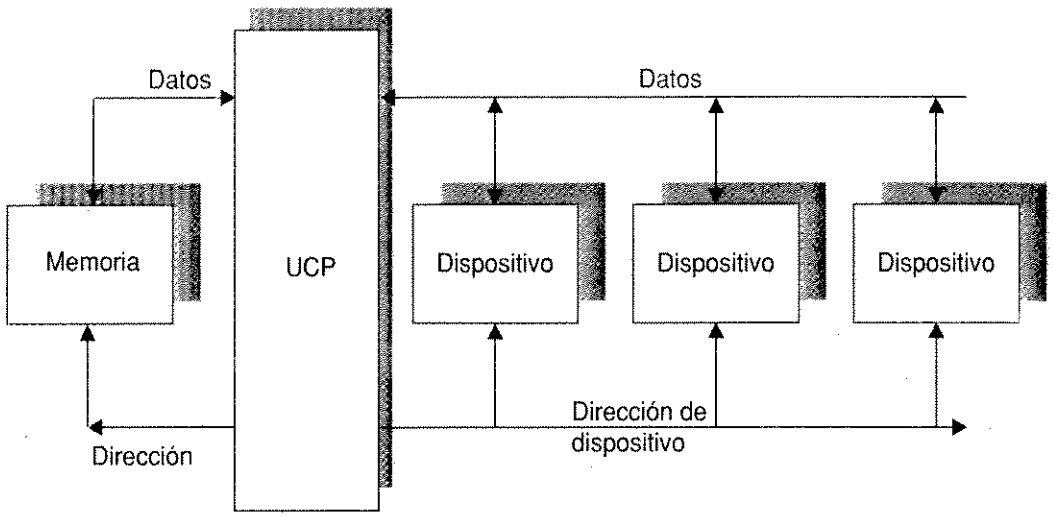


Figura 15.1. Arquitectura con buses separados.

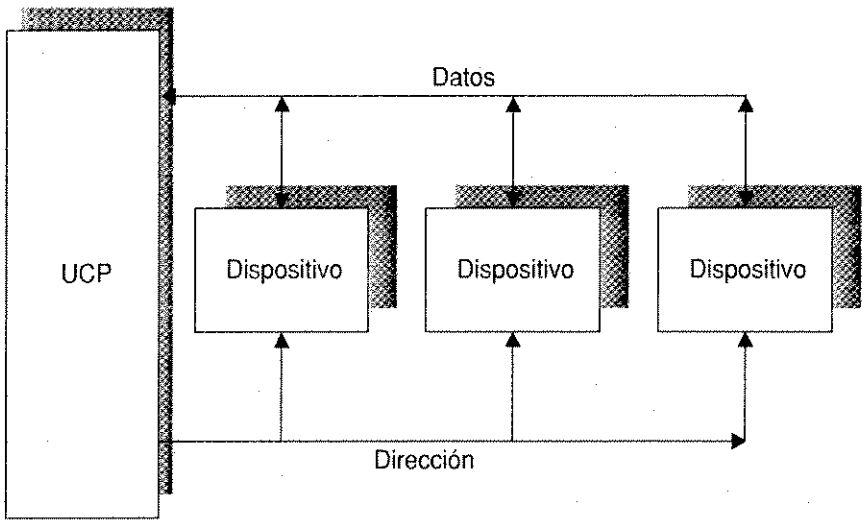


Figura 15.2. Arquitectura sobre memoria.

La interfaz física de un dispositivo suele efectuarse mediante un conjunto de registros. Con buses para memoria y para dispositivos de E/S separados lógicamente, el computador precisa de dos tipos de instrucciones de ensamblado: unas para acceder a la memoria y otras para acceder a los registros del dispositivo. Las últimas suelen tener el siguiente aspecto:

```
IN AC, PUERTO
OUT AC, PUERTO
```

donde `IN` lee los contenidos del registro del dispositivo identificado por `PUERTO` sobre el acumulador `AC`, y `OUT` escribe los contenidos del acumulador sobre el registro del dispositivo. (Con el término `PUERTO`, se hace referencia a una dirección del bus de E/S del dispositivo.) Puede haber también otras instrucciones adicionales para consultar el estatus del dispositivo. El Intel 486 y el Pentium son ejemplos de una arquitectura que permite el acceso a los dispositivos empleando instrucciones especiales.

Si los dispositivos se encuentran sobre el mismo bus lógico, ciertas direcciones indicarán una posición de memoria y otras permitirán acceder a un dispositivo. Esta aproximación se conoce como **E/S sobre memoria** (memory mapped I/O). El Motorola M68000 y la gama de computadores PowerPC disponen de E/S sobre memoria.

Para controlar las operaciones del dispositivo (por ejemplo, inicializar el dispositivo y prepararlo para transferir datos) y la transferencia de datos (por ejemplo, iniciar o efectuar la transferencia de datos), es preciso interactuar con aquél. Hay que destacar dos tipos de mecanismos generales para efectuar y controlar la E/S: mecanismos de control dirigido por estatus, y mecanismos de control dirigidos por interrupción.

15.1.1 Dirigido por estatus

En este tipo de mecanismo de control, cada programa realiza comprobaciones explícitas para determinar el estatus de un dispositivo dado. Una vez determinado el estatus del dispositivo, el programa podrá realizar las acciones pertinentes. Habitualmente, hay tres clases de instrucciones hardware que soportan este tipo de mecanismo:

- Operaciones de test, que permiten al programa determinar el estatus de un dispositivo dado.
- Operaciones de control, que indican al dispositivo que realice operaciones no relacionadas con transferencias (como, por ejemplo, posicionar las cabezas de lectura de un disco).
- Operaciones de E/S, que realizan la transferencia real de los datos entre el dispositivo y la unidad central de procesamiento (UCP).

Aunque la E/S dirigida por estatus era lo normal no hace muchos años, dado su bajo coste, hoy en día, como consecuencia del continuo descenso del precio del hardware, la mayoría de los dispositivos funcionan dirigidos por interrupción. Sin embargo, lógicamente, se pueden deshabilitar las interrupciones y emplear en su lugar métodos de sondeo del estatus de los dispositivos. Por otra parte, las interrupciones aumentan el grado de no determinismo de los sistemas de tiempo real, y de hecho a veces se prohibe su uso por razones de seguridad; por ejemplo, cuando no es posible acotar el número de interrupciones.

15.1.2 Dirigido por interrupción

Incluso con el mecanismo dirigido por interrupción hay muchas variaciones posibles, dependiendo de cómo deban iniciarse y controlarse las transferencias. Tres variantes de este mecanismo son: controlado por programa, iniciado por programa, y controlado por programa de canal.

Dirigido por interrupción y controlado por programa

Aquí, un dispositivo pide una interrupción como resultado de encontrar ciertos eventos (por ejemplo, la llegada de datos). Cuando se reconoce la petición, el procesador suspende el proceso en ejecución e invoca un proceso concreto de manejo de la interrupción, el cual efectúa las acciones

adecuadas en repuesta a la interrupción. Cuando el proceso de manejo de la interrupción ha realizado su cometido, se restaura el estado del procesador a su situación anterior a la interrupción, y se devuelve el control del procesador al proceso suspendido.

Dirigido por interrupción e iniciado por programa

Este tipo de mecanismo de entrada/salida se suele denominar **acceso directo a memoria** (DMA; Direct Memory Access). Se sitúa un dispositivo DMA entre el dispositivo de entrada/salida y la memoria principal. El dispositivo DMA asume las funciones del procesador en la transferencia de los datos entre el dispositivo de E/S y la memoria. Aunque la E/S es iniciada por el programa, el dispositivo DMA controla la transferencia real de los datos (un bloque cada vez). Para cada fragmento de datos a transferir, se efectúa una petición de un ciclo de memoria desde el dispositivo DMA, realizándose la transferencia cuando se accede a dicha petición. Tras completarse la transferencia de todo el bloque, se generará una interrupción de *transferencia completa* desde el dispositivo. Esto se gestiona por el mecanismo dirigido por interrupción y controlado por programa.

La expresión **robo de ciclo** alude al impacto de los dispositivos DMA sobre el acceso a la memoria. El robo de ciclo puede acarrear en un comportamiento no determinista, y hace muy difícil calcular el tiempo de ejecución de un programa para el peor caso (véase la Sección 13.7).

Dirigido por interrupción y controlado por programa de canal

La entrada/salida controlada por programa de canal extiende el concepto de entrada/salida iniciada por programa, al eliminar gran parte de la carga del procesador central en la gestión de los dispositivos de entrada/salida. El mecanismo consta de tres elementos principales: el canal hardware y los dispositivos conectados, el programa de canal, también llamado «guión» *(script)*, y las instrucciones de entrada/salida.

Las operaciones del hardware del canal incluyen las de los dispositivos DMA indicados anteriormente. Además, el canal dirige las operaciones de control del dispositivo siguiendo las indicaciones de programa del canal. La ejecución de los programas de canal se inicia desde la aplicación. Una vez que se ha indicado al canal que ejecute cierto programa de canal, el canal seleccionado y el dispositivo procederán por cuenta propia hasta que finalice el programa de canal concreto u ocurra alguna condición excepcional. Los programas de canal suelen constar de una o más palabras de control de canal, que son decodificadas y ejecutadas por turno por el canal.

En muchos aspectos, un canal hardware puede verse como un procesador autónomo que comparte memoria con el procesador central. Su impacto sobre los tiempos de acceso a memoria puede ser impredecible (como con los dispositivos DMA).

15.1.3 Elementos necesarios para los dispositivos dirigidos por interrupción

Como ha podido comprobarse en la sección anterior, las interrupciones juegan un papel importante en el control de la entrada/salida. Permiten que la entrada/salida se efectúe asíncronamen-

te, evitando así, la «espera ocupada» o activa (busy waiting), o la constante comprobación de estatus, que sólo es necesaria si se emplea un mecanismo puramente basado en el control del estatus.

Para permitir la entrada y la salida dirigidas por interrupción, se precisa de los siguientes mecanismos.

Mecanismos de cambio de contexto

Cuando ocurre una interrupción, debe preservarse el estado actual del procesador y activarse la rutina de servicio correspondiente. Una vez servida la interrupción, habrá que retomar el proceso y continuar su ejecución. Otra posibilidad es que el planificador seleccione un nuevo proceso como consecuencia de la interrupción. El procedimiento completo se conoce como **cambio de contexto** (context switching), y sus acciones pueden resumirse como sigue:

- Preservación del estado del procesador inmediatamente anterior a la aparición de la interrupción.
- Establecimiento del procesador en el estado adecuado para procesar la interrupción.
- Restauración del estado del proceso suspendido, tras completar el procesamiento de la interrupción.

El estado de un proceso en ejecución sobre un procesador consta de:

- La posición en memoria de la instrucción actual (o la siguiente) en la secuencia de ejecución.
- La información de estatus del programa (que podrá contener información relativa al modo de procesamiento, a la prioridad actual, a la protección de memoria, a las interrupciones permitidas, y otras).
- Los contenidos de los registros programables.

El tipo de cambio de contexto provisto se caracteriza por la capacidad del hardware para preservar y restaurar el estado del proceso. Se pueden distinguir tres niveles de cambio de contexto:

- Básico: sólo se guarda y se actualiza el contador de programa.
- Parcial: se guardan el contador de programa y los registros de estatus del programa para alojar los nuevos valores.
- Completo: se guarda el contexto completo del proceso y se aloja uno nuevo.

Dependiendo del grado necesario de preservación del estado, puede que haya que complementar las acciones del hardware con soporte explícito de software. Por ejemplo, puede que un cambio parcial de contexto sea adecuado para un modelo de manejo de interrupciones que contemple al manejador como un procedimiento o subrutina. Sin embargo, si el manejador es visto como un proceso separado con sus propias áreas de pila y de datos, será necesario un manejador

de bajo nivel que se haga cargo completamente del cambio de contexto. Si, por otra parte, el hardware «puede» con un cambio de contexto completo, dicho manejador de bajo nivel no será necesario. La mayoría de los procesadores sólo proporcionan un cambio parcial de contexto. Sin embargo, el procesador ARM proporciona una interrupción rápida, donde se guardan también algunos de los registros de propósito general.

Hay que hacer constar que algunos procesadores modernos permiten la búsqueda, decodificación y ejecución de instrucciones en paralelo. Algunos incluso permiten la ejecución de instrucciones al margen de la secuencia especificada en el programa. Este capítulo parte de la premisa de que las interrupciones son **precisas** (Walker y Cragon, 1995), en el sentido de que, cuando se ejecuta un manejador de interrupción:

- Todas las instrucciones que ha tomado el procesador antes de la instrucción interrumpida han finalizado su ejecución y han modificado correctamente el estado del programa.
- Aquellas instrucciones tomadas por el procesador tras la instrucción interrumpida no han sido ejecutadas y no han modificado el estado del programa.
- Si fue la misma instrucción la que causó la interrupción (por ejemplo, provocando una violación de memoria), dicha instrucción ha sido ejecutada totalmente, o no se ha ejecutado en absoluto.

Si una interrupción no es precisa, se supone que la recuperación es transparente al software de manejo de la interrupción. Véase Walker y Cragon (1995) para una taxonomía de manejo de interrupciones.

Identificación del dispositivo de la interrupción

Los dispositivos difieren en la forma como deben ser controlados; consecuentemente, requerirán diferentes rutinas de manejo de interrupciones. Para invocar el manejador apropiado, deberá existir algún modo de identificar el dispositivo de la interrupción. Se pueden distinguir cuatro tipos de mecanismos para la identificación del dispositivo de la interrupción: vectorizados, por estatus, por sondeo, y por primitiva de lenguaje de alto nivel.

- Un mecanismo **vectorizado** para la identificación del dispositivo consta de un conjunto de posiciones de memoria dedicadas (generalmente contiguas) llamado tabla de **vector de interrupción**, y una correspondencia mediante hardware de las direcciones de los dispositivos con el vector de interrupción. Puede utilizarse un vector de interrupción para cada dispositivo concreto, o puede ser compartido por varios dispositivos. El programador deberá vincular explícitamente cada posición concreta del vector de interrupción con una rutina de servicio de interrupción. Esto podrá hacerse cargando en la posición del vector la dirección de la rutina de servicio, o la dirección de una instrucción que efectúa una bifurcación a la rutina requerida. Con ello, se consigue enlazar directamente la rutina de servicio con cierta posición del vector de interrupción, el cual, a su vez, enlaza indirectamente con uno o varios dispositivos. Así, invocar y ejecutar cierta rutina de servicio supone implícitamente haber identificado el dispositivo que generó la interrupción.

- El mecanismo basado en **estatus** para la identificación del dispositivo de la interrupción se emplea en configuraciones donde hay varios dispositivos conectados a un único controlador de dispositivo, los cuales no poseen vectores de interrupción separados. También se emplea cuando todas las interrupciones se manejan inicialmente desde una rutina general de servicio. Con este mecanismo, cada interrupción tiene una palabra de estatus asociada que especifica el dispositivo que causó la interrupción y la razón de la misma (entre otros datos). La información de estatus puede ser cargada automáticamente por el hardware en una posición de memoria dedicada para una interrupción dada o para una clase de interrupciones, o podrá obtenerse mediante alguna instrucción apropiada.

- El mecanismo de identificación del dispositivo por **sondeo** (polling) es el más simple de todos. Cuando aparece una interrupción, se invoca una rutina general de servicio de interrupción para manejarla. En ella, se interroga el estatus de cada dispositivo para determinar cuál de ellos solicitó la interrupción. Una vez identificado el dispositivo de la interrupción, se sirve ésta de la manera apropiada.

- En algunas arquitecturas modernas, el manejo de la interrupción se asocia directamente a una **primitiva del lenguaje de alto nivel**. Con estos sistemas, la interrupción se contempla como un mensaje de sincronización proveniente del canal asociado activado, que a su vez identifica el dispositivo concreto.

Identificación de la interrupción

Una vez identificado el dispositivo, la rutina de manejo de interrupción apropiada deberá determinar la causa de la misma. En general, la causa puede determinarse por la información de estatus del dispositivo, o bien la misma interrupción la señala si el dispositivo dispone de diferentes interrupciones, o diferentes canales.

Control de interrupciones

Una vez encendido o inicializado el dispositivo, habrá que ignorar sus interrupciones, a menos que hayan sido habilitadas. Este control (habilitación/inhabilitación) de interrupciones puede efectuarse mediante los siguientes mecanismos de control:

- Los mecanismos de **control por estatus de interrupción** proporcionan indicadores (flags), bien por medio de una tabla de estado o bien a través de dispositivo y palabras de estatus de programa, para habilitar e inhabilitar las interrupciones. Para acceder (y modificar) dichos indicadores, se procede mediante instrucciones de bit normales o mediante instrucciones especiales de comprobación de bit.

- Los mecanismos de **control por máscara de interrupción** asocian cada interrupción del dispositivo con un bit concreto de una palabra. Si el bit está activo (a uno) se habilita la interrupción; si está inactivo (a cero) la interrupción está inhabilitada. La palabra de máscara de interrupción puede direccionarse mediante instrucciones de bit (u orientadas a palabras) normales, o puede que haya que utilizar instrucciones especiales de enmascaramiento de interrupciones.

- Los mecanismos de **control por nivel de interrupción** asocian un nivel a cada dispositivo. El nivel actual del procesador determina qué dispositivos podrán o no interrumpir. Sólo aquellos dispositivos con un nivel lógico mayor podrán hacerlo. Cuando se activa el nivel lógico más alto, sólo se permiten aquellas interrupciones que no pueden ser inhabilitadas (por ejemplo, un fallo de tensión en la red). Esto no inhabilita explícitamente las interrupciones, por lo que podrán seguir generandose interrupciones en niveles inferiores al nivel del procesador. Por ello, no será necesario rehabilitarlas cuando el nivel del procesador baje.

Control de prioridad

Algunos dispositivos presentan mayor urgencia que otros, y por ello suele haber algún mecanismo de prioridad de interrupciones. Este mecanismo puede ser estático o dinámico, y suele estar relacionado con los mecanismos de control de interrupciones de los dispositivos y con los niveles de prioridad del procesador.

15.1.4 Un ejemplo simple de sistema de E/S

Para ilustrar los diversos componentes de un sistema de E/S, se describe una máquina sencilla. Se basa superficialmente en la serie Motorola 68000.

Cada dispositivo soportado por la máquina dispone de tantos tipos de registro como sean necesarios para su operación. Estos registros se asocian a posiciones de memoria. Los más corrientes son los registros **de control y de estatus**, que alojan toda la información referente al estatus del dispositivo, y permiten habilitar e inhabilitar las interrupciones del mismo. Los registros **de búfer de datos** actúan como registros búfer donde almacenar temporalmente los datos que se transfieren entre la máquina y el dispositivo.

Un registro de control y de estatus, desde el punto de vista del computador, suele tener la siguiente estructura:

```
bits
  15 - 12  : Errores                    -- indica error en un dispositivo
  11       : Ocupado                    -- indica dispositivo ocupado
  10 -  8  : Selección de unidad        -- para controlar más de un
                                        -- dispositivo
   7       : Hecho/preparado            -- E/S hecha o dispositivo listo
   6       : Habilita interrupción      -- habilita las interrupciones
   5 -  3  : Reservado                  -- reservado para un uso futuro
   2 -  1  : Función del dispositivo    -- indica la función requerida
   0       : Habilita dispositivo       -- habilita el dispositivo
```

Una estructura habitual para un registro de búfer de datos de un dispositivo orientado a carácter es:

```
bits
  15 -  8  : Sin utilizar
   7 -  0  : Datos
```

El bit 0 es el bit menos significativo del registro.

Un dispositivo puede tener más de uno de estos registros (su número concreto depende de la naturaleza del dispositivo). Por ejemplo, cierta interfaz paralela/temporizador de Motorola tiene 14.

Las interrupciones permiten que los dispositivos notifiquen al procesador que precisan ser servidos, y que se encuentran vectorizados. Ante una interrupción, el procesador guarda el valor actual del contador de programa (PC; program counter) y de la palabra de estatus del procesador (PSW; processor status word) sobre la pila del sistema. El registro PSW contiene, entre otras cosas, la prioridad actual del procesador. Su estructura real se muestra a continuación:

```
bits
  15 - 11 : Información del modo
  10 -  8 : Sin utilizar
   7 -  5 : Prioridad
   4 -  0 : Códigos de condición
```

Los códigos de condición contienen información sobre el resultado de la última operación del procesador.

Los nuevos PC y PSW se cargan desde dos posiciones de memoria consecutivas preasignadas (el vector de interrupción). La primera palabra contiene la dirección de la rutina de servicio de la interrupción, y la segunda contiene el PSW, que incluye la prioridad a la que habrá que manejar la interrupción. Un manejador de interrupción de baja prioridad podrá ser interrumpido por cualquier interrupción de mayor prioridad.

Más adelante, se utilizará este ejemplo de sistema de E/S.

15.2 Requisitos del lenguaje

Como ya se indicó, una de las principales características de un sistema embebido es tener que interactuar con dispositivos de entrada y salida, cada uno con sus características peculiares. La programación de dichos dispositivos ha sido, tradicionalmente, baluarte del programador de lenguaje de ensamblado; aunque cada vez más, ciertos lenguajes (como C, Modula-1, Java, occam2 y Ada) proporcionar algunos mecanismos de alto nivel para este tipo de trabajos de bajo nivel. Esto último, facilita la lectura, escritura y mantenimiento de las rutinas de control de dispositivos y de interrupciones. Sin embargo, perdura la gran dificultad de decidir qué características de un lenguaje de alto nivel son precisas para permitir la programación de manejadores de dispositivos utilizables. Aunque la cuestión aún no está madura, hay dos requisitos importantes: **mecanismos de encapsulamiento** y **modelo abstracto** de manejo de dispositivo.

15.2.1 Mecanismos de encapsulamiento y modularidad

Las interfaces de bajo nivel de los dispositivos dependen necesariamente de la máquina, por lo que no suelen ser portables.[1]

En cualquier sistema software, es importante separar las secciones de código no portables de las portables. Allá donde sea posible, se aconseja encapsular todo el código dependiente de máquina en unidades claramente identificables. Con Modula-1, el código relativo a los dispositivos debe encapsularse en un tipo especial de módulo, denominado **módulo de dispositivo** (device module). En Ada, se emplean los mecanismos de paquete y de tipo protegido. En Java, tanto clases como paquetes son mecanismos apropiados de encapsulamiento. En occam2, el único mecanismo es el procedimiento, y en C el archivo.

15.2.2 Modelos abstractos de manejo de dispositivos

Un dispositivo puede verse como un procesador que efectúa cierta tarea establecida, por lo que el sistema informático aparece como compuesto de varios procesos paralelos. Existen diversos modelos con los cuales el «proceso» del dispositivo podrá comunicarse y sincronizarse con los procesos en ejecución en el procesador principal. Todos ellos deben proporcionar:

(1) **Mecanismos para la representación, direccionamiento y manipulación de los registros de los dispositivos**

Cada registro del dispositivo puede representarse como una variable del programa, un objeto, o incluso un canal de comunicación.

(2) **Una representación adecuada de las interrupciones**

En ella, se pueden encontrar diversas representaciones:

(a) **Procedimiento**

Se contempla la interrupción como una llamada a un procedimiento (en cierto sentido, es un procedimiento remoto invocado por el proceso del dispositivo). Cada comunicación y sincronización requerida deberá ser programada en el procedimiento de manejo de interrupción. El procedimiento no es anidado: sólo podrá accederse al estado global o al estado local de manejador.

(b) **Proceso esporádico**

Se ve la interrupción como una petición de ejecución de cierto proceso. El manejador es un proceso esporádico, y puede acceder tanto a los datos persistentes locales

[1] Aun así, en los últimos años ha habido ciertos intentos de producir una interfaz de controlador uniforme (UDI; Uniform Driver Interface). UDI define una arquitectura y un conjunto de interfaces neutras con respecto al sistema operativo (y al hardware), para su uso entre el controlador del dispositivo y el sistema que lo rodea. Esto permite desarrollar los controladores de los dispositivos independientemente de los sistemas operativos, y así usar el mismo controlador en diversas plataformas hardware y sistemas operativos.

como a los globales (si se dispone de un mecanismo de comunicación por variables compartidas en el modelo de concurrencia).

(c) **Notificación asíncrona**

Se ve la interrupción como una notificación asíncrona dirigida hacia un proceso. El manejador podrá acceder tanto al estado local del proceso como al estado global. Son posibles tanto el modelo de reanudación como el de terminación.

(d) **Sincronización por variable de condición compartida**

Se contempla la interrupción como una sincronización de condición dentro de un mecanismo de sincronización por variable compartida; por ejemplo, una operación de señal sobre un semáforo o una operación «envía» sobre una variable de condición en un monitor. El manejador podrá acceder tanto al estado local del proceso/monitor como al estado global.

(e) **Sincronización basada en mensajes**

Se contempla la interrupción como un mensaje vacío enviado por un canal de comunicación. El proceso receptor podrá acceder al estado local del proceso.

Todas las posibilidades anteriores, excepto la aproximación procedimental, precisan de un cambio de contexto completo al ejecutar el manejador en presencia de un proceso. Si los manejadores cumplen ciertas restricciones, puede optimizarse el procedimiento. Por ejemplo, si el manejador en un modelo de notificación asíncrona presenta una semántica de reanudación y no accede a ningún dato local al proceso, bastaría un cambio parcial de contexto para manejar la interrupción.

No todos estos modelos aparecen en los lenguajes y en los sistemas operativos de tiempo real. El más popular es el modelo procedimental, dado que precisa de muy poco soporte. Los sistemas de tiempo real implementados en C y C++ suelen adoptar este modelo, y representan los registros del dispositivo como variables del programa. Para los sistemas secuenciales, el modelo de eventos asíncronos es idéntico, en la práctica, al modelo procedimental, puesto que sólo se puede interrumpir un proceso y no es necesario identificar el proceso o el evento. El modelo Ada es un híbrido entre el modelo de procedimiento y un modelo de sincronización por variable de condición compartida. La interrupción se corresponde con un procedimiento protegido, y se accede a los registros mediante variables del programa. Java para tiempo real contempla las interrupciones como eventos asíncronos donde el manejador es un objeto planificable. Modula-1 y Euclid de tiempo real asocian interrupciones respectivamente a variables de condición y a semáforos (una vez más, los registros se representan como variables de programa). Por ello, poseen modelos de variable compartida puros. Occam2 ve una interrupción como un mensaje sobre un canal; los registros de los dispositivos se representan también como canales.

En el resto de la sección se consideran en detalle cinco lenguajes: Modula-1, Ada, Java para tiempo real, occam2 y C.

15.3 Modula-1

Modula-1 fue uno de los primeros lenguajes de programación de alto nivel que ofreció utilidades para la programación de controladores de dispositivos.

En Modula-1, la unidad de modularidad y encapsulamiento es el módulo. Para controlar el acceso a los recursos compartidos se emplea un tipo de módulo especial, denominado **módulo de interfaz** (interface module), que posee las características de un monitor. Los procesos interaccionan mediante señales (variables de condición) mediante los operadores `WAIT` (espera), `SEND` (envía) y `AWAITED` (esperada) (véase el Capítulo 8). Para encapsular la interacción con los dispositivos se emplea un tercer tipo de módulo denominado **módulo de dispositivo** (device module). Solamente desde el interior de un módulo de dispositivo será posible utilizar los mecanismos para el manejo de interrupciones.

15.3.1 Direccionamiento y manipulación de registros de dispositivos

Asociar una variable con un registro es bastante sencillo. En Modula-1, esto se expresa añadiendo en la declaración una dirección en octal al nombre de la variable. Por ejemplo, se puede definir un registro de búfer de datos para la sencilla arquitectura de E/S descrita en la Sección 15.1.4 así:

```
var rbd[177562B] char;
```

donde 177562B es la dirección en octal del registro sobre la memoria.

Asignar un carácter al registro de búfer de caracteres es también una cuestión sencilla, dado que el tipo no tiene estructura interna alguna. El registro de control y estatus es más interesante. En Modula-1, sólo es posible aplicar tipos de datos escalares sobre los registros de dispositivos; en consecuencia, aquellos registros con estructura interna se considera que son del tipo predefinido *bits*, cuya definición es:

```
TYPE BITS = ARRAY 0:num_de_bits_en_la_palabra OF BOOLEAN;
```

Las variables de este tipo se empaquetan en una sola palabra. El siguiente código Modula-1 muestra cómo definir un registro de control y estatus sobre la dirección octal `177560B`:

```
VAR rce[177560B] : BITS;
```

Para acceder a los diversos campos del registro, el programador proporciona un índice sobre el array. Por ejemplo, el siguiente código habilita el dispositivo:

```
rce[0] := TRUE;
```

y lo siguiente inhibe las interrupciones:

```
rce[6] := FALSE;
```

En general, estos mecanismos no son lo suficientemente potentes como para manejar de forma adecuada todos los tipos posibles de registros. Con la estructura general del registro de control y estatus dada anteriormente:

```
bits
  15 - 12  : Errores
  11       : Ocupado
  10 -  8  : Selección de unidad
   7       : Hecho/preparado
   6       : Habilita interrupción
   5 -  3  : Reservado
   2 -  1  : Función del dispositivo
   0       : Habilita dispositivo
```

seleccionar una unidad (bits 8-10) mediante valores booleanos resulta complicado. Por ejemplo, las siguientes sentencias seleccionan la unidad del dispositivo con el valor 5.

```
rce[10] := TRUE;
rce[9]  := FALSE;
rce[8]  := TRUE;
```

Merece la pena tener en cuenta que en muchas máquinas se puede asociar más de un registro de dispositivo a la misma dirección física. En consecuencia, habrá varias variables ligadas a la misma posición de memoria. Además, estos registros suelen emplearse sólo para lectura o para escritura. Por estas razones, manipular los registros de dispositivo es una tarea delicada. Si el registro de control y estatus del ejemplo anterior fuera un par de registros asociados a la misma posición de memoria, el código presentado probablemente no surtiría el efecto deseado. Esto es así porque para establecer cierto bit pudiera ser necesario algo de código adicional con el que cargar el valor actual sobre el acumulador de la máquina. Dado que el registro de control sólo permite la escritura, se obtendría el valor del registro de estatus. Por todo ello, se aconseja definir más variables con las que representar los registros del dispositivo, las cuales se manipularán de la forma habitual. Una vez construido el contenido preciso del registro, podrá ser asignado al registro real del dispositivo. Tales variables suelen llamarse **registros de dispositivo en sombra**.

15.3.2 Manejo de interrupciones

Los mecanismos para el manejo de interrupciones en Modula-1 se basan en una noción de dispositivo hardware ideal. Este dispositivo presenta las siguientes propiedades (Wirth, 1977a):

- Se sabe cuantas interrupciones pueden producirse para cada operación del dispositivo.
- Tras cada interrupción, el estatus del dispositivo indica si ocurrirá o no alguna otra interrupción asociada.
- Ninguna interrupción se presenta de modo inesperado.
- Cada dispositivo tiene una posición de interrupción única.

Los mecanismos proporcionados por Modula-1 pueden resumirse en los siguientes puntos:

- Cada dispositivo dispone de un módulo de dispositivo asociado.
- Cada módulo de dispositivo tiene especificada una prioridad hardware en su cabecera tras el nombre del módulo.
- *Todo el código del módulo se ejecuta en la prioridad hardware indicada.*
- Cada interrupción que se maneja en el interior de un módulo de dispositivo concreto requiere un proceso denominado **proceso de dispositivo**.
- Cuando se está ejecutando un proceso de dispositivo, sólo él tiene acceso al módulo (esto es, sólo él posee el bloqueo del monitor utilizando la cota máxima de prioridad especificada en el encabezado del módulo de dispositivo).
- No se permite que un proceso de dispositivo invoque ningún procedimiento no local, y tampoco que envíe señales a otros procesos de dispositivos.
- Cuando un proceso de dispositivo envía una señal, la semántica de la operación «envía» es diferente de la usual en los procesos ordinarios Modula-1; en este caso, no se reanuda el proceso receptor, pero el proceso que lanza la señal continúa. De nuevo, esto garantiza que el proceso no se bloque.
- Las sentencias `WAIT` del interior de los procesos de dispositivo sólo podrán tener rango 1 (el nivel más alto).
- Cada interrupción se considera como una forma de señal. El proceso de dispositivo, sin embargo, en lugar de emitir una petición `WAIT` emite una petición `DOIO` (hazES).
- La dirección del vector con la que interrumpe cada dispositivo se especifica en la cabecera del proceso.
- *Sólo los procesos de dispositivo pueden contener sentencias* `DOIO`.
- Las llamadas `DOIO` y `WAIT` reducen la prioridad del procesador y, en consecuencia, liberan el bloqueo del monitor.
- Sólo puede haber un ejemplar de cada proceso de dispositivo activo.

Como ejemplo, considere un módulo de dispositivo que maneja un reloj de tiempo real para la máquina cuya arquitectura se mostró en la Sección 15.1.4. Al recibir una interrupción, el manejador emite una señal a un proceso que se encuentra a la espera de un pulso (tick) del reloj.

```
DEVICE MODULE rtr[6]; (* prioridad hardware 6 *)

  DEFINE tic;
  VAR tic : SIGNAL;
```

```
  PROCESS reloj[100B];
    VAR rce[177546B] : BITS;
  BEGIN
    rce[0] := TRUE; (* habilita el dispositivo *)
    rce[6] := TRUE; (* habilita las interrupciones *)
    LOOP
      DOIO;
      WHILE AWAITED(tic) DO
        SEND(tic);
      END
    END
  END;
BEGIN
  reloj; (* crea una instancia del proceso reloj *)
END rtr;
```

La cabecera del módulo de dispositivo especifica una prioridad de interrupción de 6, con la cual se ejecutará todo el código del módulo. El valor `100B` en la cabecera del proceso indica que el dispositivo interrumpirá mediante el vector en la posición `100` (en octal). Tras habilitar las interrupciones, el proceso del dispositivo entra en un sencillo bucle de espera por una interrupción (el `DOIO`), y después envía un número suficiente de señales (es decir, una por cada proceso en espera). Observe que el proceso del dispositivo no abandona su acceso de exclusión mutua en el módulo al enviar una señal, sino que continúa hasta ejecutar una sentencia `WAIT` o `DOIO`.

A continuación, se indica cómo se comporta Modula-1 con respecto a las características generales de un dispositivo manejado por interrupciones, que se indicaron en las Secciones 15.1.2 y 15.1.3.

- **Control de dispositivo**: los registros de E/S se representan por variables.
- **Cambio de contexto**: la interrupción provoca un cambio de contexto inmediato al proceso de manejo de la interrupción, que espera usando `DOIO`.
- **Identificación del dispositivo de la interrupción**: la dirección del vector de interrupción se da en el encabezamiento del proceso del dispositivo.
- **Identificación de la interrupción**: en el ejemplo anterior, sólo es posible una interrupción. En otras ocasiones, por contra, deberá comprobarse el registro de estatus del dispositivo para identificar la causa de la interrupción.
- **Control de interrupción**: el control de la interrupción está dirigido por estatus, y proporcionado por un indicador en el registro del dispositivo.
- **Control de prioridad**: la prioridad del dispositivo se indica en el encabezado del módulo del dispositivo. *Todo* el código del módulo va con la misma prioridad –es decir, el módulo del dispositivo tiene una cota de prioridad por hardware, y se ejecuta con el Protocolo de Acotación de Prioridad Inmediato (véase la Sección 13.11)–.

15.3.3 Un ejemplo de un controlador de terminal

Para ilustrar más aún la aproximación de Modula-1 al control de dispositivos, se incluye un sencillo módulo para el manejo de un terminal. El terminal tiene dos componentes: un *display* (pantalla) y un teclado. Cada componente dispone de un registro de control y estatus, un registro búfer, y una interrupción.

Para permitir que otros procedimientos lean y escriban caracteres se dan dos procedimientos. Cada procedimiento accede a búferes limitados que permiten imprimir por adelantado en la entrada y proporcionan un búfer a la salida. Estos búferes deberán incluirse en el módulo del dispositivo, porque los procesos de dispositivo *no pueden* efectuar llamadas a procedimientos no locales. Aunque es posible emplear módulos separados para el display y el teclado, se han combinado para mostrar el hecho de que un módulo de dispositivo puede gestionar más de una interrupción.

```
DEVICE MODULE terminal[4];

  DEFINE leecar, escribecar;

  CONST n=64;                (* talla del búfer *)

  VAR ETCL[177560B]: BITS; (* estado del teclado *)
      BTCL[177562B]: CHAR; (* búfer del teclado *)
      EDIS[177564B]: BITS; (* estado del display *)
      BDIS[177566B]: CHAR; (* búfer del display *)
      in1, in2, out1, out2 : INTEGER;
      n1, n2 : INTEGER;
      nolleno1, nolleno2,•MODULA-1 627
      novacio1, novacio2 : SIGNAL;
      buf1, buf2 : ARRAY 1:n OF CHAR;

  PROCEDURE leecar(VAR car : CHAR);
  BEGIN
     IF n1 = 0 THEN WAIT(novacio1) END;
     car := buf1[out1];
     out1 := (out1 MOD n)+1;
     DEC(n1);
     SEND(nolleno1)
  END leecar;

  PROCEDURE escribecar(car : CHAR);
  BEGIN
     IF n2 = n THEN WAIT(nolleno2) END;
     buf2[in2] := ch;
```

```
      in2 := (in2 MOD n)+1;
      INC(n2);
      SEND(novacio2)
   END escribecar;

   PROCESS manejadorteclado[60B];
   BEGIN
      ETEC[0] := TRUE; (* habilita el dispositivo *)
      LOOP
         IF n1 = n THEN WAIT(nolleno1) END;
         ETEC[6] := TRUE;
         DOIO;
         ETEC[6] := FALSE;
         buf1[in1] := BTEC;
         in1 := (in1 MOD n)+1;
         INC(n1);
         SEND(novacio1)
      END
   END manejadorteclado;

   PROCESS manejadordisplay[64B];
   BEGIN
      EDIS[0] := TRUE; (* habilita el dispositivo *)
      LOOP
         IF n2 = 0 THEN WAIT(novacio2) END;
         BDIS := buf2[out2];
         out2 := (out2 MOD n)+1;
         EDIS[6] := TRUE;
         DOIO;
         EDIS[6] := FALSE;
         DEC(n2);
         SEND(nolleno2)
      END
   END manejadordisplay;

BEGIN
   in1 :=1; in2 := 1;
   out1 := 1; out2 :=1;
   n1 :=0; n2 := 0;
   manejadorteclado;
   manejadordisplay
END terminal;
```

Mecanismos de temporización

Modula-1 no proporciona funcionalidades directas para manejar el tiempo, las cuales deberán proporcionarse desde la aplicación. Esto requiere un módulo de dispositivo que maneje la interrupción de reloj y lance con ella una señal regular (por ejemplo, una vez por segundo). A continuación, se muestra este módulo, que es una versión modificada del definido anteriormente. Se supone que el reloj hardware pulsa cada cincuentésima de segundo.

```
DEVICE MODULE relojhardware[6];
  DEFINE tic;
  VAR tic : SIGNAL;

  PROCESS manejador[100B];
    VAR cuenta : INTEGER;
      registroestatus[177546B] : BITS;
  BEGIN
    cuenta := 0;
    registroestatus[0] := TRUE;
    registroestatus[6] := TRUE;
    LOOP
      DOIO;
      cuenta := (cuenta+1) MOD 50;
      IF cuenta = 0 THEN
        WHILE AWAITED(tic) DO
          SEND(tic)
        END
      END
    END
  END manejador;
BEGIN
  manejador
END relojhardware;
```

En estas condiciones, es fácil proporcionar un módulo de interfaz para gestionar la hora actual (llamémosla `TimeOfDay`).

```
INTERFACE MODULE RelojSistema;
  (* define procedimientos para obtener y establecer la hora actual *)
  DEFINE GetTime, SetTime;

  (* importa el tipo abstacto time, y la señal de tick *)
  USE time, initialise, add, tick;
```

```
  VAR TimeOfDay, unsegundo : time;

  PROCEDURE SetTime(t: time);•MODULA-1 629
  BEGIN
    TimeOfDay := t
  END SetTime;

  PROCEDURE GetTime(VAR t: time);
  BEGIN
    t := TimeOfDay
  END GetTime;

  PROCESS reloj;
  BEGIN
    LOOP
      WAIT(tick);
      addtime(TimeOfDay, unsegundo)
    END
  END reloj;
BEGIN
  inittime(TimeOfDay, 0, 0, 0);
  inittime(unsegundo, 0, 0, 1);
  reloj;
END RelojSistema;
```

Observe que el proceso reloj es lógicamente redundante. El proceso del dispositivo podría incrementar `TimeOfDay` directamente, ahorrando así un cambio de contexto. Sin embargo, en Modula-1 no se permite que un proceso de dispositivo invoque a un procedimiento no local.

Retardo de un proceso

En los sistemas de tiempo real, a menudo es preciso demorar un proceso durante un periodo (véase el Capítulo 12). Aunque Modula-1 no dispone de posibilidades directas para hacerlo, es posible programarlo. La solución se deja como ejercicio para el lector (véase el Ejercicio 15.6).

15.3.4 Problemas con la aproximación al control de dispositivos en Modula-1

Modula-1 fue diseñado para romper el bastión de la programación en lenguaje ensamblador (en lo relativo a la interfaz con los dispositivos). En general, el programa se consideró un éxito, aunque se han formulado ciertas críticas a las funcionalidades que ofrece.

- Modula-1 no permite que un proceso de dispositivo invoque a un procedimiento no local, dado que los procesos de dispositivo deben mantenerse tan reducidos como sea posible, y deben ejecutarse en el nivel de prioridad hardware del dispositivo. La invocación de procedimientos de otros módulos, cuya implementación se desconoce en el proceso, podría derivar en retrasos inaceptables. Es más, podría ser necesario que estos procedimientos se ejecutaran en la prioridad del dispositivo. Lamentablemente, por esta restricción los programadores deberán incorporar funcionalidades extra en un módulo de dispositivo que *no* se encuentra directamente asociado al control del dispositivo (como en el ejemplo del controlador del terminal, donde se incluye un búfer acotado en el modulo del dispositivo), o bien incluir procesos extra donde se espere el envío de una señal desde el proceso del dispositivo. El primer caso podría derivar en módulos de dispositivo muy grandes, y el segundo en un sistema innecesariamente ineficaz.
- Modula-1 sólo permite una instancia de proceso de dispositivo, dado que la cabecera del proceso contiene la información necesaria para asociar el proceso a la interrupción. Esto hace mucho más difícil compartir código entre dispositivos similares; el problema proviene del hecho de no poder efectuar llamadas a procedimientos no locales.
- Modula-1 se diseñó para máquinas con E/S sobre memoria, y en consecuencia es difícil emplear estas posibilidades para la programación de dispositivos controlados mediante instrucciones especiales. Sin embargo, no es difícil pensar en una sencilla extensión para solventar este problema. Young (1982) sugiere la posibilidad de emplear la siguiente notación:

  ```
  VAR x AT PORT 46B : INTEGER;
  ```

  Así, el compilador sería capaz de reconocer cuándo se está empleando un puerto, y podría generar las instrucciones correctas.
- Ya se indicó que muchos registros de dispositivo son de solo lectura o de solo escritura. En Modula-1 no es posible definir variables de sólo lectura o de sólo escritura. Además, habrá que suponer que el compilador no optimizará el acceso a los registros de los dispositivos y los «cacheará» mediante registros locales.

15.4 Ada

Ada dispone de tres formas para sincronizar y comunicar tareas:

- Mediante citas (rendezvous).
- Mediante unidades protegidas.
- Mediante variables compartidas.

En general, Ada presupone que el dispositivo y el programa tienen acceso a registros de dispositivo en memoria compartida, los cuales podrán ser especificados usando sus técnicas de especificación de la representación.

En Ada 83, las interrupciones se representaban mediante llamadas entry generadas por tareas hardware. En la actual versión de Ada, se considera este mecanismo obsoleto, y posiblemente sea eliminado de la próxima revisión del lenguaje. Por ello, no se discutirá en el libro.

El método preferido para el control de dispositivos es encapsular las operaciones del dispositivo en una unidad protegida. Cada interrupción podrá ser manejada por una llamada a un procedimiento protegido.

15.4.1 Direccionamiento y manipulación de los registros de dispositivo

Ada presenta al programador un conjunto completo de posibilidades para la especificación de la implementación de los tipos de datos, las cuales se conocen con el nombre colectivo de **cláusulas de representación**, e indican cómo se correlacionan los tipos del lenguaje con el hardware subyacente. Cada tipo sólo puede representarse de una forma. La representación se especifica independientemente de la estructura lógica del tipo. Por supuesto, la especificación de la representación del tipo es opcional, y puede delegarse en el compilador.

Las cláusulas de representación suponen un compromiso entre estructuras abstractas y concretas. Hay cuatro especificaciones distintas:

(1) **Cláusula de definición de atributos**: permite establecer diversos atributos para un objeto, tarea o subprograma; por ejemplo, el tamaño (en bits) de cada objeto, el alineamiento en memoria, la máxima capacidad de almacenamiento para una tarea, o la posición de un objeto.

(2) **Cláusula de representación de enumeración**: permite indicar los valores internos de los literales de un tipo enumerado.

(3) **Cláusula de representación de registro**: permite asignar desplazamientos y longitudes a cada componente de un registro (`record`) dentro de unidades de almacenamiento aisladas.

(4) **Cláusula at**: éste era el principal mecanismo en Ada 83 para ubicar un objeto en una dirección concreta; hoy en día este mecanismo se considera obsoleto (pueden emplearse atributos), y en consecuencia no se comentará.

Si una implementación no puede obedecer una petición de especificación, el compilador debería rechazar el programa o generar una excepción en ejecución.

Para mostrar el empleo de estos mecanismos, considere las siguientes declaraciones de tipo, que representan un típico registro de control y estatus como el de la sencilla máquina definida anteriormente.

```
type T_Error is (Ninguno, Error_Lectura, Error_Escritura,
                 Fallo_Potencia, Otro);
type T_Funcion is (Lee, Escribe, Busca);
type T_Unidad is new Integer range 0 ..7;

type T_RCE is record
  Errores : T_Error;
```

```
  Ocupado      : Boolean;
  Unidad       : T_Unidad;
  Hecho        : Boolean;
  HabilitaI    : Boolean;
  FunDis       : Function_T;
  HabilitaDis  : Boolean;
end record;
```

Una cláusula de representación de enumeración especificará los códigos internos de los literales del tipo enumerado. Los códigos internos de las funciones del dispositivo anterior podrían ser, por ejemplo:

```
01 -- LEE
10 -- ESCRIBE
11 -- BUSCA
```

Y en Ada, esto se especificará así:

```
type T_Funcion is (Lee, Escribe, Busca);
for T_Funcion use (Lee => 1, Ecribe => 2, Busca => 3);
```

De igual modo, para `T_Error`:

```
type T_Error is (Ninguno, Error_Lectura, Error_Escritura,
                 Fallo_Potencia, Otros);
for T_Error use (Ninguno => 0, Error_Lectura => 1, Error_Escritura => 2,
                 Fallo_Potencia => 3, Otros => 4);
-- observe que, en realidad, ésta es la asignación por defecto
```

Una cláusula de representación de registro especifica la representación del almacenamiento de los registros; es decir, el orden, la posición y el tamaño de sus componentes. Los bits del registro se numeran desde 0; el rango en la cláusula del componente especifica el número de bits que se reservarán.

Por ejemplo, el registro de control y estatus viene dado por:

```
Word : constant := 2; -- número de unidades de memoria por palabra
Bits_In_Word : constant := 16; -- bits por palabra
for T_Rce use
  record
     HabilitaDis at 0*Word range 0..0; -- en la palabra 0 bit 0
     FunDis      at 0*Word range 1..2;
     HabilitaI   at 0*Word range 6..6;
     Hecho       at 0*Word range 7..7;
     Unidad      at 0*Word range 8 .. 10;
     Ocupado     at 0*Word range 11 .. 11;
```

```
    Errores at 0*Word range 12 .. 15;
end record;

for T_Rce'Size use Bits_In_Word; -- la talla de un objeto de tipo T_RCE
for C_Rce'Alignment use Word; -- el objeto debe alinearse por palabras
for C_Rce'Bit_order use Low_Order_First;
-- el primer bit del byte es el menos significativo
```

Un atributo `Size` especifica la cantidad de almacenamiento asociado al tipo. En este ejemplo, el registro es una sola palabra (word) de 16 bits. El atributo `Alignment` indica al compilador que deberá situar los objetos coincidiendo con una frontera que sea un número entero de unidades de almacenamiento (en este caso el límite de cada palabra). El atributo `Bit_order` (orden de bits) indica si la máquina numera como bit 0 el bit más significativo (big endian) o el bit menos significativo (little endian).

Observe que los bits 3, 4 y 5 (que fueron reservados para un uso futuro) no han sido especificados.

Finalmente, es preciso declarar el registro actual y ubicarlo en la posición de memoria precisa. En Ada, `Address` (dirección) es un tipo definido por cada implementación definido en el paquete `System`. El paquete hijo (`System.Storage_Elements.To_Address`) proporciona una función para convertir un valor entero al tipo `Address`.

```
Trce : T_Rce;
for Trce'Address use System.Storage_Elements.To_Address(8#177566#);
```

Después de haber construido la representación abstracta de los datos del registro, y ubicado apropiadamente la variable definida en su posición correcta, podrá manipularse el registro hardware mediante asignaciones sobre esta variable:

```
Trce := (HabilitaDis => True, FunDis => Lee,
         HabilitaI => True, Hecho => False,
         Unidad => 4, Errores => Ninguno);
```

El uso de este registro agregado asume que se le asignará un valor «de una vez» para el registro completo . Para asegurar que `FunDis` no se establece antes que el resto de campos del registro, será necesario emplear un registro transitorio de control (en la sombra):

```
Rce_Temp : T_Rce;
```

Es en este registro temporal donde se asignan los valores de control, y es el que se copia sobre la variable del registro real:

```
Trce := Rce_Temp;
```

En la mayoría de los casos, el código para esta asignación asegurará que el registro de control completo se actualiza en una sola acción. Si quedara alguna duda, podrá utilizarse el pragma

`Atomic` (que instruye al compilador para que genere la actualización como una sola operación o produzca un mensaje de error).

Tras la finalización de la operación de E/S, el dispositivo mismo podrá alterar los valores del registro; esto se señala en el programa como cambios en los valores de las componentes del registro:

```
if Trce.Errores = Error_Lectura then
  raise Error_Disco;
end if;
```

El objeto `Trce` es pues un conjunto de variables compartidas que se comparten entre la tarea de control del dispositivo y el dispositivo mismo. Es necesario que haya exclusión mutua entre estos dos procesos concurrentes (y paralelos) para proporcionar fiabilidad y prestaciones. En Ada, esto se realiza mediante un objeto protegido.

15.4.2 Manejo de interrupciones

Ada define el siguiente modelo para una interrupción:

- Una interrupción hardware representa una clase de eventos que son detectados por el hardware o por el sistema software.
- La **ocurrencia** de una interrupción se compone de su **generación** y su **entrega**.
- La generación de una interrupción es el evento del hardware o software subyacente que hace disponible la interrupción al programa.
- La entrega es la acción que invoca cierta parte del programa (llamada manejador de interrupción) en respuesta a la ocurrencia de la interrupción. Entre la generación de la interrupción y su entrega, se dice que la interrupción se encuentra **pendiente**. El manejador se invoca una sola vez para cada entrega de la interrupción. La **latencia** de una interrupción es el tiempo que ésta pasa en el estado pendiente.
- Mientras se está gestionando una interrupción, se **bloquean** las demás posibles interrupciones con el mismo origen; se impide la entrega de cualquier otra ocurrencia futura de la interrupción. En general, dependerá del dispositivo el que una interrupción bloqueada permanezca pendiente o se pierda.
- Ciertas interrupciones se encuentran **reservadas**. No se permite que el programador proporcione un manejador para una interrupción reservada. Generalmente, una interrupción reservada es manejada directamente por el sistema de soporte de ejecución de Ada (por ejemplo, una interrupción de reloj empleada para implementar la sentencia `delay`).
- A cada interrupción no reservada se le asigna un manejador por defecto por parte del sistema de soporte de ejecución.

Manejo de interrupciones mediante procedimientos protegidos

En Ada, un manejador se representa principalmente como un procedimiento protegido sin parámetros. Cada interrupción tiene un identificador discreto único soportado por el sistema. La forma en que se representa este identificador único depende de la implementación; pudiera ser, por ejemplo, la dirección del vector de interrupción hardware asociado con la interrupción. En el caso de que Ada se encuentre implementado sobre un sistema operativo que permita señales, cada señal se corresponderá con un identificador de interrupción concreto. Esto permite programar los manejadores de señales en el lenguaje.

La identificación de los procedimientos de manejo de interrupciones protegidos se efectúa mediante uno de estos dos pragmas:

```
pragma Attach_Handler(Nombre_Manejador, Expresion);
   -- Podrá aparecer en la especificación o el cuerpo de un objeto
   -- protegido del nivel de biblioteca y permite la asociación estática
   -- de un manejador dado con la interrupción identificada por la
   -- expresión; el manejador queda vinculado cuando se crea el objeto
   -- protegido.
   -- Genera Program_Error:
   --       (a) cuando se crea el objeto protegido y la interrupción se
   --           encuentra reservada,
   --       (b) si la interrupción ya posee un manejador programado
   --           previamente, o
   --       (c) si cualquier prioridad acotada definida no está en el
   --           rango Ada.Interrupt_Priority.

pragma Interrupt_Handler(Nombre_Manejador);
   -- Puede aparecer en la especificación de un objeto protegido
   -- del nivel de biblioteca y permite la asociación dinámica del
   -- procedimiento sin parámetros indicado como manejador de una
   -- o más interrupciones. Los objetos creados a partir de tal
   -- tipo protegidodeberán ser del nivel de biblioteca.
```

El Programa 15.1 define el soporte del Anexo para la Programación de Sistemas para la identificación de interrupciones y el enlace dinámico de manejadores. En todos los casos en los que se genere `Program_Error`, no se cambiará el manejador actualmente enlazado.

Debería tenerse en cuenta que la función `Reference` proporciona los mecanismos mediante los cuales se soportan las entradas de tarea de interrupciones. Como se mencionó anteriormente, este modelo de manejo de interrupciones se considera obsoleto, y no debiera usarse.

Es posible que una implementación también permita la asociación de nombres a interrupciones mediante el Programa 15.2; esto se usará en los siguientes ejemplos.

Programa 15.1. El paquete Ada.Interrupts.

```
package Ada.Interrupts is
   type Interrupt_Id is definido_por_implementación; -- debe ser discreto
   type Parameterless_Handler is access protected procedure;

   function Is_Reserved(Interrupcion : Interrupt_Id) return Boolean;
      -- Devuelve True si la Interrupcion está reservada,
      -- en caso contrario devuelve False.
   function Is_Attached(Interrupcion : Interrupt_Id) return Boolean;
      -- Devuelve True si la Interrupción se encuentra vinculada a un
      -- manejador, en caso contrario devuelve False.
      -- Genera Program_Error si la interrupcion está reservada.
   function Current_Handler(Interrupcion : Interrupt_Id)
                            return Parameterless_Handler;
      -- Devuelve una variable de acceso al manejador actual de
      -- la Interrupcion. Si no se ha vinculado un manejador de usuario,
      -- se devuelve un valor que representa el manejador por defecto.
      -- Genera Program_Error si la Interrupcion está reservada.
   procedure Attach_Handler(Nuevo_Manejador : Parameterless_Handler;
                            Interrupcion : Interrupt_Id);
      -- Asigna el Nuevo_Manejador como manejador actual.
      -- Si el Nuevo_Manejador es null, se vuelve
      -- al manejador por defecto.
      -- Genera Program_Error:
      --    (a) si el objeto protegido asociado con el
      --       Nuevo_Manejador no ha sido identificado con
      --       pragma Interrupt_Handler,
      --    (b) si la Interrupción ya está reservada,
      --    (c) si el manejador actual fue vinculado estáticamente
      --       usando pragma Attach_Handler.
   procedure Exchange_Handler(
             Viejo_Manejador : out Parameterless_Handler;
             Nuevo_Manejador : Parameterless_Handler;
             Interrupcion : Interrupt_Id);
      -- Asigna el Nuevo_Manejador como manejador actual de la
      -- Interrupción actual y devuelve el manejador anterior en
      -- Viejo_Manejador.
      -- Si el Nuevo_Manejador es null, se vuelve
      -- al manejador por defecto.
      -- Genera Program_Error:
      --    (a) si el objeto protegido asociado con el
      --       Nuevo_Manejador no ha sido identificado con
      --       pragma Interrupt_Handler,
```

(Continuación)

```
      --    (b) si la Interrupción ya está reservada,
      --    (c) si el manejador actual fue vinculado estaticamente
      --       usando pragma Attach_Handler.
   procedure Detach_Handler(Interrupcion : Interrupt_Id);
      -- Repone el manejador por defecto para la Interrupcion dada.
      -- Genera Program_Error:
      --    (a) si la Interrupcion esta reservada,
      --    (b) si el manejador actual fue vinculado estaticamente
      --       usando pragma Attach_Handler.
   function Reference(Interrupcion : Interrupt_Id)
            return System.Address;
      -- Devuelve una Direccion que puede usarse para vincular
      -- una entrada de tarea a una Interrupcion mediante una
      -- clausula de direccion sobre una entrada.
      -- Genera Program_Error si no se puede vincular la
      -- Interrupcion de esta manera.
private
   ... -- sin especificar en el lenguaje
end Ada.Interrupts;
```

15.4.3 Un ejemplo sencillo de controlador

Un tipo usual de equipamiento contenido en un sistema embebido es el convertidor analógico-digital (ADC; analogue to digital converter). Las muestras del convertidor son factores del entorno, como la temperatura o la presión; el convertidor, traduce las medidas que recibe, generalmente del orden de milivoltios, y devuelve valores enteros escalados sobre un registro hardware. Considere un sencillo convertidor que dispone de un registro de resultados de 16 bits en la dirección hardware 8#150000#, y un registro de control en la dirección 8#150002#. El computador es una máquina de 16 bits, y el registro de control se estructura como sigue:

Bit	Nombre	Significado
0	Inicia A/D	Poner a 1 para iniciar una conversión.
6	Habilita/Inhibe Interrupciones	Poner a 1 para habilitar las interrupciones.
7	Hecho	A 1 indica que la conversión está completada.
8-13	Canal	El convertidor tiene 64 entradas analógicas, este valor indica el canal que se precisa.
15	Error	A 1 el convertidor indica un fallo en el dispositivo.

Programa 15.2. Paquete `Ada.Interrupts.Names`.

```
package Ada.Interrupts.Names is
   definido_por_implementación : constant Interrupt_Id := definido_por_implementación;
   ...
   definido_por_implementación : constant Interrupt_Id := definido_por_implementación;
private
   ... -- sin especificar en el lenguaje
end Ada.Interrupts.Names;
```

El controlador para este ADC se estructura como un tipo protegido dentro de un paquete, de modo que la interrupción que genera pueda procesarse como una llamada a un procedimiento protegido, y así poder atender a más de un ADC:

```
package Manejador_Dispositivo_ADC is
  Medicion_Max : constant := (2**16)-1;
  type Canal is range 0..63;
  subtype Medicion is Integer range 0..Medicion_Max;
  procedure Lee(Ca: Canal; M : out Medicion);
    -- potencialmente bloqueante
  Error_Conversion : exception;

private
   for Canal'Size use 6;
   -- indica que sólo se deben usar seis bits
end Manejador_Dispositivo_ADC;
```

Para cualquier petición, el controlador efectuará tres intentos antes de generar una excepción. El cuerpo del paquete continúa como sigue:

```
with Ada.Interrupts.Names; use Ada.Interrupts;
with System; use System;
with System.Storage_Elements; use System.Storage_Elements;
package body Manejador_Dispositivo_ADC is
  Bits_In_Word : constant := 16;
  Word : constant := 2; -- bytes en cada Word (palabra)
  type Indicador is (Inactivo, Activo);

  type Registro_Control is
  record
    Inicia_Ad : Indicador;
    HabilitaI : Indicador;
```

```
   Hecho : Indicador;
   Ca : Canal;
   Error : Indicador;
end record;

for Registro_Control use
   -- especifica el formato del registro de control
   record
      Inicia_Ad at 0*Word range 0..0;
      HabilitaI at 0*Word range 6..6;
      Hecho     at 0*Word range 7..7;
      Ca        at 0*Word range 8..13;
      Error     at 0*Word range 15..15;
end record;

for Registro_Control'Size use Bits_In_Word;
   -- la talla del registro es de 16 bits
for Registro_Control'Alignment use Word;
   -- en una frontera de palabra (word)
for Registro_Control'Bit_order use Low_Order_First;

type Registro_Datos is range 0 .. Medicion_Maxima;
for Registro_Datos'Size use Bits_In_Word;
   -- la talla del registro es de 16 bits

Dir_Reg_Contr : constant Address := To_Address(8#150002#);
Dir_Reg_Datos : constant Address := To_Address(8#150000#);
Prioridad_Adc : constant Interrupt_Priority := 63;
Reg_Control   : aliased Registro_Control;
     -- aliased significa que se usan punteros para acceder a él
for Reg_Control'Address use Dir_Reg_Control;
     -- especifica la dirección del registro de control
Reg_Datos : aliased Registro_Datos;
for Reg_Datos'Address use Dir_Reg_Datos;
     -- especifica la dirección del registro de datos

protected type Interfaz_Interrupcion(Id_Int : Interrupt_Id;
               Rc : access Registro_Control;
               Rd : access Registro_Datos) is
   entry Lee(Can : Canal; M : out Medicion);
private
```

```
   entry Hecho(Can : Canal; M : out Medicion);
   procedure Manejador;
   pragma Attach_Handler(Manejador, Id_Int);
   pragma Interrupt_Priority(Prioridad_Adc);
      -- véase la Sección 13.11 sobre prioridades
   Ocurrio_Interrupcion : Boolean := False;
   Siguiente_Peticion : Boolean := True;
end Interfaz_Interrupcion;•ADA 639
Interfaz_Adc : Interfaz_Interrupcion(Names.Adc,
                Reg_Control'Access,
                Reg_Datos'Access);
   -- Esto presupone que 'Adc' esta registrado como un
   -- Interrupt_Id en Ada.Interrupts.Names
   -- 'Access proporciona la dirección del objeto

protected body Interfaz_Interrupcion is
   entry Lee(Can : Canal; M : out Medicion)
         when Siguiente_Peticion is
      Registro_Sombra : Registro_Control;
   begin
      Registro_Sombra := (Inicia_Ad => Activo, Ienable => Activo,
            Hecho => Inactivo, Ca => Can, Error => Inactivo);
      Rc.all := Registro_Sombra;
      Ocurrio_Interrupcion := False;
      Siguiente_Peticion := False;
      requeue Hecho;
   end Lee;

   procedure Manejador is
   begin
      Ocurrio_Interrupcion := True;
   end Manejador;

   entry Hecho(Can : Canal; M : out Medicion)
                        when Ocurrio_Interrupcion is
   begin
      Siguiente_Peticion := True;
      if Rc.Hecho = Activo and Rc. Error = Inactivo then
            M := Medicion(Rd.all);
      else
         raise Error_Conversion;
```

```
      end if;
    end Hecho;
  end Interfaz_Interrupcion;

  procedure Lee(Ca : Canal; M : out Medicion) is
  begin
    for I in 1..3 loop
      begin
        Interfaz_Adc.Lee(Ca,M);
        return;
      exception
        when Error_Conversion => null;
      end;
    end loop;
    raise Error_Conversion;
  end Lee;
end Manejador_Dispositivo_Adc;
```

Las tareas cliente simplemente llaman al procedimiento `Lee` indicándole el número del canal del cual se desea leer, y una variable de salida para el valor concreto leído. Dentro del procedimiento, un bucle interno intenta tres conversiones llamando a la entrada `Lee` en el objeto protegido asociado al conversor. Dentro de esta entrada, se establecen los valores apropiados del registro de control `Rc`. Una vez que ha sido asignado el registro de control, la tarea cliente se reencola con una entrada privada para esperar la interrupción. El indicador `Siguiente_Peticion` se utiliza para asegurar que sólo sobresale una llamada a `Lee`.

Una vez que ha llegado una interrupción –como una llamada a un procedimiento protegido sin parámetros (parameterless)–, se establece la barrera sobre la entrada `Hecho`; esto ocasiona que se ejecute la entrada `Hecho` (como parte del manejador de la interrupción), lo que asegura que ha sido activado `Rc.Hecho` y que no se ha activado el indicador de error. Si éste es el caso, se construye el parámetro `M`, usando una conversión de tipo, desde el valor del registro búfer. (Observe que este valor no puede encontrarse fuera de rango por el subtipado de `Medicion`.) Si la conversión no hubiera tenido éxito, se generaría la excepción `Error_Conversion`; ésta es atrapada por el procedimiento Lee, que realiza en total tres intentos de conversión antes de permitir la propagación del error.

El ejemplo anterior muestra que suele ser necesario, cuando se escriben controladores de dispositivos, convertir de un tipo a otro. En estas circunstancias, las rígidas características de tipado de Ada pueden ser algo irritante. Sin embargo, es posible soslayar esta dificultad usando una función genérica proporcionada a tal efecto, como una unidad de biblioteca predefinida:

```
generic
  type Origen  (<>) is limited private;
  type Destino (<>) is limited private;
```

```
function Ada.Conversion_No_Comprobada(O : Origen) return Destino;
pragma Convention(Intrinsic, Ada.Conversion_No_Comprobada);
pragma Pure(Ada.Conversion_No_Comprobada);
```

El efecto de una conversión no comprobada es la copia de un patrón de bits origen hacia el destino. El programador debe asegurarse bien de que tal conversión tiene sentido, y de que cualquier patrón posible es aceptable como destino.

15.4.4 Acceso a dispositivos de E/S mediante instrucciones especiales

Si se precisan instrucciones especiales, habrá que integrar código de ensamblado junto al código Ada. Los mecanismos de inserción de código máquina permiten que los programadores escriban código Ada que contenga objetos invisibles para Ada. Esto se consigue de forma controlada permitiendo que el código máquina solamente pueda operar dentro del contexto del cuerpo de un subprograma. Más aún, si un subprograma contiene sentencias de código, entonces sólo podrá contener sentencias de código y cláusulas «use» (como es normal, se siguen permitiendo comentarios y pragmas).

Como sería de esperar, los detalles y las características del empleo de fragmentos de código dependen grandemente de la implementación; podrán usarse pragmas y atributos específicos de la implementación para imponer restricciones concretas y convenciones de llamada sobre la utilización de los objetos que definen código. Una sentencia de código tiene la siguiente estructura:

```
sentencia_codigo ::= expresion_cualificada
```

La expresión cualificada debería ser de un tipo declarado dentro del paquete de biblioteca predefinido denominado `System.Machine_Code`. Este paquete es el que proporciona declaraciones de registros (en Ada estándar) para representar las instrucciones de la máquina destino. El siguiente ejemplo muestra ésta aproximación:

```
D : Datos; -- de entrada

procedure Op_Entrada; pragma Inline(Op_Entrada);

procedure Op_Entr is
  use System.Machine_Code;
begin
  Formato_De_Mi_Maquina'(Code => Instruccion_Entrada, Reg => 1, Port => 1);
  Formato_De_Mi_Maquina'(CODE => SAVE, REG =>, S'Address);
end;
```

El pragma `Inline` instruye al compilador para que incluya código en línea, en lugar de una llamada a un procedimiento, donde quiera que se use el subprograma.

Incluso aunque este procedimiento de inserción de código esté definido en Ada, el lenguaje deja bastante claro (ARM 13.8.4) que una implementación no tiene por qué proporcionar un paquete `Machine_Code`, a menos que esté soportado el Anexo de Programación de Sistemas. Si no es así, no se permite el uso de fragmentos de código máquina.

15.5 Java para tiempo real

Aunque Java fue diseñado originalmente para sistemas embebidos y está lleno de posibilidades para la programación gráfica y el manejo de archivos, el lenguaje no dice nada sobre cómo programar controladores de dispositivos o cómo manejar interrupciones. Java para tiempo real pretende corregir esta situación, aunque actualmente sólo proporciona un soporte limitado.

15.5.1 Direccionamiento y manipulación de registros de dispositivos

Java para tiempo real soporta la posibilidad de acceder a los registros de los dispositivos mediante la noción de memoria de bajo nivel (raw memory). Se permite que cada implementación soporte un rango de tipos de memoria, como la memoria DMA o la memoria compartida. A este grupo bien podría pertenecer la memoria donde se reservan los registros de E/S: `IO_Page`. Para leer o escribir sobre esa memoria puede emplearse la clase `RawMemoryAccess` (definida en el Programa 15.3). Esta aproximación no permite hacer corresponder objetos definidos por el usuario sobre la memoria a bajo nivel, dado que esto podría permitir, en potencia, violaciones del mecanismo de tipado. En consecuencia, habrá que acceder a los registros individuales mediante tipos de datos básicos (byte, int, short, long) y usar las operaciones de manejo de bit de bajo nivel.

Programa 15.3. Un fragmento de la clase `RawMemoryClass`.

```
public class RawMemoryAccess
{
  protected RawMemoryAccess(long base, long talla);
  protected RawMemoryAccess(RawMemoryAccess memoria, long base,
                                                     long talla);

  public static RawMemoryAccess create(java.lang.Object tipo, long talla)
         throws SecurityException, OffsetOutOfBoundsException,
                SizeOutOfBoundsException,
                UnsupportedPhysicalMemoryException;

  public static RawMemoryAccess create(java.lang.Object tipo,
                                       long base, long talla)
```

(Continuación)

```
        throws SecurityException, OffsetOutOfBoundsException,
               SizeOutOfBoundsException,
               UnsupportedPhysicalMemoryException;

  public byte getByte(long desplazamiento)
        throws SizeOutOfBoundsException, OffsetOutOfBoundsException;

  public void getBytes(long desplazamiento, byte[] bytes, int bajo,
                       int numero) throws
               SizeOutOfBoundsException, OffsetOutOfBoundsException;

  // igualmente para enteros, enteros largos y enteros cortos

  public void setByte(long desplazamiento, byte valor) throws
               SizeOutOfBoundsException, OffsetOutOfBoundsException;

  public void setBytes(long desplazamiento, byte[] bytes, int bajo,
                       int numero) throws
               SizeOutOfBoundsException, OffsetOutOfBoundsException;

  // igualmente para enteros, enteros largos y enteros cortos
  ...
}
```

Considere de nuevo el registro de control y estatus del sencillo ADC mostrado en la Sección 15.4.3.

Bit	Nombre	Significado
0	Inicia A/D	Poner a 1 para iniciar una conversión.
6	Habilita/Inhibe Interrupciones	Poner a 1 para habilitar las interrupciones.
7	Hecho	A 1 indica que la conversión está completada.
8-13	Canal	El convertidor tiene 64 entradas analógicas, éste valor indica el canal que se precisa.
15	Error	A 1 el convertidor indica un fallo en el dispositivo.

Lo primero es crear una clase para el registro. El constructor para esta clase crea una instancia de la clase `RawMemoryAccess` que indica `IO_Page` como tipo de memoria. El método `ponPalabraControl` permitirá acceder al registro en cuestión.

```
public class RegistroControlYEstatus
{
  RawMemoryAccess memoriaDirecta;

  public RegistroControlYEstatus(long base, long talla)
  {
    memoriaDirecta = RawMemoryAccess.create(IO_Page, base, talla);
  }

  public void ponPalabraControl(short valor)
  {
    rawMemory.setShort(0, valor);
  }
}
```

Ahora, mediante un registro sombra para el dispositivo y los operadores lógicos de bit, puede construirse el patrón de bits adecuado. Por ejemplo, para iniciar una conversión sobre el canal 6:

```
byte sombra, canal;
final byte inicia = 01;
final byte habilita = 040;
final long direccionRcs = 015002;
final long tallaRce = 2;
RegistroControlYEstatus rcs = new
        RegistroControlYEstatus(direccionRce, tallaRce);

canal = 6;
sombra = (canal << 8) | inicia | habilita);
rcs.ponPalabraControl(sombra);
```

15.5.2 Manejo de interrupciones

Java para tiempo real contempla una interrupción como un evento asíncrono (véase la Sección 10.7). El acontecimiento de una interrupción es así equivalente a una llamada al método `fire`. La asociación entre la interrupción y el evento se obtiene invocando al método `bindTo` de la clase `AsyncEvent`. El parámetro es del tipo string, y se emplea de un modo dependiente de la implementación; una posible aproximación sería pasar la dirección del vector de interrupción. Cuando ocurre la interrupción, se invoca el método `fire` del manejador concreto. Ahora, es posible asociar el manejador con un objeto planificable y proporcionarle los parámetros adecuados de prioridad y liberación del bloqueo.

```
AsyncEvent Interrupcion = new AsyncEvent();
AsyncEventHandler ManejadorInterrupcion = new BoundAsyncEventHandler(
                    paramsPri, paramsLib, null, null, null);
Interrupt.addHandler(ManejadorInterrupcion);
Interrupt.bindTo("177760");
```

Programa 15.4. Un fragmento de la clase `POSIXSignalHandler`.

```
public final class POSIXSignalHandler
{
  public static final int SIGABRT;
  public static final int SIGALRM;
  public static final int SIGBUS
  ...

  public static synchronized void addHandler(int señal,
                AsyncEventHandler manejador);

  public static synchronized void removeHandler(int señal,
                AsyncEventHandler manejador);

  public static synchronized void setHandler(int señal,
                AsyncEventHandler manejador);

}
```

En el caso de que el programa Java para tiempo real se ejecute sobre un sistema operativo compatible POSIX, puede usarse la clase `POSIXSignalHandler` (véase el Programa 15.4) para asociar manejadores de eventos asíncronos con la aparición de una señal POSIX. Curiosamente, no están soportadas las señales de POSIX para tiempo real.

15.6 Occam2

En las primeras secciones de este capítulo se ha mostrado como se puede hacer corresponder un modelo de comunicación y sincronización de variables compartidas sobre máquinas con E/S sobre memoria. Sin embargo, los modelos no tratan elegantemente con máquinas con instrucciones especiales, y se recurre a procedimientos especiales, código de ensamblado en línea, o variables de tipos especiales reconocibles para el compilador. En esta sección, se examina el lenguaje occam2 como ejemplo de lenguaje de programación concurrente basado en mensajes que emplea mensajes para controlar los dispositivos.

Aunque occam2 se diseñó para el transputer, en la siguiente discusión se considera como un lenguaje independiente de la máquina. Primeramente se presenta el modelo, y después se trae a

colación su implementación sobre máquinas de E/S sobre memoria y máquinas con instrucciones especiales. Como con el control de dispositivos con variables compartidas, es preciso tener en cuenta tres cuestiones: encapsulamiento de dispositivos, manipulación de registros y manejo de interrupciones.

Mecanismo de modularidad y encapsulamiento

El único mecanismo de encapsulamiento que proporciona occam2 es el procedimiento, y habrá que usarlo para encapsular los controladores de dispositivos.

Direccionamiento y manipulación de los registros del dispositivo

Los registros de dispositivos se corresponden con `PORTS` (puertos), que son conceptualmente similares a los canales de occam2. Por ejemplo, si cierto registro de 16 bits está en la posición `X`, habrá que definir un `PORT P` como el siguiente:

```
PORT OF INT16 P:
PLACE P AT X:
```

Observe que esta dirección puede interpretarse como una posición de memoria o como una dirección de dispositivo, según la implementación.

Para interactuar con el registro del dispositivo habrá que leer o escribir sobre este puerto:

```
P ! A -- escribe el valor de A sobre el puerto

P ? B -- lee el valor del puerto sobre B
```

No puede definirse un puerto sólo para leer o para escribir.

La importante diferencia entre puertos y canales en occam2 es que no hay sincronización alguna asociada a la interacción con el puerto. Ni la lectura ni la escritura pueden ocasionar la suspensión de la ejecución de un proceso; siempre se escribe el valor sobre la posición indicada, e, igualmente, el valor es leído siempre. De esta manera, un puerto es un canal donde el compañero siempre está listo para comunicar.

Occam2 proporciona mecanismos para manipular registros de dispositivos usando operaciones de desplazamiento y expresiones lógicas de bit. Sin embargo, no tiene un equivalente al tipo bit de Modula-1 o a las especificaciones de representación de Ada.

Manejo de interrupciones

En occam2, cada interrupción se maneja como una cita con el proceso hardware. Asociada a la interrupción se encuentra una dirección, independiente de la implementación, que, en el caso sencillo de entrada/salida descrito en este capítulo, es la dirección del vector de interrupción; así, se hace corresponder un canal sobre esta dirección (`ADDR`):

```
CHAN OF ANY Interrupcion:
PLACE Interrupcion AT ADDR:
```

Tenga en cuenta que se trata de un canal, y no de un puerto. Esto es así porque la interrupción lleva asociada una sincronización, mientras que no hay ninguna asociada al acceso al dispositivo. El protocolo de datos para este canal también será dependiente de la implementación.

El manejador de la interrupción podrá esperar datos de entrada desde el canal designado:

```
INT ANY: -- define ANY para que sea del tipo del protocolo
SEQ
  -- el uso de los puertos habilita la interrupcion
  Interrupcion ? ANY
  -- acciones precisas cuando aparece una interrupción
```

La plataforma de ejecución deberá sincronizarse con el canal indicado cuando aparezca una interrupción externa. Para garantizar tiempos de respuesta, el proceso manejador de la interrupción suele disponer de alta prioridad. Así, no sólo se ejecutará por la interrupción, sino que será ejecutado realmente en un corto periodo de tiempo (suponiendo que ningún otro proceso de mayor prioridad se esté ejecutando).

Para dar cuenta de las interrupciones, que se perderán si no se manejan dentro de un periodo específico, es preciso verlo como si el hardware enviara un tiempo límite de espera (timeout) sobre la comunicación. El hardware deberá enviar conceptualmente:

```
ALT
  Interrupcion ? ANY
    SKIP
  CLOCK ? AFTER Tiempo PLUS Timeout
    SKIP
```

y el manejador debe ejecutar:

```
Interrupcion ! ANY
```

Esto es así porque sólo se puede asociar un tiempo límite de espera a una entrada.

Implementación en máquinas con E/S sobre memoria y con instrucciones especiales

La correlación entre el modelo de occam2 de control de dispositivos y el de las máquinas con E/S sobre memoria sólo requiere hacer corresponder las peticiones de entrada y salida sobre puertos con operaciones de lectura y escritura sobre los registros de los dispositivos. La aplicación del modelo a las máquinas con instrucciones especiales precisa lo siguiente:

- Asociar un PORT de occam2 con cada puerto de E/S mediante la sentencia PLACE.
- Situar los datos enviados sobre un PORT de occam2 en un acumulador adecuado para su uso con la instrucción de la máquina para la salida.

- Hacer disponibles los datos recibidos desde un PORT de occam2, mediante un acumulador dispuesto al efecto, tras la ejecución de la instrucción para la entrada.

15.6.1 Un ejemplo de controlador de dispositivo

Para ilustrar el empleo de los mecanismos de entrada/salida de bajo nivel que proporciona occam2, se desarrollará un proceso que controla un convertidor analógico a digital (ADC; analogue to digital converter) sobre una máquina con E/S sobre memoria. El convertidor es el mismo que se describió en la Sección 15.4.3 y se implementó en Ada y Java. Para leer un valor de entrada analógico, se proporciona una dirección de canal (que no debe confundirse con un canal de occam2) sobre los bits 8 a 13, y después se establecerá el bit 0 para iniciar la conversión. Una vez cargado el valor sobre los registros de resultado, el dispositivo interrumpirá al procesador. Después se comprobará el indicador de error antes de leer los resultados del registro. Durante esta interacción, sería deseable poder inhabilitar la interrupción.

El controlador del dispositivo recibirá peticiones y proporcionará respuestas una y otra vez, y se programará como un PROC con una interfaz de dos canales. Cuando se le pasa una dirección (de uno de los ocho canales de entrada analógica) hacia `entrada`, se devuelve un resultado de 16 bits por el canal `salida`.

```
CHAN OF INT16 peticion:
CHAN OF INT16 resultado:
PROC ADC(CHAN OF INT16 entrada, salida)
  -- cuerpo de PROC, véase más adelante
PRI PAR
  ADC(peticion, resultado)
  PAR
    -- resto del programa
```

Se prefiere PRI PAR, dado que el ADC debe manejar una interrupción cada vez que es empleado, y por ello deberá correr con la prioridad más alta.

Dentro del cuerpo del PROC, deberán declararse el canal de interrupción y los dos PORT:

```
PORT OF INT16 Registro.Control:
PLACE Registro.Register AT #AA12#:
PORT OF INT16 Registro.Bufer:
PLACE Registro.Bufer AT #AA14#:
CHAN OF ANY Interrupcion:
PLACE Interrupcion AT #40#:
INT16 R.Control: -- variable que representa el registro de control
```

donde #AA12# y #AA14# son las direcciones hexadecimales predefinidas de los dos registros, y #40# es la dirección en el vector de interrupción.

Para dar instrucciones al hardware de que efectúe una operación, se precisa establecer los registros 0 y 6 delregistro de control; a la vez, el resto de los registros que no sean del 8 al 13 (inclusive) deberán ponerse a cero. Esto se obtiene empleando las siguientes constantes;

```
VAL INT16 cero IS 0:
VAL INT16 Ya IS 65:
```

Una vez recibida una dirección desde el canal «entrada», debe asignarse este valor a los bits entre el 8 y el 10 en el registro de control. Esto se hace mediante la operación de desplazamiento. Las acciones que hay que llevar a cabo para arrancar una conversión son:

```
INT16 Direccion:
SEQ
  entrada ? Direccion
  IF
    (Direccion < 0) OR (Direccion > 63)
      salida ! MOSTNEG INT16 -- condición de error
    TRUE
      SEQ
        R.Control := cero
        R.Control := Direccion << 8
        R.Control := R.Control BITOR Ya
        Registro.Control ! Control.R
```

Cuando llegue la interrupción, se lee el registro de control y se comprueba el indicador de error y «Hecho». Para ello, habrá que aplicar una máscara sobre el registro de control contra las constantes adecuadas:

```
VAL INT16 Hecho IS 128:
VAL INT16 Error IS MOSTNEG INT16:
```

MOSTNEG se representa como 1 000 000 000 000 000.

Las comprobaciones son como sigue:

```
SEQ
  Registro.Control ? R.Control
  IF
    ((Done BITAND R.Control) = 0) OR
    ((Error BITAND R.Control) <> cero)
      -- error
    TRUE
      -- el valor concreto se encuentra en el registro búfer
```

Aunque el controlador del dispositivo deba correr con una prioridad alta, esto no suele ser así en el caso del proceso cliente, por lo que el controlador se verá demorado si intenta invocar directamente a éste y no está preparado. Con los dispositivos que generan datos asíncronamente, este retraso podría provocar que el controlador perdiera una interrupción. Para evitar esto, hay que manejar los datos de entrada con su correspondiente búfer. A continuación se muestra un búfer circular apropiado. Dese cuenta de que, dado que el cliente desea leer del búfer y que el ALT en el búfer no puede tener guardas de salida, se precisa otro sencillo elemento búfer. Para asegurar que el controlador del dispositivo no se vea retrasado por causa del algoritmo de planificación, ambos procesos búfer (así como el controlador) deberán ejecutarse con prioridad elevada.

```
PROC bufer(CHAN OF INT pon, dame)
  CHAN OF INT Peticion, Respuesta:
  PAR
    VAL INT Talla.Buf IS 32:
    INT tope, base, contenidos:
    [Talla.Bufer]bufer:
    SEQ
      contenidos := 0
      tope := 0
      base := 0
      INT ANY:
      WHILE TRUE
        ALT
          contenidos < Talla.Buf & pon ? bufer [tope]
            SEQ
              contenidos := contenidos + 1
              tope := (tope + 1) REM Talla.Buf
          contenidos > 0 & Peticion ? ANY
            SEQ
              Respuesta ! bufer[base]
              contenidos := contenidos - 1
              base := (base + 1) REM Talla.Buf
    INT Temp: -- proceso búfer simple
    VAL INT ANY IS 0: -- valor de relleno
    WHILE TRUE
      SEQ
        Peticion ! ANY
        Respuesta ? Temp
        dame ! Temp
:
```

Ahora, podemos tener el código completo para el PROC. El controlador del dispositivo se encuentra, de nuevo, estructurado para que se efectúen tres intentos para obtener una lectura correcta.

```
PROC ADC(CHAN OF INT16 entrada, salida)
  PORT OF INT16 Registro.Control:
  PLACE Registro.Control AT #AA12#:
  PORT OF INT16 Registro.Bufer:
  PLACE Registro.Bufer AT #AA14#:

  CHAN OF ANY Interrupcion:
  PLACE Interrupcion AT #40#:
  TIMER CLOCK:

  INT16 R.Control: -- variable que representa el búfer de control
  INT16 R.Bufer:   -- variable que representa el búfer de resultados
  INT Tiempo:

  VAL INT16 cero IS 0:
  VAL INT16 Ya IS 65:
  VAL INT16 Hecho IS 128:
  VAL INT16 Error IS MOSTNEG INT16:
  VAL INT Timeout IS 600000: -- o cualquier otro valor apropiado
  INT ANY:
  INT16 Direccion:
  BOOL Encontrado, Error:
  CHAN OF INT16 Entrada.Bufer:

  PAR
    bufer(Entrada.Bufer, salida)
    INT16 Intento:
    WHILE TRUE
      SEQ
        entrada ? Direccion
        IF
          (Direccion < 0) OR (Direccion > 63)
            Entrada.Bufer ! MOSTNEG INT16
                    -- condición de error
          TRUE
            SEQ
              Intento := 0
              Error := FALSE
              Encontrado := FALSE
              WHILE (Intento < 3) AND ((NOT Encontrado) AND (NOT Error))
                -- Se hacen tres intentos para obtener una lectura del
```

```
-- ADC. Esta lectura puede ser correcta o marcada
-- como un error
SEQ
   R.Control := cero
   R.Control := Direccion << 8
   R.Control := R.Control BITOR Ya
   Registro.Control ! R.Control
   CLOCK ? Tiempo
   ALT
      Interrupcion ? ANY
         SEQ
            Registro.Control ? R.Control
            IF
               ((Hecho BITAND R.Control) = 0) OR
                    ((Error BITAND R.Control) <> cero)
                  SEQ
                     Error := TRUE
                     Entrada.Bufer ! MOSTNEG INT16
                     -- condicion de error
               TRUE
                  SEQ
                     Encontrado := TRUE
                     Registro.Bufer ? R.Bufer
                     Entrada.Bufer ! R.Bufer
         CLOCK ? AFTER Tiempo PLUS Timeout
            -- El dispositivo no responde
            Intento := Intento + 1
IF
   (NOT Encontrado) AND (NOT Error)
      Entrada.Bufer ! MOSTNEG INT16
   TRUE
      SKIP
```

15.6.2 Dificultades con el control de dispositivos en occam2

El ejemplo anterior muestra algunas de las dificultades en la confección de controladores de dispositivos y manejadores de señales en occam2. En concreto, no existe una relación directa entre la prioridad hardware del dispositivo y la prioridad asignada al proceso controlador. Para garantizar que se les da la prioridad adecuada a los dispositivos más urgentes, es preciso ordenar

apropiadamente todos los controladores de dispositivos en el nivel más externo del programa en una construcción `PRI PAR`.

La otra dificultad principal proviene de la carencia de estructuras de datos para representar registros de dispositivos. Esto hace que el programador deba emplear técnicas de manipulación de bit de bajo nivel, propensas a provocar errores.

15.7 C y otros lenguajes de tiempo real primitivos

La primera generación de lenguajes de programación de tiempo real (RTL/2, Coral 66 y otros) no proporcionaban soporte para la programación concurrente o para la programación de dispositivos. Las interrupciones solían contemplarse como llamadas a procedimientos, y muy frecuentemente la única posibilidad disponible para acceder a los registros de los dispositivos era permitir la inclusión de código en lenguaje de ensamblado. Por ejemplo, RTL/2 (Barnes, 1976) disponía de una sentencia de código como ésta:

```
code talla_codigo, talla_pila
  mov R3,@variable
  ...
  ...
@rtl
```

Una desventaja de esta aproximación es que para acceder a las variables RTL/2 se necesita conocer la estructura del código generado por el compilador.

Otra característica común en los lenguajes de tiempo real primitivos era que tendían a ser débilmente tipados, de modo que las variables podían ser tratadas como secuencias de bits de longitud fija. Esto permitía manipular individualmente los bits y los campos de los registros empleando operadores de bajo nivel, tales como el desplazamiento lógico y las instrucciones de rotación. Sin embargo, las desventajas del tipado débil sobrepasaban, con mucho, los beneficios que ofrecía su flexibilidad.

El lenguaje C, aunque más reciente que RTL/2 y Coral 66, ha continuado con esta tradición. Los registros de los dispositivos se direccionan mediante variables puntero a las que se puede asignar las posiciones en memoria de los registros. Éstos se manipulan mediante operadores lógicos de bit de bajo nivel o mediante los campos de bits de las definiciones de estructuras. Esto último recuerda a las cláusulas de representación de registros de Ada, aunque, en realidad, depende tanto de la máquina como del compilador. Para mostrar el empleo de los mecanismos de manipulación de C, se presentan dos ejemplos; en el primero se usan operadores lógicos de bit de bajo nivel, y en el segundo campos de bits.

Considere de nuevo el registro de control y estatus para el sencillo ADC de la Sección 15.4.3.

Usando los operadores lógicos de bit, debemos definir primero un conjunto de máscaras que se corresponden con cada posición de bit.

```
#define INICIA    01 /* los números que comienzan por 0 están en octal */
#define HABILITA 040
#define ERROR    0100000
```

El campo «Canal» puede definirse bien en base a bits, o desplazando el valor numérico hasta su posición correcta. Este modo es el que se usará a continuación, donde se precisa del canal número 6:

```
unsigned short int *registro, sombra, canal;

registro = 0xAA12;
canal    = 6;
sombra   = 0;

sombra |= (canal << 8) | INICIA | HABILITA ;
*registro = sombra;
```

Con campos de bits, esto se convierte en:

```
struct {
  unsigned int inicia       : 1; /* campo de un bit de longitud */
  unsigned int              : 5; /* campo sin nombre de 5 bits */
  unsigned int interrupcion : 1; /* campo de un bit */
  unsigned int hecho        : 1; /* campo de un bit */
  unsigned int canal        : 6; /* campo de 6 bits */
  unsigned int error        : 1; /* campo de un bit */
} registro_control;

registro_control *registro, sombra;
registro = 0xAA12;
sombra.inicia = 1;
shadow.interrupcion = 1;
shadow.canal = 6;
shadow.error = 0;

*registro = sombra;
```

En relación con este ejemplo, hay que resaltar dos puntos:

- C no proporciona garantía alguna sobre el ordenamiento de los campos, de modo que el compilador podrá decidir empaquetar los campos en la palabra de diferente forma a como supone el programador.
- C no aborda el problema de si la máquina numera los bits de izquierda a derecha o de derecha a izquierda.

Visto lo visto, queda claro que no se deben usar campos de bits para acceder a los registros de los dispositivos, a menos que el programador conozca cómo son implementados por un compilador concreto para la máquina concreta que está usando. Incluso en este caso, el código no será portable.

Por razones de portabilidad, el programador de C se verá forzado a usar los operadores lógicos de bit de bajo nivel. El siguiente ejemplo muestra cómo éstos pueden hacerse fácilmente ilegibles (aunque posiblemente se produzca un código más eficiente). El siguiente procedimiento establece `n` bits comenzando con la posición `p` en el registro apuntado por `reg` hasta `x`.

```
unsigned int ponbits(unsigned int *reg, unsigned int n,
        unsigned int p, unsigned int x)
{
  unsigned int datos, mascara;

  datos = (x & (~(~0 << n))) << (p);  /* datos a aplicar la máscara */
  mascara = ~(~0 << n);               /* máscara */
  *reg &= ~(mascara << (p));          /* limpia los bits actuales */
  *reg |= data;                       /* OR sobre los datos */
}
```

Este código C es muy conciso: «~» realiza un complemento bit a bit, «<<» es un desplazamiento a la izquierda (con un relleno de 0), «&» es un «and» de bits, y «|» es un «or» de bits.

Con la arquitectura de E/S simple indicada en la Sección 15.1.4, los manejadores de interrupción se asignan ubicando la dirección de un procedimiento sin parámetros en el lugar apropiado del vector de interrupción. Una vez que se ejecuta el procedimiento, cualquier comunicación y sincronización con el resto del programa deberá programarse directamente.

Aunque POSIX da mecanismos alternativos que, en teoría, podrían utilizarse para proporcionar un modelo alternativo para el manejo de interrupciones (por ejemplo, asociando una interrupción con una variable de condición), no hay actualmente ningún mecanismo estándar para vincular manejadores definidos por el usuario a las interrupciones.

15.8 Planificación de controladores de dispositivos

Dado que muchos sistemas de tiempo real incluyen componentes de E/S, es importante que el análisis de planificabilidad tenga en cuenta cualquier característica concreta de este tipo de programación de bajo nivel. Ya se indicó que las técnicas de DMA y de programa de control de canal suelen ser demasiado impredecibles (en su comportamiento temporal) como para poder ser analizadas. Por eso, fijaremos nuestra atención en las aproximaciones de programa dirigido por eventos y por estatus.

Donde una interrupción libera la ejecución de ciertos procesos esporádicos, hay que considerar también el coste asociado con el manejador de la interrupción. La prioridad de este manejador suele ser mayor que la del proceso esporádico, lo que implica que aquellos procesos cuya prioridad sea mayor que la de los procesos esporádicos (pero menor que la del manejador de la interrupción) sufrirán ciertas interferencias. Realmente, éste es un caso de inversión de la prioridad, puesto que el único cometido del manejador es activar cierto proceso esporádico, y su prioridad debería ser, en el caso ideal, la misma que la del proceso esporádico. Sin embargo, la mayoría de las plataformas hardware precisan que las prioridades de interrupción sean mayores que las prioridades software ordinarias. Para modelar el manejador de interrupción, se incluye un «proceso» extra en la prueba de planificabilidad. Su «periodo» es igual al del proceso esporádico, su prioridad igual a la del nivel de prioridad de la interrupción, y su tiempo de ejecución igual al comportamiento en su peor caso.

Para los dispositivos dirigidos por estatus, puede analizarse el código de control de la manera habitual. Tales dispositivos, sin embargo, presentan una dificultad particular. El protocolo para emplear un dispositivo de entrada suele ser como sigue: petición de lectura, espera hasta que el hardware efectúa la lectura, y acceso al registro para realizar la lectura desde el programa. El problema está en cómo tratar con el retraso mientras se toma la lectura. En función de la duración del retraso, tenemos tres aproximaciones posibles:

- Espera ocupada sobre el indicador «hecho».
- Replanificación del proceso para un instante posterior.
- Para procesos periódicos, separación de la acción entre periodos.

La espera ocupada es aceptable cuando el retraso es pequeño. Desde el punto de vista de la planificación, el «retardo» se contempla como tiempo de cómputo, y siempre que esté acotado, la aproximación del análisis no sufre modificación alguna. Para protegerse contra cualquier fallo en el dispositivo (como cuando nunca se establezca el bit «hecho»), se puede usar un algoritmo con tiempo de espera acotado (véase la Sección 12.4).

Si el retraso es notable, lo más eficaz es suspender el proceso, lanzar otro trabajo, y volver al proceso con E/S en otro momento, cuando esté disponible el resultado. Así, si el tiempo de lectura fuera de 30 ms, el código podría ser:

```
begin
  -- prepara la lectura
  delay Milliseconds(30);
  -- toma y emplea la lectura
end;
```

Desde el punto de vista de la planificación, este esquema tiene tres implicaciones importantes. En primer lugar, no es fácil calcular los tiempos de respuesta. Analizando separadamente cada mitad del proceso, el tiempo de respuesta total se obtiene sumando los dos subtiempos de respuesta y el retardo de 30 ms. Aunque hay un retraso en el proceso, se ignorará cuando se considere el impacto sobre otros procesos de menor prioridad. En segundo lugar, el tiempo de cálculo extra

que supone el retraso y la replanificación debe añadirse al tiempo de ejecución del peor caso posible del proceso (véase la Sección 16.3, donde se discute cómo incluir estas sobrecargas del sistema). En tercer lugar, hay un impacto sobre el bloqueo. Recuerde cómo era la sencilla ecuación para calcular el tiempo de respuesta de un proceso (véase la Sección 13.7):

$$R_i = C_i + B_i + I_i$$

B_i es el tiempo de bloqueo (esto es, el tiempo máximo que puede retrasarse el proceso debido a las acciones de los procesos de menor prioridad). En la Sección 13.10 se consideraron diversos protocolos para la compartición de recursos, los cuales tenían todos la propiedad de que B_i contenía sólo un bloqueo. Sin embargo, cuando un proceso sufre un retardo (y permite la ejecución de procesos de menor prioridad), puede bloquearse de nuevo cuando sea liberado de la cola de retardo. Debido a ello, la ecuación de tiempo de respuesta se convierte en:

$$R_i = C_i + (N + 1)B_i + I_i$$

donde N es el número de retardos internos.

En el caso de los procesos periódicos, hay otra forma de tratar con este retardo explicito. Este método se denomina **desplazamiento del periodo**, e implica iniciar la lectura en un periodo y tomar el resultado en el siguiente. Por ejemplo:

```
-- prepara la primera lectura
loop
  delay until Proxima_Ejecucion;
  -- comprueba el indicador 'hecho'
  -- toma la lectura y lo utiliza
  -- prepara la siguiente lectura
  Proxima_Ejecucion := Proxima_Ejecucion + Periodo;
end loop;
```

Ésta es una aproximación sencilla sin impacto alguno sobre la planificación. Obviamente, la lectura tiene una antigüedad de un periodo, que pudiera no ser aceptable para la aplicación. Para garantizar que hay un intervalo suficiente entre el final de una ejecución y el comienzo de la siguiente, habrá que ajustar el tiempo límite del proceso. Así, si S es el tiempo que precisa el dispositivo, la constante adecuada es $D \leq T - S$. Observe que el tiempo de detención máxima de lectura está acotado por $T + D$ (o $T + R$ una vez calculado el tiempo de respuesta del peor caso posible).

15.9 Gestión de memoria

Los sistemas embebidos de tiempo real suelen tener limitada la cantidad de memoria disponible; esto ocurre por cuestiones económicas o por otras restricciones relativas al conjunto del sistema (por ejemplo, restricciones de tamaño, potencia o peso). Por esta razón, es necesario controlar có-

mo se asigna esta memoria para garantizar una utilización eficiente. Más aún, cuando haya más de un tipo de memoria (con diferentes propiedades de acceso) dentro del sistema, será necesario indicar al compilador que ubique ciertos tipos de datos en ciertas posiciones. Así, el programa será capaz de incrementar sus prestaciones y su predecibilidad, así como de interaccionar con el entorno circundante.

En este capítulo ya se ha tenido en cuenta cómo asignar elementos de datos a posiciones de memoria concretas, y cómo utilizar ciertos campos en las unidades de almacenamiento para representar tipos de datos concretos. Esta sección considera las cuestiones más generales sobre la gestión del almacenamiento. La atención se enfoca en la gestión de dos componentes básicos que emplean los compiladores para gestionar los datos en tiempo de ejecución: el montón (heap) y la pila de retorno (stack).

15.9.1 Gestión del montón

La implementación de tiempo de ejecución de la mayoría de los lenguajes de programación proporciona un gran bloque de memoria (denominado **montón**) para que el programador pueda reservar fragmentos de memoria sobre la marcha (por ejemplo, para alojar un array cuyos límites no se conocían en el momento de la compilación). Para este fin suele emplearse un asignador (generalmente el operador **new**). Éste devuelve un puntero a la memoria del montón, donde se dispone de suficiente cantidad para la estructura de datos del programa. El sistema de soporte de ejecución es el responsable de la gestión del montón. Los problemas clave son decidir cuánto espacio se precisa y cuándo puede liberarse el espacio reservado. El primero de estos problemas, en general, requiere cierto conocimiento previo. El segundo puede tratarse de formas diferentes, como:

- Exigiendo que el programador devuelva explícitamente la memoria (es propenso a errores, pero es fácil de implementar); es la aproximación tomada por el lenguaje de programación C, que dispone de las funciones `malloc` y `free` (la función `sizeof` permite conocer el tamaño de los tipos de datos en bytes, a través del compilador);

- Exigiendo que el sistema de soporte de ejecución monitorice la memoria y determine mediante lógica cuándo no se va a acceder a ella (las reglas de alcance de Ada permiten que una implementación adopte esta aproximación): cuando cierto tipo de acceso deja de estar al alcance, toda la memoria asociada con ese tipo de acceso puede ser liberada;

- Requiriendo que el sistema de soporte de ejecución monitorice la memoria y libere aquellos fragmentos que no van a ser usados (**recolección de basura**). Éste es, quizás, el mecanismo más general, dado que permite liberar la memoria aunque el tipo de acceso asociado esté todavía al alcance.[2]

Desde la perspectiva de tiempo real, las aproximaciones anteriores tienen un impacto creciente sobre la posibilidad de analizar las propiedades de temporización del programa. En concre-

[2] N. de los T.: Aun cuando el término "compactación automática de memoria" parezca más apropiado, se conservará el término "recolección de basura" (garbagge collection), dada su amplia difusión.

to, la recolección de basura podría efectuarse cuando el montón estuviera vacío, o mediante una actividad asíncrona (recolección de basura incremental). En cualquier caso, la ejecución del recolector de basura puede tener un impacto importante sobre el tiempo de respuesta de cierta tarea crítica en el tiempo. Aunque no hay mucho trabajo sobre recolección de basura en tiempo real y siguen realizándose avances –véase Lim et al. (1999), Siebert (1999), y Kim et al. (1999)–, existe aún cierta resistencia a confiar en estas técnicas para los sistemas críticos en el tiempo.

Las siguientes subsecciones explorarán con mayor profundidad los mecanismos que proporciona Ada y Java para tiempo real.

Gestión del montón en Ada

En Ada, el montón se representa mediante uno o más **depósitos de almacenamiento** (storage pools). Los depósitos de almacenamiento se asocian con una partición concreta de Ada (en el caso de un sistema Ada no distribuible, ésta no es más que el programa completo). A cada objeto de un tipo de acceso (access) se le asocia un depósito de almacenamiento. El asignador («new») toma su memoria del depósito indicado. El mecanismo `Ada.Unchecked_Deallocation` (desasignación no comprobada) devuelve el dato al depósito. Cada implementación puede soportar un solo depósito global (que puede desvincularse cuando termina la partición), o podría soportar depósitos definidos en diferentes niveles de acceso (que podrán reclamarse cuando se abandone el alcance asociado). Por defecto, la implementación elige un depósito de almacenamiento estándar por tipo de acceso. Observe que todos los objetos a los que se accede directamente (no mediante punteros) se ubican en la pila, no en el montón.

Para proporcionar más control al usuario sobre la gestión del almacenamiento, Ada define un paquete denominado `System.Storage_Elements`, el cual se presenta en el Programa 15.5

Cada programador puede implementar sus propios depósitos de almacenamiento extendiendo el tipo `Root_Storage_Pool` y proporcionando las implementaciones concretas para los cuerpos del subprograma. Para asociar un tipo de acceso a un depósito de almacenamiento, primeramente se declara el depósito, y después se emplea el atributo `Storage_Pool`:

```
Mi_Deposito : Cierto_Tipo_De_Deposito_De_Almacenamiento;

type A is access Cierto_Objeto;
for A'Cierto_Deposito use Mi_Deposito;
```

A partir de ahora todas las llamadas a «new» que usan `A` llamarán automáticamente a `Allocate`; las llamadas a `Ada.Unchecked_Deallocation` llamarán a `Deallocate`; ambas se refieren a `Mi_Deposito`. Además, la implementación invocará a `Deallocate` cuando el tipo de acceso ya no sea alcanzable.

Finalmente, hay que decir que Ada no precisa que una implementación soporte la recolección de basura. Sin embargo, soporta un pragma `Controlled`, que indica que *no* se efectúe recolección de basura sobre cierto tipo concreto.

Programa 15.5. El paquete Ada `System.Storage_Pools`.

```
with Ada.Finalization;
with System.Storage_Elements;

package System.Storage_Pools is

   pragma Preelaborate(System.Storage_Pools);

   type Root_Storage_Pool is abstract new
        Ada.Finalization.Limited_Controlled with private;

   procedure Allocate(Deposito : in out Root_Storage_Pool;
             Direccion_Almacenamiento : out Address;
             Talla_Elementos_Almacenados :
                  in System.Storage_Elements.Storage_Count;
             Alineamiento : in System.Storage_Elements.Storage_Count)
             is abstract;

   procedure Deallocate(Deposito : in out Root_Storage_Pool;
                  Storage_Address : in Address;
             Talla_Elementos_Almacenados : in System.
             Storage_Elements.Storage_Count;
             Alineamiento : in System.Storage_Elements.Storage_Count)
             is abstract;

   function Storage_Size(Deposito : Root_Storage_Pool) return
            System.Storage_Elements.Storage_Count is abstract;
private
...
end System.Storage_Pools;
```

Gestión del montón en Java para tiempo real

A diferencia de lo que ocurre en Ada, todos los objetos de Java se reservan del montón, y el lenguaje precisa de un recolector de basura para su correcta implementación. Java para tiempo real reconoce que es necesario permitir una gestión de memoria que no se vea afectada por las vaguedades del comportamiento del recolector de basura. Con este fin, presenta la noción de **áreas de memoria**, algunas de las cuales existen al margen del montón tradicional en Java, y nunca padecen la recolección de basura. El Programa 15.6 define la clase abstracta para cualquier área de memoria. Cuando se entra en cierta área de memoria, todas las asignaciones de objetos se efectúan sobre esa área.

Mediante esta clase abstracta, Java para tiempo real define diversos tipos de memoria, entre ellos los siguientes.

Programa 15.6. La clase abstracta Java `MemoryArea`.

```
public abstract class MemoryArea
{
  protected MemoryArea(long tallaEnBytes);

  public void enter(java.lang.Runnable logica);
  // asocia esta area de memoria al hilo actual
  // durante la duración del método logica.run

  public static MemoryArea getMemoryArea(java.lang.Object objeto);
  // obtiene el area de memoria asociada al objeto

  public long memoryConsumed();
  // cantidad de bytes consumidos en esta area de memoria
  public long memoryRemaining();
  // cantidad de bytes restantes

  public synchronized java.lang.Object newArray(java.lang.class tipo,
         int numero) throws IllegalAccessException,
         InstantiationException, OutOfMemoryError;
  // asigna un array

  public synchronized java.lang.Object newInstance(java.lang.class tipo)
         throws IllegalAccessException, InstantiationException,
         OutOfMemoryError;
  // asigna un objeto

  public long size(); // la talla del area de memoria
}
```

- *Memoria inmortal*

  La memoria inmortal es compartida por todos los hilos de una aplicación. Los objetos creados en memoria inmortal no padecen nunca recolección de basura, y sólo se liberan cuando termina el programa.

```
public final class ImmortalMemory extends MemoryArea
{
  public static ImmortalMemory instance();
}
```

  Existe también una clase denominada `InmortalPhysicalMemory`, que dispone de las mismas características de la memoria inmortal pero permite reservar objetos sobre un rango de direcciones físicas.

- *Memoria con alcance*

  La memoria con alcance (scoped memory) es un área de memoria donde pueden reservar aquellos objetos con tiempo de vida bien definido. Se puede *entrar* en la memoria con alcance explícitamente (empleando el método `enter`), o de modo implícito (asociándolo a un `RealtimeThread` en el momento de creación del hilo). A cada memoria con alcance se le asocia un contador de referencias. Éste se incrementa en cada llamada a `enter` y con cada creación de un hilo asociado. Se decrementa cuando vuelve el método `enter` y con la finalización de cada hilo asociado. Cuando el contador de referencias llega a 0, se ejecuta el método de finalización de cada objeto residente en la memoria con alcance, y se reclama la memoria. La memoria con alcance se puede anidar mediante llamadas anidadas al método `enter`.

  La clase `ScopedMemory` (definida en el Programa 15.7) es una clase abstracta que dispone de varias subclases, entre las que se incluyen:

  - `VTMemory`: las reservas pueden tomarse un tiempo variable.
  - `LTMemory`: las reservas ocurren en tiempo lineal (con respecto al tamaño del objeto).
  - `ScopedPhysicalMemory`: permite la reserva de objetos sobre posiciones físicas.

  La definición de estas subclases se proporciona en el Apéndice A.

Programa 15.7. La clase Java para tiempo real `ScopedMemory`.

```
public abstract class ScopedMemory extends MemoryArea
{
  public ScopedMemory(long talla);

  public void enter(java.lang.Runnable logica);

  public int getMaximumSize();

  public MemoryArea getOuterScope();

  public java.lang.Object getPortal();

  public void setPortal(java.lang.Object objeto);
}
```

Para evitar la posibilidad de punteros colgantes, se establece un conjunto de restricciones sobre la utilización de las diversas áreas de memoria:

- Los objetos del montón sólo pueden referenciar otros objetos del montón y objetos en memoria inmortal (es decir, no pueden acceder a la memoria con alcance).

- Los objetos inmortales sólo pueden referenciar objetos del montón y objetos en memoria inmortal.
- Los objetos con alcance sólo pueden referenciar objetos del montón, objetos inmortales, y objetos del mismo alcance o de un alcance exterior.

Cuando se crean hilos de tiempo real y manejadores de eventos asíncronos, es posible indicar los parámetros de memoria. Éstos pueden ser empleados por el planificador como parte de la política de control de admisión y/o con el propósito de garantizar una adecuada recolección de basura. El Programa 15.8 define la clase `MemoryParameters`.

Programa 15.8. La clase Java para tiempo real `MemoryParameters`.

```
public class MemoryParameters
{
  public static final long NO_MAX;

  public MemoryParameters(long maxAreaMemoria, long maxInmortal)
         throws IllegalArgumentException;

  public MemoryParameters(long maxAreaMemoria, long maxInmortal,
                          long tasaAsignacion)
         throws IllegalArgumentException;

public long getAllocationRate();
public long getMaxImmortal();
public long getMaxMemoryArea();

public void setAllocationRate(long tasa);
public boolean setMaxImmortal(long maxima);
public boolean setMaxMemoryArea(long maxima);
}
```

Considere, por ejemplo, un hilo de tiempo real que desea asignar, por defecto, toda su memoria de la memoria inmortal. Sin embargo, durante ciertos momentos del cálculo desea crear algunos objetos temporales con un tiempo de vida bien definido. En primer lugar, se define el código para el hilo. El método `computo` representa la parte del código donde se precisa almacenamiento temporal. Éste crea cierta memoria lineal con alcance indicando el tamaño máximo y mínimo necesario. A continuación, define un `Runnable` local que contenga el cómputo real. El código `miMem.enter` limita el alcance de la memoria.

```
import javax.realtime.*;
public class CodigoDelHilo implements Runnable
{
  private void computo()
```

```
    {
        final int min = 1*1024;
        final int max = 1*1024;
        final LTMemory miMem = new LTMemory(min, max);
        miMem.enter(new Runnable()
        {
            public void run()
            {
                // codifique aquí el código que precisa acceder
                // a memoria temporal
            }
        });
    }

    public void run()
    {
        ...
        computo();
        ...
    }
}
```

Ahora podrá crearse el hilo. Dese cuenta de que en este ejemplo no se indica ningún parámetro más que el área de memoria y `Runnable`.

```
CodigoDelHilo codigo = new CodigoDelHilo();

RealtimeThread miHilo = new RealtimeThread(
        null, null, null, ImmortalMemory.instance(),
        null , code);
```

15.9.2 Gestión de la pila

Al igual que con la gestión del montón, los programadores de aplicaciones embebidas deberán implicarse en el tamaño de la pila. Mientras que la la especificación del tamaño de la pila de una tarea/hilo no requiere más que un soporte trivial (por ejemplo: en Ada se consigue mediante el atributo `Storage_Size` aplicado a una tarea, y en POSIX se obtiene mediante los atributos de pthread), el cálculo del tamaño de la pila es más difícil. Cuando las tareas entran en bloques y ejecutan procedimientos, sus pilas crecen. El estimar con precisión la máxima dimensión de su crecimiento requiere conocer el comportamiento de la ejecución de cada tarea. Este conocimiento es muy parecido al que se precisa para efectuar el análisis del tiempo de ejecución del peor caso (WCET; worst-case execution time). Por ello, tanto WCET como los límites de uso de la pila

en el peor caso posible pueden obtenerse mediante una sola herramienta para el análisis del flujo de control a partir del código de la tarea.

Resumen

Una de las principales características de un sistema embebido es que precisa interactuar con dispositivos de entrada y salida de propósito específico. La programación de controladores de dispositivos en lenguajes de alto nivel requiere:

- La posibilidad de pasar datos e información de control desde y hacia el dispositivo.
- La posibilidad de tratar (manejar) interrupciones.

Normalmente, la información de control y datos se pasa a los dispositivos a través de los registros de los dispositivos. A estos registros se accede bien mediante direcciones especiales en una arquitectura de E/S aplicada sobre memoria, o bien mediante instrucciones especiales de la máquina. El manejo de las interrupciones requiere cambio de contexto, identificación de dispositivos e interrupciones, control de interrupciones, y priorización de dispositivos.

La programación de dispositivos ha sido tradicionalmente un baluarte del programador de lenguaje de ensamblado, aunque lenguajes como C, Modula-1, occam y Ada han intentando proporcionar progresivamente mecanismos de alto nivel para estas funciones de bajo nivel. Esto permite que las rutinas de control de dispositivos y de interrupciones sean más fáciles de leer, escribir y mantener. El requisito principal para un lenguaje de alto nivel es que proporcione un modelo abstracto de manejo de dispositivos. También se precisan mecanismos de encapsulamiento para poder mantener dentro del programa la separación entre el código no portable y el código portable.

El modelo de manejo de dispositivo se construye sobre el modelo de concurrencia del lenguaje. Es posible considerar a un dispositivo como un procesador que efectúa un proceso prefijado. Así, podrá modelarse el sistema como un conjunto de procesos paralelos que necesitan sincronizarse y comunicarse. Existen varias formas en que pueden modelarse las interrupciones. Todas ellas deberán disponer de:

(1) Mecanismos para el direccionamiento y manipulación de los registros de dispositivos.

(2) Una representación adecuada para las interrupciones.

En un modelo de control de dispositivos de variables compartidas puras, el controlador y el dispositivo se comunican mediante registros compartidos del dispositivo, y la interrupción proporciona la condición de sincronización. Modula-1 proporciona al programador tal modelo. Los procesos manejadores se encapsulan en módulos de dispositivo que tienen la funcionalidad de los monitores. Se accede a los registros de los dispositivos como objetos escalares o como arrays de bits, y las interrupciones se ven como señales sobre una variable de condición.

En Ada, los registros del dispositivo pueden definirse como escalares y como tipos de registro definidos por el usuario, con un completo conjunto de posibilidades para corresponder los tipos sobre el hardware subyacente. Las interrupciones se ven como llamadas a procedimiento generadas por el hardware sobre un objeto protegido.

Solo occam2 presenta al programador una visión de control de dispositivos basada en mensajes. Los registros de los dispositivos son accedidos bajo la forma de canales especiales, denominados puertos, y las interrupciones se tratan como mensajes con contenido libre que pasan por los canales.

Java para tiempo real soporta acceso a registros de E/S sobre memoria mediante la clase `RawMemoryClass`; sin embargo, carece de poder expresivo para manipular los registros de los dispositivos. Las interrupciones se ven como eventos asíncronos.

La programación de bajo nivel también implica el tema más general de tratar con los recursos de memoria de los procesadores. En este capítulo se ha considerado tanto la gestión de la pila como la del montón. Ada no precisa de un recolector de basura (la memoria puede desasignarse explícitamente, y las reglas de alcance del lenguaje permiten la desasignación automática cuando un tipo de acceso se encuentra fuera de alcance). El lenguaje también permite al usuario definir depósitos de almacenamiento que permiten que los programadores definan sus propias políticas de gestión de memoria.

Java para tiempo real reconoce que la política de reserva de memoria de Java no es sostenible en los sistemas de tiempo real. En consecuencia, permite asignar memoria del exterior del montón, y soporta la noción de memoria con alcance, lo que permite que la memoria sea reclamada automáticamente sin la intervención del recolector de basura.

Lecturas complementarias

Barr, M. (1999), *Programming Embedded Systems in C and C++*, Sebastopol, CA: O'Reilly.

Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., y Turnbull, M. (2000) *The Real-Time Specification for Java*, Reading, MA: Addison-Wesley.

Project UDI (2000), *Uniform Driver Interface (UDI)* `http://www.project-udi.org/`, accedido en junio de 2000.

Ejercicios

15.1 Considere un computador embebido en un sistema de monitorización de pacientes (suponga el sistema de E/S simple visto en este capítulo). El sistema está preparado para que se genere una interrupción en el nivel de prioridad de hardware más elevado cada vez que

late el corazón del paciente, mediante el vector de la posición 100 (octal). Además, puede administrarse una estimulación eléctrica suave por medio de un registro de control del dispositivo, cuya dirección está en 177760 (octal). El registro está configurado de modo que cada vez que se le asigna un valor entero x, el paciente recibe x voltios durante un corto intervalo de tiempo.

Si no se registra ningún latido durante un periodo de 5 segundos, significa que la vida del paciente corre peligro. Cuando falla el corazón del paciente, deberán efectuarse dos acciones: la primera notificárselo a una tarea de «supervisión», de modo que suene la alarma hospitalaria; la segunda sería administrar una única estimulación eléctrica (choque) de 5 voltios. Si el paciente no respondiera, se incrementaría el voltaje en un voltio cada 5 segundos.

Escriba un programa Ada que monitorice el latido del paciente, e inicie las acciones anteriormente indicadas. Puede suponerse que la tarea de supervisión se proporciona siguiendo la siguiente especificación:

```
task Supervisor is
   entry Suena_Alarma;
end Supervisor;
```

15.2 El gobierno español está considerando introducir impuestos por el uso de las carreteras. Un posible sistema sería situar estaciones de detección a intervalos regulares en las vías públicas, de modo que cuando pasara un vehículo se registrarían sus detalles, almacenando el cargo correspondiente por el uso. Al final de cada mes, el propietario del vehículo recibiría una factura por la cantidad adeudada por uso de las vías públicas.

Cada vehículo precisaría de un dispositivo de interfaz que interrumpiría el computador de a bordo cuando cada estación solicitara sus detalles. El computador tiene un tamaño de palabra de 16 bits, y dispone de E/S sobre memoria con todos los registros de E/S 16 bits. Las interrupciones están vectorizadas, y la dirección del vector de interrupción asociado con la estación de detección es 8#60#. Al recibir cada interrupción, un registro de entrada de solo lectura situado en 8#177760# contiene el coste básico por utilizar el actual tramo de autovía. La prioridad hardware de la interrupción es 4.

El software del computador deberá responder a la interrupción en un intervalo de 5 segundos, y pasar su identificación a la estación de detección mediante su dispositivo de interfaz. Esto se efectúa escribiendo sobre un banco de cinco registros de control y estatus ubicados en 8#177762#. La estructura de estos registros es la que se muestra en la Tabla 15.1.

El banco de registros RCE es de solo lectura.

Escriba una tarea Ada que haga de interfaz con la estación de detección. La tarea debería responder a las interrupciones desde el dispositivo de interfaz y responsabilizarse del envío de los detalles correctos del vehículo. También debería leer el registro de datos que contiene el coste por el uso de la carretera, y transferir el coste total actual del viaje a una tarea que lo mostraría en una unidad de presentación en el salpicadero del vehículo. Se supone que se dispone de lo siguiente:

Tabla 15.1. Estructura del registro para el pago por utilización de carreteras.

Registro	Bits	Significado
1	0–7	Identificador del vehículo, carácter 1
1	8–15	Identificador del vehículo, carácter 2
2	0–7	Identificador del vehículo, carácter 3
2	8–15	Identificador del vehículo, carácter 4
3	0–7	Identificador del vehículo, carácter 5
3	8–15	Identificador del vehículo, carácter 6
4	0–7	Identificador del vehículo, carácter 7
4	8–15	Identificador del vehículo, carácter 8
5	0	establecer a 1 para transmitir datos
5	1–4	detalles del viaje
		1 = negocios
		2 = placer
		3 = turismo exterior
		4 = policía
		5 = militar
		6 = servicios de emergencia
5	5–15	código de seguridad (0–2047)

```
package Detalles_Trayecto is
   Numero_Identificacion : constant String(1..8) :=
                           ".......";

   type Detalles_Viaje is
      (Negocios, Placer, Turismo_Exterior,
      Policia, Militar, Servicio_Emergencia);

   for Detalles_Viaje use
       (Negocios => 1, Placer => 2,
       Turismo_Exterior => 3, Policia => 4,
       Militar => 5 , Servicio_Emergencia => 6);

   subtype Codigo_Seguridad is Integer range 0 .. 2047;

   function Viaje_Actual return Detalles_Viaje;
```

```
   function Codigo return Codigo_Seguridad;
end Detalles_Trayecto;

package Interfaz_Display is

   task Manejador_Display is
      entry Pon_Coste(C : Integer);
      -- muestra el coste en el display del salpicadero
   end Manejador_Display;

end Interfaz_Display;
```

Suponga que el compilador representa el tipo `Character` como un valor de 8 bits, y el tipo `Integer` como un valor de 16 bits.

15.3 Una red de área local en anillo con ranuras o franjas es aquélla que contiene ranuras en las que pueden depositarse datos para su transmisión; estas ranuras circulan continuamente por el anillo. Considere un anillo concreto con una *única* ranura, denominada paquete, cuya dimensión es de 40 bits. En la Figura 15.3 se muestra la estructura del paquete.

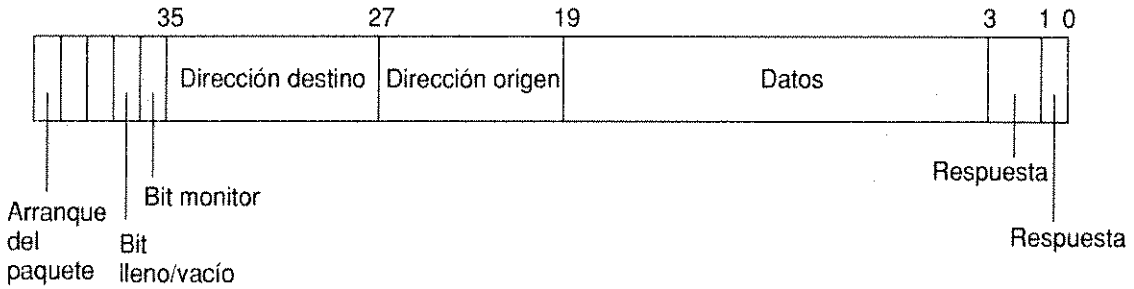


Figura 15.3. Un anillo con ranuras.

El bit 39 indica el arranque de un paquete; siempre está a 1. El bit 36 indica si el paquete está lleno o vacío (esto es, si se está utilizando la ranuras); los bits 27-34 se emplean para indicar la dirección destino de los datos; los bits 19-26 se usan para alojar la dirección origen de los datos; los bits 3-18 se usan para alojar los datos que se transmiten; y el bit 0 es un bit de paridad que podrá ignorarse en lo que nos concierne actualmente. Los bits de respuesta (bits 1-2) y el bit monitor (bit 35) se describen más adelante.

Cada proceso transmisor, al recibir un paquete vacío, establecerá el bit lleno del paquete, pondrá los bits de respuesta a cero, escribirá las direcciones origen y destino, y colocará los datos que se van a transmitir en aquél. El paquete circulará por el anillo, donde cada estación comprobará el paquete para ver si es el objeto destino del paquete. Si lo es, copiará los datos del paquete y establecerá los bits de respuesta (a 11 en binario) para indicar que el paquete está vacío. Después enviará el paquete. Si el emisor deseara enviar otro mensaje deberá esperar hasta que reciba de nuevo un paquete vacío; esto evita que un emisor monopolice el anillo.

Aunque el anillo tiene una baja tasa de error, es posible que los datos del paquete se corrompan. En concreto, es posible que se corrompa la dirección del emisor. Para evitar la posibilidad de que el paquete completo continúe circulando indefinidamente por el anillo, se incluye una estación de monitorización. La estación de monitorización lee cada paquete y establece el bit de monitor a uno. Cada estación transmisora establecerá el bit de monitor a 0 cuando transmita un paquete. Si la estación monitora lee un paquete con datos que tiene su bit de monitor establecido, entonces es que este paquete ya ha circulado por el anillo una vez, y en consecuencia su dirección se ha deteriorado. Si se encuentra esta situación de error, el bit lleno/vacío se pondrá a cero para indicar que el paquete puede volverse a emplear.

Cada interfaz de estación con el anillo está controlada por cuatro registros con E/S sobre memoria. El primer registro es un registro de control y estatus que reside en la posición octal 177760. La estructura del registro se muestra en la Tabla 15.2.

Tabla 15.2. Estructura del registro de control para el anillo con ranuras.

Bits	Significado
0	bit de paridad
1, 2	respuesta
3	bit de monitor
4	bit lleno/vacío
6	habilitación de interrupciones
10	paquete transmitido

El segundo registro y el tercero son registros con las direcciones de origen y destino que residen en las posiciones (en octal) 177762 y 177764, respectivamente. Su estructura se muestra en la Tabla 15.3.

Tabla 15.3. Estructura de los registros de dirección para un anillo con ranuras.

Bits	Significado
0-7	dirección
8-15	sin usar

El último registro es el registro de datos, que reside en la posición 177766 (octal). La llegada de un paquete se señalizará mediante una interrupción. La interrupción está vectorizada en la posición octal 60; la prioridad de la interrupción es 6. En cada interrupción, el registro de control y estatus mostrará el valor del bit lleno/vacío, el bit de monitor, los bits de repuesta y el bit de paridad. Los bits se pueden modificar escribiendo sobre el registro de control y estatus. Si el proceso de manejo de la interrupción establece el bit 10, se trans-

mitirá el paquete. De la misma forma podrán leerse y modificarse las direcciones origen y destino y los datos del contenido de paquete.

Escriba un paquete Ada que transmita y reciba enteros mediante el anillo de una sola ranura. La especificación de su paquete se da a continuación:

```
package Manejador_Anillo_En_Franjas is

   type Id_Estacion is private;
   Estacion1 : constant Id_Estacion;
   Estacion2 : constant Id_Estacion;
   Estacion3 : constant Id_Estacion;
   Estacion4 : constant Id_Estacion;•EJERCICIOS 669
   -- etcetera

   procedure Transmite (A_La_Estacion : Id_Estacion;
                        Datos : Integer);

   procedure Recibe (De_La_Estacion : out Id_Estacion;
                     Datos : out Integer);
private

   type Id_Estacion is new Short_Integer; -- 16 bits
   Estacion1 : constant Id_Estacion := 1;
   Estacion1 : constant Id_Estacion := 2;
   -- etcetera
end Manejador_Anillo_En_Franjas;
```

Puede suponerse que si los bits de respuesta en el paquete están a 0 cuando el paquete vuelve al emisor, el dato no ha sido recibido. Sin embargo, no es preciso reintentarlo; sólo habría que descartar los datos.

También podrían ignorarse las comprobaciones de paridad, y puede suponerse que un *short integer* ocupa 16 bits de memoria.

15.4 Reescriba su respuesta al Ejercicio 15.2 en occam2, Modula-1 y Java para tiempo real.

15.5 Considere un brazo robot simple conectado a un computador con un sistema de E/S sencillo, y que sólo puede moverse a lo largo del eje horizontal. El dispositivo se controla mediante dos registros: un registro de datos sobre la posición octal 177234, y un registro de control sobre la posición 177326. Cuando se habilita el dispositivo (poniendo a 1 el bit 6 del registro de control) y se indica una coordenada en el registro de datos, el brazo robot se desplaza a esa coordenada, y se genera una interrupción (en la posición octal 56 y con prioridad hardware 4) cuando el brazo está en la nueva posición.

Defina un módulo de dispositivo en Modula-1 de modo que un proceso Modula-1 pueda desplazar el brazo a una posición concreta al invocar la rutina `MUEVEAPOSICION`, definida por el módulo del dispositivo, con un parámetro que indica la posición deseada. El procedimiento debería volver cuando el brazo estuviera ubicado en la nueva posición. Puede suponerse que sólo un proceso cada vez llamará a `MUEVEAPOSICION`.

15.6 Diseñe un módulo de dispositivo en Modula-1 que permita introducir un retardo en el proceso que lo invoca, medido en pulsos de reloj. Los procesos que lo invocan interactuarán con el módulo del dispositivo mediante un procedimiento llamado `RETARDA`, que toma un parámetro entero que indica la duración del retardo medido en pulsos. El procedimiento vuelve cuando el plazo de retardo ha expirado. Puede suponerse que la prioridad del dispositivo del reloj es 6, su posición del vector de interrupción es la dirección 100 en octal, y su registro de control y estatus está en la dirección 177546 (octal) y es de 16 bits de largo. El bit 6 de este registro habilita las interrupciones cuando está a 1.

15.7 Reescriba el controlador del dispositivo de teclado dado en la Sección 15.3.3, en occam2.

15.8 Compare y contraste las restricciones que Ada y Modula-1 establecen sobre la programación de los controladores de dispositivos.

15.9 El gobierno británico está preocupado por la velocidad de los automóviles en las vías públicas. En el futuro, se plantarán balizas a intervalos regulares a lo largo de todas las carreteras; éstas transmitirán constantemente la velocidad límite actual. Los coches modernos dispondrán de computadores que monitorizarán el límite actual de velocidad e informarán al conductor cuándo excedan dicho límite.

Un coche en desarrollo actualmente, (el Yorkmobile) ya dispone de las interfaces hardware necesarias. Éstas son como sigue:

- Cada automóvil tiene un computador de «control de velocidad» de 16 bits, con E/S sobre memoria y registros de E/S de 16 bits.

- Cierto registro ubicado en la posición 177760 (en octal) hace de interfaz con con un dispositivo que monitoriza las balizas adyacentes de las carreteras. El registro siempre contiene el valor de la última limitación de velocidad recibida de dichas balizas laterales.

- Un par de registros sirven de interfaz con un dispositivo velocímetro que comprueba la velocidad del coche en relación con el límite establecido. Si se supera la velocidad límite, el dispositivo genera una interrupción por la posición 60. La prioridad de la interrupción es 5. Esta interrupción se repite cada 5 segundos hasta que el coche deja de acelerar.

- El par de registros consta de un registro de control y estatus (RCE), y de un registro de búfer de datos (RBD).

Tabla 15.4. Estructura de los registros de control del computador de velocidad.

Bits	Significado
0	0 habilita el dispositivo
1	*Si está a 1, se emplea el valor del RBD* como limitación actual de velocidad del coche
5-2	Sin usar
6	Habilita las interrupciones
11-7	Sin usar
15-12	Bits de error (0 = sin error; > 0 límite ilegal)

El registro RCE puede ser tanto leído como escrito, y reside en la dirección octal 177762. El registro RBD contiene simplemente un valor entero, que representa la velocidad límite del coche que se debe establecer. Si este valor queda fuera del rango 0-70 (mph), entonces se ha especificado un límite ilegal, y el dispositivo conserva el límite actual. La dirección del registro RBD es 177764 (octal).

- Es posible hacer parpadear cierta luz del salpicadero del coche poniendo el registro ubicado en la dirección 177750 a 1. La luz parpadeará sólo durante 5 segundos. Un valor cero apagará la luz.

Diseñe un controlador de dispositivo en occam2 que implemente el siguiente algoritmo de control de velocidad.

Cada 60 segundos el computador de velocidad deberá leer el límite de velocidad actual de la baliza del lado de la carretera. Este valor se pasa intacto al dispositivo velocímetro, que interrumpirá si la velocidad del coche excede la velocidad límite, o si el límite de velocidad es ilegal. Si el coche excede el límite, entonces la luz del salpicadero deberá parpadear hasta que el coche vuelva al límite de velocidad actual.

15.10 Rehaga el Ejercicio 15.9 usando Modula-1, Ada y Java para tiempo real.

15.11 Compare y contraste el modelo de memoria compartida de Modula-1 para el control de dispositivos con el modelo de paso de mensajes de occam-2.

Capítulo 16

El entorno de ejecución

Por su propia naturaleza, los sistemas de tiempo real deben responder oportunamente a los a eventos que ocurren en su entorno. Esto ha llevado a considerar que los sistemas de tiempo real deben ser tan rápidos como sea posible, y que no pueden tolerarse las sobrecargas introducidas por las características del lenguaje o del sistema operativo que permiten abstracciones de alto nivel (como monitores, excepciones, acciones atómicas, etc.). El término «eficiencia» se utiliza a menudo para expresar la calidad del código producido por un compilador o el nivel de abstracción proporcionado por los mecanismos soportados por un sistema operativo o un sistema de soporte de tiempo real. Este término, sin embargo, no queda bien definido. Además, la eficiencia es, en muchos casos, una métrica pobre para valorar una aplicación y su implementación. En los sistemas de tiempo real, lo realmente importante es satisfacer los tiempos límite o conseguir tiempos de respuesta adecuados en un entorno de ejecución concreto. Este capítulo considera algunos de los temas asociados con la consecución de este objetivo. Inicialmente, se examina el impacto del entorno de ejecución en el diseño e implementación de los sistemas de tiempo real. A continuación, se discuten las formas en que se puede construir el entorno de ejecución software para las necesidades de la aplicación. Después, se revisan los modelos de planificación de los núcleos para facilitar el análisis de la planificabilidad de una aplicación. Se continua mostrando cómo algunas abstracciones presentadas en este libro pueden ser realizadas por el harware en el entorno de ejecución.

16.1 El papel del entorno de ejecución

En el Capítulo 13, se consideró que el análisis de la planificabilidad es fundamental para la predicción de las propiedades del software de tiempo real. Es difícil abordar este análisis a menos que se conozcan los detalles del entorno de ejecución propuesto. La expresión «entorno de ejecución» se utiliza para designar aquellos componentes que se usan junto al código de la aplicación para completar el sistema: procesadores, redes, sistemas operativos, etc. La naturaleza del

entorno de ejecución propuesto determinará si cierto diseño satisface sus requistos de tiempo real. Evidentemente, cuanto más eficiente sea el uso del entorno de ejecución mejor se satisfarán los requisitos. Pero no siempre es así, y un diseño pobremente estructurado puede no satisfacer sus requisitos, independientemente de lo eficientemente que se implemente. Por ejemplo, un diseño que tenga una inversión de prioridad significativa fallará a la hora de satisfacer los tiempos límite alcanzables, independientemente de lo eficientemente que esté implementado –como se vio de forma tan patética en la misión Pathfinder a Marte (Jones, 1997; Reeves, 1997)–. El proceso de diseño puede verse como una progresión de **compromisos** y **obligaciones** cada vez más específicos. Los compromisos definen aquellas propiedades del diseño del sistema que los diseñadores, trabajando a un nivel más detallado, no tienen libertad para cambiar. Aquellos aspectos de un diseño para los que no se obtienen compromisos a ningún nivel concreto son el tema de las obligaciones, que deberán ser tratadas por los niveles de diseño más bajos.

El proceso de refinamiento de un diseño (una transformación de obligaciones en compromisos) está sometido normalmente a **restricciones**, impuestas sobre todo por el entorno de ejecución. También puede estar restringida la elección del entorno de ejecución y su modo de utilización. Por ejemplo, puede haber un requisito que establezca que se utilice un procesador resistente en el espacio (space-hardened), o puede haber un requisito de certificación que dicte que no se deba superar el 50 por ciento de la capacidad del procesador (o de la red).

Muchos métodos de diseño distinguen entre diseño lógico y físico. El diseño lógico se centra en la satisfación de los requisitos funcionales de la aplicación, y supone un entorno de ejecución lo suficientemente rápido. La arquitectura física es el resultado de combinar el aspecto funcional y el entorno de ejecución propuesto para producir un diseño de la arquitectura hardware y software.

La arquitectura física forma la base para afirmar que se satisfarán todos los requisitos de la aplicación una vez que se realicen el diseño detallado y la implementación. Ésta considera el análisis de la temporización (e incluso la seguridad), que asegurará (garantizará) que el sistema, una vez construido, satisfaga (con algunas hipótesis de fallo consideradas) los requisitos de tiempo real. Para realizar este análisis, será necesario realizar algunas estimaciones de la utilización de los recursos del sistema propuesto (hardware y software). Por ejemplo, se pueden realizar las estimaciones iniciales de las propiedades de temporización de la aplicación y abordar el análisis de la planificabilidad para garantizar que los tiempos límite se satisfarán una vez que se haya implementado el sistema final.

El objetivo del diseño de la arquitectura física es tomar la arquitectura funcional y transformarla sobre las funcionalidades proporcionadas en el entorno de ejecución. Cualquier desavenencia entre las suposiciones realizadas por la arquitectura funcional y las funcionalidades proporcionadas por el entorno de ejecución, debe ser consideradas durante esta actividad. Por ejemplo, la arquitectura funcional puede suponer que todas las funciones sean inmediatamente visibles para las demás. Cuando las funciones son proyectadas sobre los procesadores en la arquitectura física, puede no haber un camino de comunicación directa proporcionado por la infraestructura, y por consiguiente puede ser necesario añadir funciones extra de router en el nivel de aplicación. Además, puede que la infraestructura sólo permita el paso de mensajes de bajo nivel, mientras que las funciones pueden comunicarse utilizando llamadas a procedimientos; por lo

tanto, será necesario proporcionar una funcionalidad de RPC en el nivel de aplicación. Existe claramente un compromiso entre la sofisticación del entorno de ejecución y la necesidad de añadir funcionalidades extras de la aplicación a la arquitectura funcional durante la producción de la arquitectura física. Sin embargo, es importante también no proporcionar mecanismos sofisticados en el entorno de ejecución si no son necesarios para la aplicación, y menos aún si la aplicación necesita más funcionalidades primitivas que debe intentar construir a partir de las de alto nivel. Esto se conoce normalmente como **inversión de abstracción**.

Una vez que se han completado las actividades iniciales del diseño arquitectónico, puede comenzar en serio el diseño detallado y pueden ser producidos todos los componentes de la aplicación. Una vez que se ha realizado esto, cada componente debe ser analizado utilizando herramientas para medir características de la aplicación, como su tiempo de ejecución en el peor caso (o, por ejemplo, su complejidad si se está considerando la seguridad), para confirmar que los tiempos de ejecución estimados en el peor caso son correctos (o que ciertos módulos han resultado complejos y por lo tanto propensos a errores, motivando que se considere la diversidad en el diseño). Si estas estimaciones no fueran precisas (lo que será normal en el caso de una aplicación nueva), entonces se debe revisar el diseño detallado (si hay pequeñas desviaciones), o bien el diseñador debe volver a las actividades del diseño arquitectónico (si existen problemas serios). Si la estimación indica que todo está bien, entonces se procede a probar la aplicación. Esto implica medir la temporización actual del código, el número de errores encontrados, y así sucesivamente. El proceso está representado en la Figura 16.1 (se trata, en realidad, del ciclo de vida seguido por los métodos de diseño HRT-HOOD considerados en el Capítulo 2 y utilizados en el caso de estudio que se presenta en el Capítulo 17).

Lo importante, por tanto, no es cuánta es la eficiencia del código compilado o de las adiciones de los sistemas operativos, sino más bien que el análisis de la temporización se realice en el ciclo de vida tan pronto como sea posible. Dicho esto, un compilador fuertemente ineficiente sería claramente una herramienta inapropiada; dichas ineficiencias serían una indicación de un producto realizado pobremente.

16.2 Construcción del entorno de ejecución

Los sistemas operativos modernos, y los sistemas de soporte de ejecución asociados con lenguajes como Ada, están inundados con funcionalidad, porque intentan ser tan de propósito general como sea posible. Desde luego, si una aplicación concreta no usa ciertas funciones de un sistema operativo, sería ventajoso adaptar su forma de ejecución.

Esta posibilidad es esencial por tres razones fundamentales:

(1) Impide el uso innecesario de recursos, ya sea tiempo de procesador o memoria.

(2) Reduce la cantidad de software cuya corrección debe ser considerada durante cualquier proceso de certificación.

(3) Muchos desarrollos estándar precisan que sea eliminado el «código muerto».

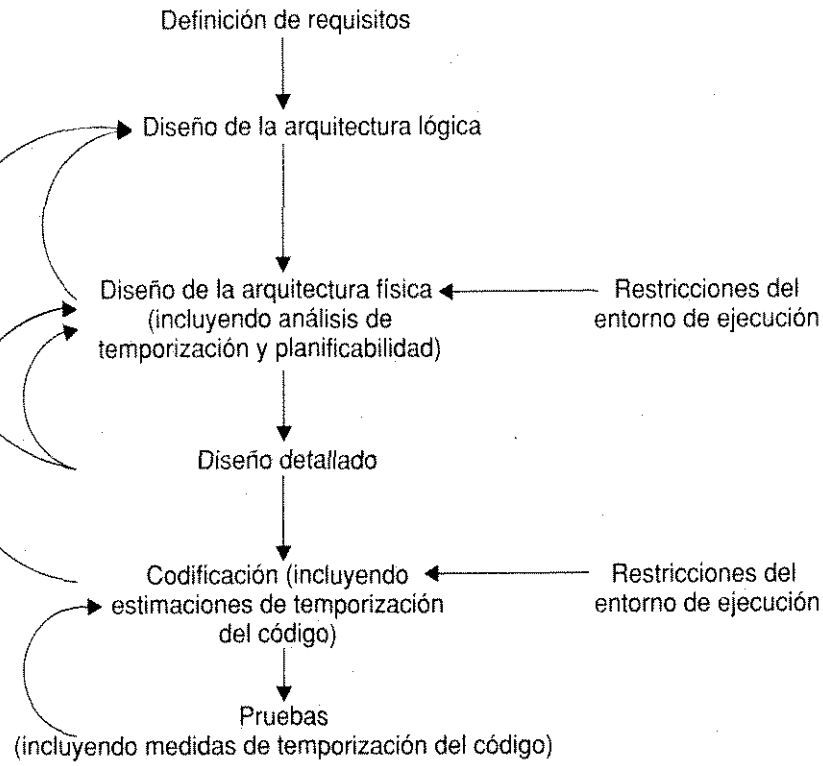


Figura 16.1. Un ciclo de vida de tiempo real estricto.

En esta sección se consideran las funcionalidades que ayudan en este proceso, y que proporcionan Ada y POSIX. Java para tiempo real no soporta, en general, componentes opcionales, ya que esto es contrario al principio de «escribir una vez, ejecutar en cualquier parte». Sin embargo, acepta que algunos componentes no puedan ser implementados si el sistema de soporte subyacente no proporciona esa funcionalidad. El caso más obvio es el de la clase que realiza la interfaz para las señales POSIX.

16.2.1 Tareas restringidas en Ada

El Anexo de Tiempo Real de Ada permite que el programador especifique un conjunto de restricciones que un sistema de ejecución debiera reconocer, y «potenciar», para proporcionar un soporte más efectivo. Los siguientes son ejemplos de restricciones que se identifican como pragmas y son comprobadas y cumplidas antes de la ejecución.

- `No_Task_Hierarchy` (sin jerarquía de tareas): esto simplifica significativamente el soporte requerido para terminación de las tareas.
- `No_Abort_Statement` (sin sentencia abort): afecta a todos los aspectos del sistema de soporte de ejecución, ya que no hay que preocuparse por que una tarea sea abortada en una cita, en un modo de operación protegido, propagando una excepción, esperando por la terminación de un hijo, etc.
- `No_Terminate_Alternatives` (sin alternativas terminate): de nuevo, simplifica el soporte requerido para terminación de las tareas.

- `No_Task_Allocators` (sin asignadores de tareas): permite configurar el sistema de ejecución con un número estático de tareas, y elimina la necesidad de asignación de memoria dinámica.

- `No_Dynamic_Priorities` (sin prioridades dinámicas): simplifica muchos aspectos del soporte para las prioridades de las tareas, como el que la prioridad no cambie dinámicamente (a diferencia de la utilización de cotas sobre las prioridades).

- `No_Asynchronous_Control` (sin control asíncrono): esto afecta a todos los aspectos del sistema de soporte a la ejecución, ya que no hay necesidad de preocuparse de una tarea que reciba un suceso asíncrono mientras está en una cita, en un modo protegido de operación, propagando una excepción, esperando por la terminación de un hijo, etc.

- `Max_Select_Alternatives` (máximo de alternativas select): permite el uso de estructuras de datos estáticas y elimina una necesidad de asignación de memoria dinámica.

- `Max_Task_Entries` (máximo de entradas a tareas): de nuevo, permite el uso de estructuras de datos estáticas y elimina la necesidad de asignación de memoria dinámica. Un valor cero indica que no se permiten citas.

- `Max_Protected_Entries` (máximo de entradas protegidas): de nuevo, permite el uso de estructuras de datos estáticas y elimina la necesidad de asignación de memoria dinámica. Un valor cero indica que no se permite la sincronización de condición para objetos protegidos.

- `Max_Tasks` (máximo de tareas): especifica el número máximo de tareas, y por tanto permite proporcionar una cantidad fija de estructuras estáticas de soporte de ejecución.

Téngase en cuanta que Ada también tiene un Anexo de Seguridad (Safety and Security Annex), que establece todas las restricciones anteriores a cero (es decir, *sin tareas*). También introduce restriciones adicionales que impiden tipos y objetos protegidos. La práctica actual en el área de aplicaciones de seguridad crítica es prohibir el uso de tareas o interrupciones. No es una opción demasiado buena, ya que impide definir un subconjunto de funcionalidades para tareas que sean predecibles y susceptibles de análisis. También es posible especificar sistemas de ejecución de forma que puedan ser implementados con un alto nivel de integridad.

Uno de los retos considerados por los programadores de Ada para la próxima década es demostrar que la programación concurrente es una técnica efectiva y segura incluso para los requisitos más rigurosos. Para este objetivo, el 8th International Real-Time Ada Workshop (Burns, 1999) definió un perfil de tareas (conocido como *perfil Ravenscar*) para utilizar en aplicaciones de alta integridad o sensibles a las prestaciones. En el perfil Ravenscar, está prohibido el uso de las siguientes funcionalidades:

- Declaraciones de tipos de tareas y objetos fuera del nivel de biblioteca. Por tanto, no hay jerarquía de tipos de tareas.

- Desasignación no comprobada de objetos protegidos y objetos tarea (y por tanto finalización). La asignación dinámica de tales objetos puede estar permitida, pero sólo si la parte

secuencial del perfil del lenguaje de alta integridad permite asignación dinámica de otros objetos.

- Reencolado.
- ATC (transferencia asíncrona de control mediante la sentencia select then abort).
- Sentencias abort.
- Entries en tareas.
- Prioridades dinámicas.
- Paquete `Calendar`.
- Retardos relativos.
- Tipos protegidos distintos de los del nivel de biblioteca.
- Tipos protegidos con más de un entry.
- Entradas protegidas con barreras distintas de las de una única variable booleana declarada dentro del mismo tipo protegido.
- Intentos de encolar más de una tarea en un único entry protegido.
- Políticas de bloqueo distintas del *bloqueo acotado*.
- Políticas de planificación distintas de *prioridades FIFO*.
- Todas las formas de la sentecia select.
- Atributos de tarea definidos por el usuario.

Además de estas restricciones, una implementación puede suponer que ninguna de las tareas del programa termine. Observe que la mayoría de estas restricciones, aunque no todas, se pueden definir utilizando el pragma `Restrictions`. Incluso con estas limitaciones, una aplicación ajustada al perfil Ravenscar tiene todavía:

- Objetos tarea restringidos, como se ha indicado anteriormente.
- Objetos protegidos restringidos, como se ha indicado anteriormente.
- Objetos suspensión.
- Pragmas atomic y volatile.
- Sentencias «delay until» (retrasar hasta).
- Política de *bloqueo acotado* y distribución con *prioridad FIFO*.
- El atributo Count (aunque no con barreras entry).

- Identificadores de tareas.
- Discriminantes de tareas.
- El paquete `Real_Time`.
- Procedimientos protegidos, como manejadores de interrupciones.

Para la exclusión mutua sencilla se proporcionan tipos protegidos sólo con interfaces de subprograma. La forma especial de entry protegido (es decir, sólo uno por objeto protegido y un máximo de un posible invocador de ese entry) está disponible como un mecanismo de señalización de eventos para permitir que se soporten tareas aperiódicas y esporádicas.

Además de las características definidas anteriormente, el Anexo de Sistemas de Tiempo Real define un número de requisitos de implementación, de documentación y de métricas. La métrica permite obtener el coste (en ciclos de procesador) del sistema de ejecución. También indica qué primitivas pueden conducir a bloqueo y cuáles no.

Las características de temporización (es decir, reloj de tiempo real y primitivas de retardo) están definidas de forma precisa. Por tanto, es posible, por ejemplo, conocer el tiempo máximo entre el valor de finalización de un retardo de tarea y el que lleva colocado en la cola de ejecución. Toda esta información es necesaria para analizar una aplicación en el contexto de su entorno de ejecución.

16.2.2 POSIX

POSIX consta de variedad de estándares. Está la base estándar, las extensiones de tiempo real, las extensiones de hilos, etc. Si se implementara en un único sistema, debería contener una enorme cantidad de software. Para ayudar a producir versiones de sistemas operativos más compactas que satisfagan las especificaciones POSIX, se ha desarrollado un conjunto de **perfiles** de aplicación; la idea es que los implementadores puedan soportar uno o más perfiles. Para sistemas de tiempo real, se han definido cuatro perfiles:

- PSE50: perfil de sistema de tiempo real mínimo. Pensado para pequeños sistemas incrustados mono/multiprocesador que controlan uno o más dispositivos externos; no se precisa interacción con el operador y no hay sistema de archivos. Sólo se soporta un único proceso con múltiples hilos.
- PSE51: perfil de sistema controlador de tiempo real. Un PSE50 extendido para procesadores potencialmente múltiples con una interfaz al sistema de archivos y E/S asíncrona.
- PSE52: perfil de sistema de tiempo real dedicado. Una extensión de PSE50 para sistemas con uno o varios sistemas procesadores con unidades de gestión de memoria; incluye múltiples procesos con múltiples hilos, pero no sistema de archivos.
- PSE53: perfil de sistema de tiempo real de multipropósito. Capaz de ejecutar una mezcla de procesos de tiempo real y no tiempo real en sistemas mono/multipocesadores con unidades de gestión de memoria, dispositivos de almacenamiento masivo, redes, etc.

La tabla 16.1 ilustra el tipo de funcionalidad proporcionada por PSE50, PSE51 y PSE52.

En general, un sistema POSIX también es libre de no soportar alguna de las unidades funcionales que elija. Todas las extensiones de tiempo real y de hilos son opcionales. Sin embargo, la conformidad con uno de los perfiles significa que se deben soportar todas las unidades funcionales precisadas.

Tabla 16.1. Perfil de funcionalidad de POSIX para tiempo real.

Funcionalidad	PSE50	PSE51	PSE52
pthreads	✓	✓	✓
fork	✗	✗	✓
semáforos	✓	✓	✓
mutexes	✓	✓	✓
paso de mensajes	✓	✓	✓
señales	✓	✓	✓
temporizadores	✓	✓	✓
ES síncrona	✓	✓	✓
ES asíncrona	✗	✓	✓
planificación de prioridad	✓	✓	✓
objetos de memoria compartida	✓	✓	✓
sistema de archivos	✗	✓	✗

Modelos de planificación

El entorno de ejecución tiene un impacto significativo en las propiedades de temporización de una aplicación. Donde haya un núcleo software, se debe tener en cuenta el gasto adicional realizado por el núcleo durante el análisis de planificabilidad de la aplicación. Las características siguientes son típicas de muchos núcleos software de tiempo real.

- No es despreciable el coste de un cambio de contexto entre procesos, y no tiene por qué ser un valor único. El coste de un cambio de contexto para un proceso periódico de prioridad alta (a continuación, por ejemplo, de una interrupción de reloj) puede ser más alto que el cambio de contexto desde un proceso a otro de prioridad más baja (al final de la ejecución del proceso de prioridad más alta). Para sistemas con un número elevado de procesos periódicos, se añadirá un coste adicional por manipulación de la cola de retardo (para tareas periódicas cuando ejecuten, digamos, una sentencia «delay until» de Ada).

- Todas las operaciones de cambio de contexto son no apropiativas.

- El coste de manejar una interrupción (distinta de la del reloj) y lanzar un proceso esporádico de la aplicación no es significativo. Además, para dispositivos controlados por DMA y programa de canal, el impacto del acceso a memoria compartida puede tener un impacto no trivial en las prestaciones del peor caso; hay que evitar dichos dispositivos en los sistemas de tiempo real estricto.
- Una interrupción de reloj (digamos cada 10 ms) puede producir el movimiento de procesos periódicos desde la cola de retardo hasta la cola de despacho. Los costes de esta operación varían dependiendo del número de procesos que se muevan. Además de lo anterior, el análisis de la planificación debe considerar las características del hardware subyacente, como el impacto de la caché y el encadenamiento de instrucciones.

16.3.1 Modelado de tiempos de cambio de contexto no triviales

La mayoría de los modelos de planificación ignoran los tiempos de cambio de contexto. Esta aproximación es, sin embargo, bastante simplista si el coste total de los cambios de contexto no es trivial en comparación con el propio código de la aplicación.

La Figura 16.2 muestra un número de eventos significativos en la ejecución de procesos periódicos típicos.

A: La interrupción de reloj que designa el tiempo hipotético en el que debiera comenzar el proceso. Suponiendo que no hay inestabilidad de lanzamiento o retardo sin desalojo, si las interrupciones se deshabilitaron debido a la operación del cambio de contexto, el manejador del reloj debería retrasar su ejecución; esto se tiene en cuenta en las ecuaciones de planificación mediante el factor de bloqueo *B*.

B: El tiempo más temprano en el que puede finalizar el manejador del reloj; esto implica el comienzo del cambio de contexto para el proceso (suponiendo que se trata del proceso ejecutable con prioridad más alta).

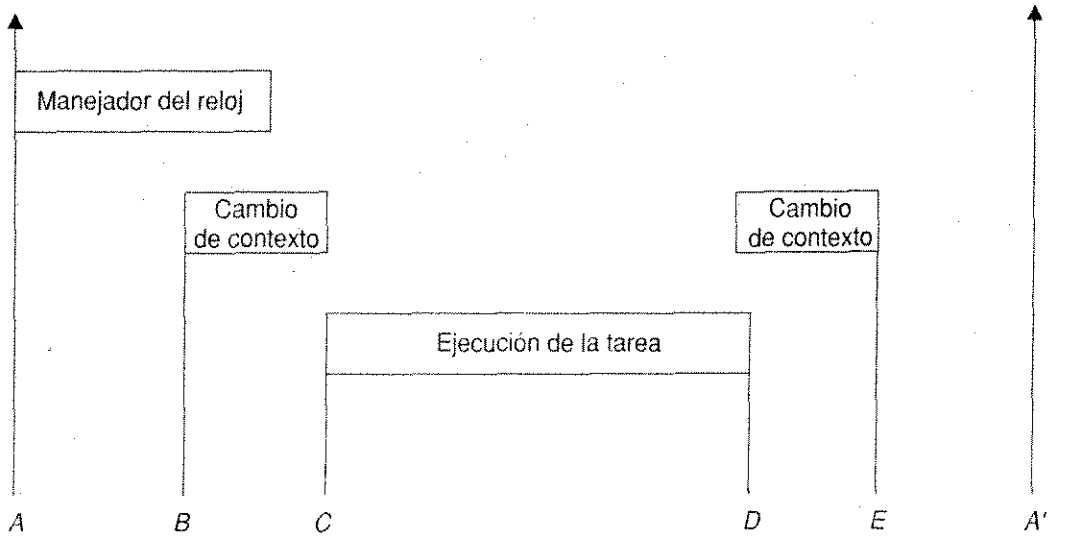


Figura 16.2. Sobrecargas en la ejecución de los procesos.

C: El comienzo actual de la ejecución del proceso.

D: La finalización del proceso (el proceso puede ser desalojado un número de veces entre C y D).

E: La finalización del cambio de contexto fuera del proceso.

A': El siguiente lanzamiento del proceso.

El requisito típico para este proceso es que finalice antes de su siguiente activación (es decir, $D < A'$), o antes de algún plazo límite anterior a su siguiente lanzamiento. Otro requisito establece un límite en el tiempo entre el comienzo de la ejecución y la terminación (es decir, $D - C$). Esto ocurre cuando la primera acción es una entrada y la última una salida (y existe un requisito de tiemplo límite entre las dos). Aunque esos factores afectan al significado del tiempo límite del propio proceso, y por tanto de su tiempo de respuesta, no afectan a la interferencia que este proceso tiene con los de prioridad más baja; aquí cuenta el coste total de ambos cambios de contexto. Recuerde que la ecuación básica de la planificación (13.7) tiene la forma:

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Esto se convierte (sólo para procesos periódicos) en:

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) \qquad \textbf{(16.1)}$$

donde CS^1 es el coste del cambio de contexto inicial (para el proceso), y CS^2 es el coste del siguiente cambio de contexto al final de su ejecución. El coste de ubicar el proceso en la cola de retardo (si es periódico) se incorpora en C_i. Hay que tener en cuenta que, en la práctica, este valor depende del tamaño de la cola; se necesitará incorporar en C_i un valor máximo.

Este tiempo de respuesta se mide desde el punto *B* de la Figura 16.2. Para medir desde el punto *C*, se elimina el primer término CS^1. Para medir desde el punto *A* (el hipotético tiempo de lanzamiento verdadero del proceso), se precisa cuantificar el comportamiento del reloj (véase la Sección 16.3.3).

16.3.2 Modelado de procesos esporádicos

Para procesos esporádicos activados por otros procesos esporádicos o por procesos periódicos, la Ecuación (16.1) es un modelo de comportamiento válido. Sin embargo, el tiempo de computación para el proceso C_i debe incluir los costes adicionales de bloqueo para el agente que controla su activación.

Cuando los procesos esporádicos son activados desde una interrupción, puede invertirse la prioridad. Incluso si el proceso esporádico tiene una prioridad baja (debido a que tiene un tiempo límite largo), la interrupción sería ejecutada a un nivel de prioridad de hardware elevado. Sea Γs el conjunto de procesos esporádicos lanzados por interrupciones. Cada fuente de interrupción

se supone que tendrá las mismas características de llegada que el proceso esporádico que lanza. La interferencia adicional que estos manejadores de interrupciones tienen en cada proceso de aplicación está dada por:

$$\sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

donde IH es el coste de manejar la interrupción (y devolver al proceso en ejecución, habiendo lanzado el proceso esporádico).

Esta representación supone que todos los manejadores de interrupciones deben tener el mismo coste; si esto no es así, entonces IH debe ser definido para cada k. La ecuación (16.1) se convierte ahora en:

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j) + \sum_{k \in \Gamma_s} \left\lceil \frac{R_i}{T_k} \right\rceil IH \qquad \textbf{(16.2)}$$

16.3.3 Modelado del manejador de reloj de tiempo real

Para soportar procesos periódicos, el entorno de ejecución debe tener acceso al reloj de tiempo real que generará interrupciones en los instantes adecuados. Un sistema ideal utilizará un temporizador de intervalos, e interrumpirá sólo cuando un proceso periódico necesite ser lanzado. La aproximación más común, sin embargo, es aquélla en la que el reloj interrumpe a intervalos regulares (digamos una vez cada 10 ms), y el manejador debe decidir si ninguno, uno o un número determinado de procesos periódicos deben ser activados. La aproximación ideal se puede modelar de una forma idéntica a la que se presentó para los procesos esporádicos (véase la Sección 16.3.2). Con el método del reloj regular, es necesario desarrollar un modelo más detallado, ya que los tiempos de ejecución del manejador del reloj pueden variar considerablemente. La Tabla 16.2 proporciona los tiempos posibles para este manejador (para un periodo de reloj de 10 ms). Tenga en cuenta que si se supuso que el peor caso ocurriría en todas las ocasiones sobre el 100 por ciento que el procesador debería estar asignado al manejador del reloj. Por otro lado, todo este cómputo ocurre a un nivel alto de prioridad hardware (el más alto), y por tanto se produce una notable inversión de la prioridad. Por ejemplo, con las figuras dadas en la tabla, con el MCM (mínimo común múltiplo) de 25 procesos periódicos se deberían sufrir 1.048 μs de interferencia por los procesos de aplicación de prioridad más alta que fueron lanzados. Si el proceso se activó por sí mismo, entonces sólo debería sufrir 88 μs. El intervalo de tiempo se representa por $B - A$ en la Figura 16.2.

En general, el coste de mover N procesos periódicos desde la cola de retardo hasta la de ejecutables puede representarse por la fórmula siguiente:

$$C_{clk} = CT^c + CT^s + (N-1)CT^m$$

donde CT^c es el coste constante (suponiendo que siempre hay al menos un proceso en la cola de retardo), CT^s es el coste de hacer un único movimiento, y CT^m es el coste de cada movimiento

Tabla 16.2. Costes adicionales por el manejo del reloj.

Estado de la cola	Tiempo de manejo del reloj, μs
No hay procesos en la cola	16
Hay procesos en la cola pero ninguno eliminado	24
Un proceso eliminado	88
Dos procesos eliminados	128
Veinticinco procesos eliminados	1048

consecuente. Este modelo es apropiado si observamos que el coste de mover sólo un proceso es, a veces, alto cuando se compara con el coste de mover procesos extra. Con el núcleo aquí considerado, esos costes fueron:

CT^c	24 μs
CT^s	64 μs
CT^m	40 μs

Para reducir el pesimismo de suponer que se consume un coste computacional de C_{clk} en cada ejecución del manejador del reloj, esta carga puede ser difundida sobre un número de pulsos de reloj. Esto es válido si el periodo más corto de cualquier proceso de aplicación, T_{min} es más grande que el periodo de reloj T_{clk}. Por tanto, M está definida por:

$$M = \left\lceil \frac{T_{min}}{T_{clk}} \right\rceil$$

Si M es mayor que 1, entonces la carga del manejador del reloj puede ser extendida sobre M ejecuciones. En esta situación, el manejador de reloj se modela como un proceso con periodo T_{min}, y tiempo de computación C'_{clk}:

$$C'_{clk} = M(CT^c + CT^s) + (N - M)CT^m$$

Esto supone que $M <= N$.

La ecuación (16.2) se convierte ahora en:

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j)$$

$$+ \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

$$+ \left\lceil \frac{R_i}{T_{min}} \right\rceil C'_{clk} \qquad \textbf{(16.3)}$$

Para proporcionar mejoras adicionales (al modelo) se precisa una representación más exacta de la ejecución actual de los manejadores de reloj. Por ejemplo, utilizando CT^c y CT^s se puede derivar fácilmente la ecuación siguiente:

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j)$$

$$+ \sum_{k \in \Gamma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH + \left\lceil \frac{R_i}{T_{clk}} \right\rceil CT_c$$

$$+ \sum_{g \in \Gamma p} \left\lceil \frac{R_i}{T_g} \right\rceil CT_s \qquad \textbf{(16.4)}$$

donde Γp es el conjunto de los procesos periódicos.

Se deja como ejercicio para el lector la incorporación del modelo de manejo de reloj de tres parámetros (véase el Ejercicio 16.2).

16.3.4 El impacto de la caché en el análisis del tiempo de ejecución del peor caso

Ya se han mencionado en la Sección 13.12.1 los problemas de emprender el análisis WCET para procesos que se ejecutan en procesadores modernos. En particular, es necesario modelar el comportamiento de la caché y el encadenamiento de instrucciones del procesador. En la Ecuación (16.4) se ven afectados los valores C_i y C_j. Si esos valores se calculan siguiendo un análisis detallado de la arquitectura del procesador, se necesita tener en cuenta los desalojos producidos por las interrupciones en las ecuaciones de planificación. Por lo demás, los valores utilizados serán optimistas. Por suerte, para los sistemas de tiempo real estricto es necesario establecer límites en la frecuencia con la que pueden ocurrir las interrupciones. Cada manejador de interrupciones se trata como un proceso esporádico, de la misma forma que un proceso periódico de la prioridad más alta. La Ecuación (16.4) ya especifica el número de desalojos que pueden ocurrir mientras se está ejecutando el proceso *i*. Es simplemente el número de veces que cada proceso de prioridad más alta se puede reactivar durante el tiempo de respuesta del proceso *i*. Cada desalojo limpiará potencialmente la caché y el conducto de instrucciones. Esto lleva a las aproximaciones siguientes para integrar las penalizaciones de desalojo. Supongamos que C_i es el valor del peor caso calculado utilizando modelos que tiene en cuenta los beneficios conseguidos por las cachés y el encadenamiento de instrucciones (pipelining) en *ausencia* de interrupción. Calcular γ, la máxima penalización posible que se puede derivar de una interrupción, será el tiempo necesario para rellenar la caché y el encadenamiento. La Ecuación (16.4) se puede modificar ahora para calcular el efecto de las interrupciones en el proceso *i* (Busquets y Wellings, 1996):

$$R_i = CS^1 + C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil (CS^1 + CS^2 + C_j)$$

$$+\sum_{k\in\Gamma s}\left\lceil\frac{R_i}{T_k}\right\rceil(IH+\gamma)+\left\lceil\frac{R_i}{T_{clk}}\right\rceil(CT_c+\gamma)$$

$$+\sum_{g\in\Gamma p}\left\lceil\frac{R_i}{T_g}\right\rceil CT_s \qquad (16.5)$$

Naturalmente, es bastante pesimista, porque no todos los desalojos necesitarán el reinicio de la caché. Además, algunos bloques de memoria que se han reemplazado tendrían que haberlo sido en cualquier caso. Una aproximación menos pesimista intenta identificar el número de bloques de la caché.

16.4 Soporte hardware

Cuando se introducen procesos concurrentes en la solución de cualquier problema de tiempo real, se producen los costes adicionales de planificación, comunicación entre procesos, y otros. La Sección 16.3.3 ha intentado modelar estos costes adicionales en el análisis de la planificabilidad. Se han realizado varios intentos para reducir dichos costes adicionales proporcionando soporte directo del hardware. Esta sección considera brevemente dos núcleos hardware. El primero es el transputer que se diseñó para ejecutar programas en occam2 de forma eficiente, y el segundo es el coprocesador de tareas de Ada (ATAC; Ada Tasking Coprocessor).

En los últimos años, se han realizado iniciativas para soportar la máquina virtual de Java directamente en hardware (por ejemplo el procesador picoJava de Sun Microsystems (Sun Microsystems, 2000) o el aJ-100 de aJile Systems Inc (aJile Systems, 2000)). Este soporte va más allá del soporte de la ejecución concurrente, e intenta mejorar las prestaciones en la interpretación del byte code Java.

16.4.1 El transputer y occam2

El transputer fue diseñado como una máquina occam2 que, en un único chip, tenía un procesador de 32 bits, un coprocesador de coma flotante de 64 bits, memoria interna, y un número de enlaces de comunicación para conexión directa con otros transputers. Un bus de direcciones reúne la memoria externa con la interna mediante un espacio de direcciones continuas. Normalmente, un transputer tendrá 16 kbytes de memoria interna; ésta actúa, en efecto, como un conjunto de registros no compartidos para la ejecución de procesos.

Los enlaces se conectan al procesador principal mediante interfaces de enlace. Estas interfaces pueden gestionar las comunicaciones independientemente del enlace (incluyendo el acceso directo a memoria). Como resultado, un transputer puede comunicar con todos los enlaces (en ambas direcciones), ejecutar un proceso interno, y realizar una operación de coma flotante simultáneamente.

El transputer tiene un conjunto de instrucciones reducido, pero con una pila de operaciones de sólo tres registros. Cada instrucción ha sido diseñada para ser usada en la fase de generación de código del compilador occam2; la programación directa en ensambaldor, aunque está permitida, no se ha considerado en el diseño del conjunto de instrucciones. Aunque hay un conjunto reducido de intrucciones máquina, no todas ellas están disponibles directamente; las que son directamente accesibles son precisamente aquéllas que se generan normalmente desde los programas occam2 reales.

Lamentablemente, el transputer sólo soporta un modelo de prioridad limitado. Pero gracias a esta restricción, se puede construir un sistema de soporte de ejecución que está fundido en el chip. El resultado de esta arquitectura (más el axioma de que los cambios de contexto sólo tienen lugar cuando la pila de operaciones está vacía) es un tiempo de cambio de contexto muy pequeño.

Aunque las características operativas de un único transputer son impresionantes, sólo cuando se agrupan varios es cuando se contempla su potencial total. Los transputers utilizan comunicación punto a punto, lo que tiene la desventaja de que un mensaje debe ser enviado hacia su destino por medio de intermediarios si no hay enlace disponible. A pesar de todo, las tasas de transferencia de enlace son muy elevadas y las tasas de fallos de transmisión muy bajas, lo que proporciona una máquina para tiempo real de potencia y fiabilidad considerables.

16.4.2 ATAC y Ada

Se han realizado varios intentos de producir máquinas Ada –por ejemplo Ericsson (1986); Runner y Warshawsky (1988)–. La considerada aquí es un coprocesador de tareas Ada (ATAC) diseñado por Roos (1991).

ATAC es un disposistivo hardware diseñado para soportar los modelos de tareas y reloj de Ada 83. También anticipó algo de las características de Ada 95, como el soporte para la herencia de prioridad y «delay until». Su objetivo es eliminar de la CPU de la aplicación el peso de soportar la gestión de tareas Ada, permitiendo, por tanto, que se ejecuten las tareas de forma eficiente sin los costes adicionales en los que incurre normalmente el sistema de soporte de ejecución de Ada.

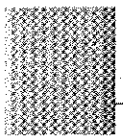
La comunicación entre la CPU y ATAC está basada en instrucciones de lectura y escritura en memoria estándar. La interfaz proporciona un conjunto de operaciones primitivas que incluye:

- `CreateTask`: crear una nueva tarea.
- `ActTasks`: activar una o más tareas creadas.
- `Activated`: activación de la señal para crear tarea.
- `EnterTBlock`: introducir un nuevo bloque de tareas.
- `ExitTBlock`: esperar a que las tareas dependientes salgan de un bloque de tareas.
- `EntryCall`: hacer una llamada a una entry.
- `TimedECall`: hacer una llamada a una entry temporizada.

- `SelectArg`: abrir una alternativa select.
- `SelectRes`: elegir una alternativa en un select.
- `RndvCompl`: hacer ejecutable al invocador tras la finalización de la cita.
- `Activate`: hacer ejecutable una tarea suspendida.
- `Suspend`: suspender la tarea actual.
- `Switch`: efectuar una replanificación.
- `Delay`: retrasar una tarea.

El ATAC también considera todas las interrupciones, e interrumpe la CPU sólo si una tarea de prioridad más alta llega a ser ejecutable. Se utiliza un temporizador interno para dar soporte a las funcionalidades de retardo de Ada y al paquete `calendar`.

El objetivo global de ATAC es aumentar las prestaciones de las tareas de Ada en dos órdenes de magnitud sobre un sistema de ejecución software puro.



Resumen

El entorno de ejecución es un componente básico de cualquier sistema de tiempo real implementado. Soporta la aplicación, pero también introduce costes adicionales y restringe las funcionalidades que puede utilizar la aplicación. Se podría utilizar un sistema operativo normal (OS) para proporcionar un entorno de ejecución, pero esto se rechaza habitualmente por:

- El tamaño del OS (es decir, la ocupación de la memoria).
- La eficiencia de las funciones clave (como el cambio de contexto).
- La complejidad, y por tanto la fiabilidad, del OS completo.

En este capítulo se ha mostrado cómo se puede construir un entorno de ejecución para las necesidades específicas de una aplicación, cómo se pueden modelar sus costes adicionales, y cómo se puede proporcionar soporte hardware. En otras partes del libro se han introducido también temas de importancia para el entorno de ejecución. Por ejemplo:

- Su papel en el confinamiento de los daños (es decir, como cortafuegos).
- Su papel en la detección de errores.
- Su papel en la facilitación de las comunicaciones en un sistema distribuido.

El segundo tema tiene varias facetas. Se pueden monitorizar varios aspectos de la ejecución de la aplicación (violación de los límites en arrays, violación de memoria, o excesos de tiempo). También se pueden ejecutar funcionalidades de «pruebas incrustadas» (built-in test) en el modo back-

ground para probar partes del hardware con el fin de aislar componentes defectuosos y generar datos de mantenimiento para eliminar fallos.

Como muchas funcionalidades de un entorno de ejecución son importantes para un amplio rango de aplicaciones, hay una necesidad de reutilizar componentes garantizados e ir hacia la provisión de entornos estándar. El uso de lenguajes e interfaces de sistemas operativos estandarizados ayudará a conseguirlo.

Lecturas complementrias

Allen, R. K., Burns, A., y Wellings, A. J. (1995), «Sporadic Tasks in Hard Real-Time Systems», *Ada Letters*, XV(5), 46–51.

Burns, A., Tindell, K., y Wellings, A. J. (1995), «Effective Analysis for Engineering Real-Time Fixed Priority Schedulers», *IEEE Transactions on Software Engineering*, 21(5), 475-480.

Venners, B. (1999), *Inside the Java 1.2 Virtual Machine*, New York: Osborne McGraw-Hill.

Ejercicios

16.1 ¿Debiera preocuparse el programador de sistemas de tiempo real de los costes de implementación de las funcionalidades del lenguaje?

16.2 Desarrolle un modelo de manejador de reloj que incorpore los parámetros CT^c, CT^s y CT^m (veáse la Sección 16.3.3).

16.3 En lugar de utilizar una interrupción de reloj para planificar procesos periódicos, ¿cuales deberían ser las consecuencias de tener un solo acceso a un reloj de tiempo real?

16.4 Un proceso periódico de periodo 40 ms está controlado por una interrupción de reloj que tiene una granularidad de 30 ms. ¿Cuál puede ser el tiempo de respuesta del peor caso calculado de este proceso?

Capítulo 17

Un caso de estudio en Ada

En este capítulo se presenta un caso de estudio que incluye muchas de las características descritas en este libro. Idealmente, el estudio debería presentarse en Ada, Java para tiempo real, C (y POSIX) y occam2. Desafortunadamente el espacio es limitado, y el estudio se restringe únicamente a Ada.

17.1 Drenaje de una mina

El ejemplo elegido está basado en otro que aparece normalmente en la literatura. Tiene que ver con el software necesario para gestionar un sistema simplificado de control de una bomba para una mina (Kramer et al., 1983; Sloman y Kramer, 1987; Shrivastava et al., 1987; Burns y Lister, 1991; Joseph, 1996; de la Puente et al., 1996), y posee muchas de las cualidades que caracterizan a los sistemas de tiempo real embebidos. Se supone que el sistema será implementado sobre un único procesador con arquitectura de I/O asignada a memoria.

El sistema se utiliza para bombear a la superficie el agua extraída del sumidero del pozo de una mina. El principal requisito de seguridad es que la bomba no debe funcionar cuando el nivel de gas metano en la mina sea alto, debido al riesgo de explosión. En la Figura 17.1 se presenta un diagrama esquemático del sistema.

En la Figura 17.2 se muestra la relación entre el sistema de control y los elementos externos. Observe que sólamente los sensores de agua inferior y superior se comunican por medio de interrupciones (indicado con flechas discontinuas); el resto de dispositivos son o bien sondeados, o bien controlados directamente.

17.1.1 Requisitos funcionales

Los requisitos funcionales del sistema se pueden dividir en cuatro componentes: el accionamiento de la bomba, la monitorización del entorno, la interacción con el operador y la monitorización del sistema.

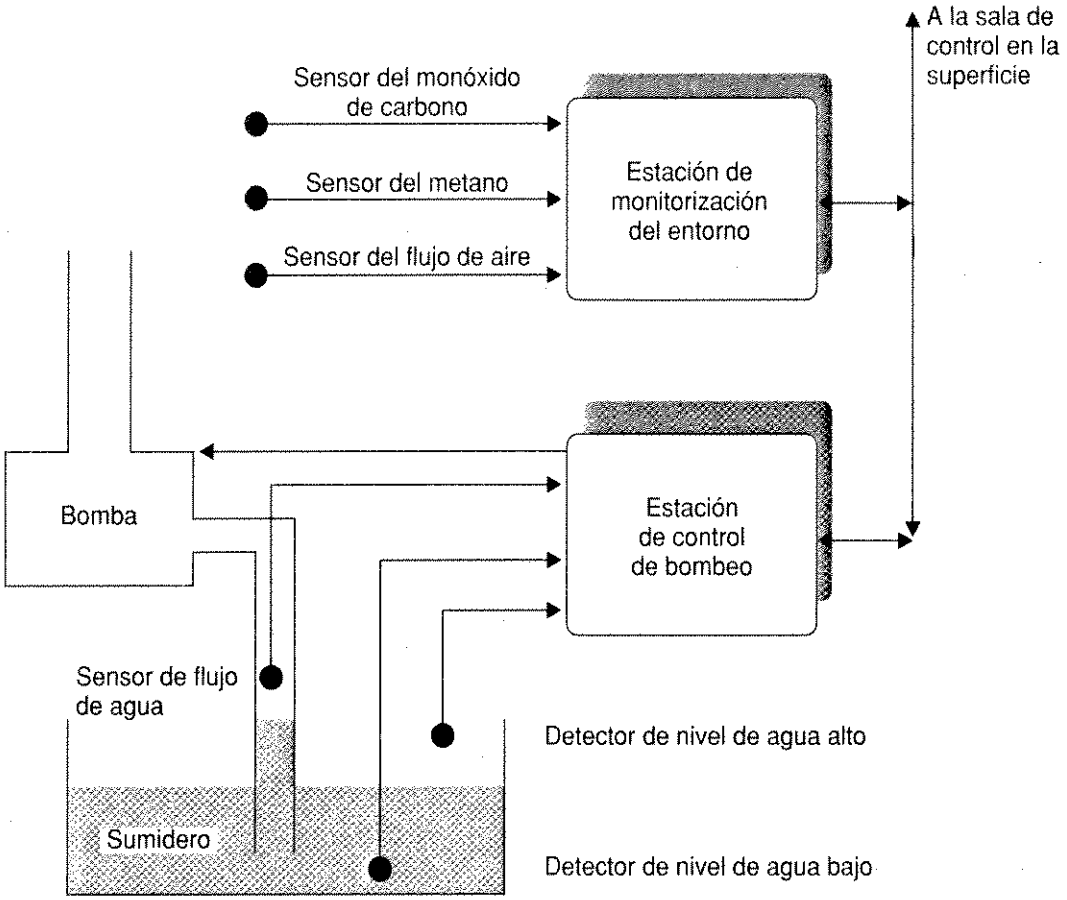


Figura 17.1. Un sistema de control de drenaje de una mina.

Accionamiento de la bomba

El controlador de la bomba necesita monitorizar los niveles de agua en el sumidero. Cuando el agua alcanza el nivel superior (o cuando lo solicita el operador), se enciende la bomba y el sumidero es drenado hasta que el agua alcanza el nivel inferior. En este punto (o cuando lo solicita el operador) se apaga la bomba. Puede detectarse el flujo de agua en las tuberías si es preciso.

Solamente si el nivel de metano está por debajo de un nivel crítico, se permite el funcionamiento de la bomba.

Monitorización del entorno

El entorno debe ser monitorizado para detectar el nivel de metano en el aire, ya que existe un nivel por encima del cual no es segura la extracción del carbón o el funcionamiento de la bomba. La monitorización también mide el nivel de monóxido de cabono en la mina y detecta si hay un flujo de aire adecuado. Si los niveles de gas o el flujo de aire llegan a ser críticos, se debe señalizar con alarmas.

Interacción con el operador

El sistema se controla desde la superficie mediante la consola del operador. El operador es informado de todos los eventos críticos.

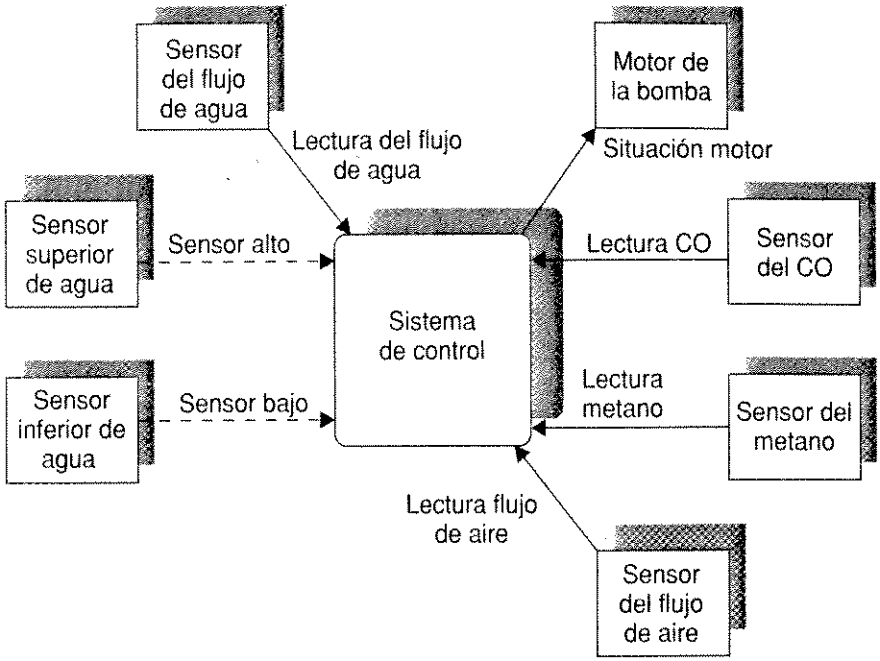


Figura 17.2. Gráfico que muestra los dispositivos externos.

Monitorización del sistema

Todos los eventos del sistema son guardados en un archivo de base de datos, de donde pueden ser recuperados y mostrados bajo demanda.

17.1.2 Requisitos no funcionales

Los requisitos no funcionales pueden referirse a tres componentes: tiempo, confiabilidad y seguridad. Este caso de estudio se centra en los requisitos temporales, y no trata los de confiabilidad y seguridad –véase Burns y Lister (1991) para una consideración completa de los aspectos de confiabilidad y seguridad–.

Respecto a temporización de las acciones del sistema, existen diversos requisitos. La siguiente lista es una adaptación de Burns y Lister (1991):

(i) Periodos de monitorización

Los máximos periodos de lectura de los sensores de entorno pueden venir fijados por ley. Para este ejemplo, se asume que estos periodos son iguales para todos los sensores, concretamente 100 ms. Para el caso del metano, puede que haya requisitos más estrictos basados en la proximidad de la bomba y en la necesidad de asegurar que nunca funcione cuando el nivel de metano sea crítico. Esto se discute en (ii). En la Sección 15.8 se describió cómo se puede analizar un controlador de dispositivo. En este caso se utilizará la aproximación de «desplazamiento del periodo» para los sensores de CH_4 y CO. Para que la lectura llegue a estar disponible, cada uno de estos sensores de entorno necesita 40 ms. Por tanto, necesitan un tiempo límite de 60 ms.

El objeto del flujo de agua se ejecuta periódicamente, y tiene dos funciones. Mientras la bomba está en funcionamiento, comprueba que hay flujo de agua; pero cuando la bomba está apagada (o desactivada), también comprueba que el agua ha dejado de fluir. Esta última comprobación es una confirmación de que la bomba se ha parado. A este objeto se le da un periodo de 1 segundo, dado que hay un retardo en el flujo del agua, y el estado real de la bomba se determina con los resultados de dos lecturas consecutivas. Para asegurarse de que dos lecturas consecutivas están realmente separadas en un segundo (aproximadamente), al objeto se le da un tiempo límite ajustado de 40 ms (esto es, dos lecturas estarán separadas en al menos 960 ms, pero no en más de 1.040 ms).

Se supone que los detectores de nivel de agua están dirigidos por eventos y que el sistema debe responder en 200 ms. La física de la aplicación muestra que debe haber al menos 6 segundos entre las interrupciones de los indicadores de los dos niveles de agua.

(ii) Tiempo límite de parada

Para evitar explosiones, la bomba debe ser apagada dentro de un tiempo límite desde que el nivel de metano sobrepasa el umbral crítico. Este límite está relacionado con el periodo de muestreo del metano, con la velocidad a la que se acumula el metano, y con los márgenes de seguridad entre el nivel crítico y el nivel al que explota. Con una lectura directa del sensor, la relación puede ser expresada por la inecuación:

$$R(T+D) < M$$

donde:

R es la tasa de acumulación del metano

T es el periodo de muestreo

D es el tiempo límite de parada

M es el margen de seguridad.

Si se utiliza «desplazamiento de periodo» se necesita un periodo de tiempo adicional:

$$R(2T+D) < M$$

Adviértase que el periodo, T, y el tiempo límite, D, pueden compensarse uno con otro, y los dos pueden ser compensados con el margen de seguridad, M. Cuanto mayor sea el periodo o el tiempo límite, más conservador debe ser el margen de seguridad; cuanto menor sea el periodo o el tiempo límite, más cerca de sus límites de seguridad puede operar la mina. El diseñador puede modificar D, T o M, siempre que se satisfaga el tiempo límite y los requisitos de periodicidad.

Para este ejemplo, se supone que la presencia de bolsas de metano puede producir incrementos rápidos de nivel, y por tanto se asume un requisito de tiempo límite (desde que el metano sube hasta que la bomba es desactivada) de 200 ms. Esto se puede obtener con una configuración del sensor de metano a 80 ms con un tiempo límite de 30 ms. Este nivel asegurará que se obtie-

nen lecturas correctas del sensor (esto es, que el desplazamiento entre dos lecturas es al menos de 50 ms).

(iii) Tiempo límite de información al operador

El operador debe ser informado en 1 segundo de la existencia de lecturas críticamente altas de metano o de monóxido de carbono, en 2 segundos de lecturas críticamente bajas de flujo de aire, y en 3 segundos de un fallo en el funcionamiento de la bomba. Comparados con el resto de requisitos temporales, *éstos son fáciles de cumplir.*

En resumen, la Tabla 17.1 define los periodos, o tiempos mínimos, entre llegadas («periodo»), y los tiempos límite (en milisegundos) para los sensores.

Tabla 17.1. Atributos de entidades periódicas y esporádicas.

	Periódico/esporádico	Periodo	Tiempo límite
Sensor CH_4	P	80	30
Sensor CO	P	100	60
Flujo de aire	P	100	100
Flujo de agua	P	1000	40
Detectores de nivel de agua	E	6000	200

17.2 El método de diseño HRT-HOOD

En el Capítulo 2 se presentó el proceso de desarrollo HRT-HOOD, que se centra en el diseño de las arquitecturas lógica y física, y en el uso de una notación orientada al objeto. En este capítulo se utiliza una versión simplificada de este método.

HRT-HOOD facilita el diseño de la arquitectura lógica de un sistema, ya que proporciona distintos tipos de objetos, que son:

- **Pasivos**: objetos pasivos que no tienen ningún control sobre la ejecución de sus operaciones, y que no invocan espontáneamente operaciones en otros objetos.
- **Activos**: objetos que pueden controlar cuándo se ejecutan las invocaciones de sus operaciones, y que pueden invocar espontáneamente a operaciones de otros objetos. Los objetos activos son la clase más general de objetos, y no incluyen ninguna restricción.
- **Protegidos**: objetos que pueden controlar cuándo se ejecutan las invocaciones de sus operaciones, pero que no invocan espontáneamente a operaciones de otros objetos; en general, los objetos protegidos pueden no tener restricciones arbitrarias de sincronización, y deben poder ser analizables los momentos de bloqueo que imponen sobre quienes les invocan.

- **Cíclicos**: objetos que representan actividades periódicas, que pueden invocar espontáneamente operaciones de otros objetos, y que sólo tienen interfaces muy restrictivas.

- **Esporádicos**: objetos que representan actividades esporádicas; pueden invocar espontáneamente operaciones de otros objetos; cada objeto tiene una **única** operación, que es llamada para invocar a la actividad esporádica.

Un programa complejo de tiempo real diseñado utilizando HRT-HOOD contendrá en el nivel terminal (esto es, después de una descomposición completa del diseño) únicamente objetos cíclicos, esporádicos, protegidos y pasivos. Los objetos activos sólo se permiten como actividades de segundo plano, ya que no pueden ser completamente analizados. Los tipos de objetos activos se utilizan durante la descomposición del sistema principal, pero deben ser transformados en alguno de los tipos anteriores antes de alcanzarse el nivel terminal.

En la Figura 17.3 se muestra la representación diagramática de un diseño HRT-HOOD. Muestra la descomposición jerárquica de un objeto «Padre» en dos objetos hijos («Hijo1» e «Hijo2»). El objeto «Padre' es un objeto activo (indicado por la letra «A» en la esquina superior izquierda del objeto), y tiene dos operaciones: «Operación1» y «Operación2». Cada uno de los objetos hijo implementa la funcionalidad de una de las operaciones. Para implementar la funcionalidad de «Operación1», «Hijo1» utiliza las posibilidades que proporciona «Hijo2» (un objeto pasivo) y un «Tío». Un objeto tío es aquél que se define en un nivel superior de descomposición. El diagrama también muestra el flujo de datos y de excepciones.

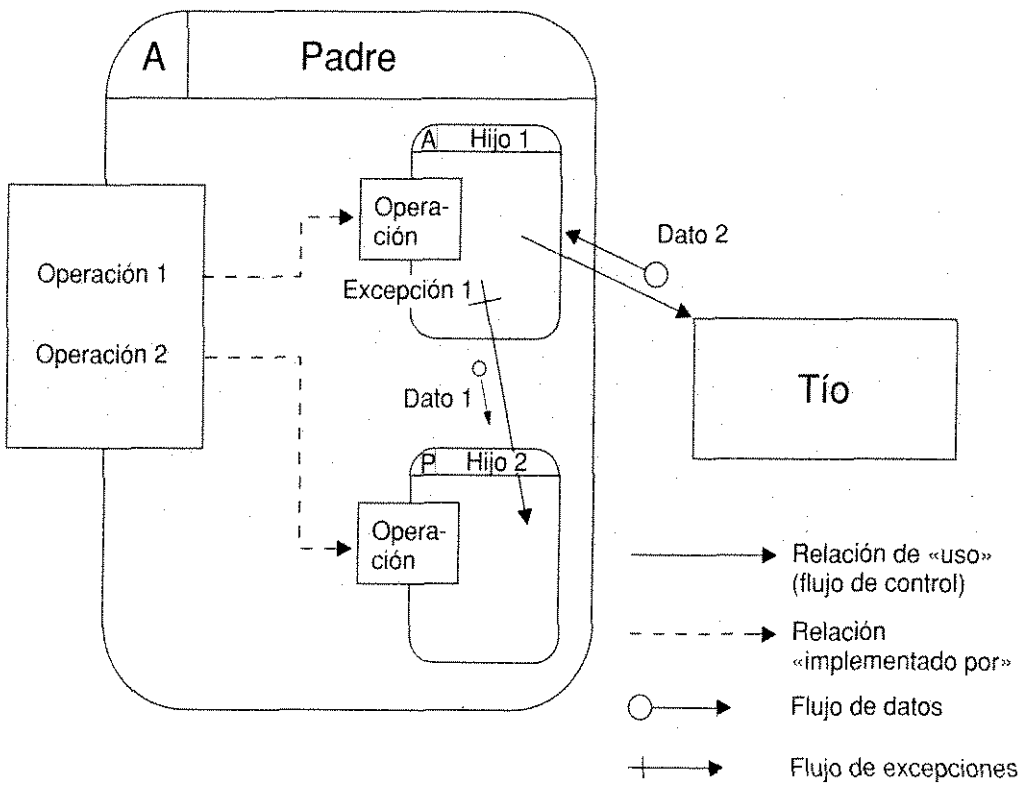


Figura 17.3. Notación diagramática HRT-HOOD.

17.3 El diseño de la arquitectura lógica

La arquitectura lógica trata aquellos requisitos que son independientes de las restricciones físicas (por ejemplo, la velocidad del procesador) que impone el entorno de ejecución. Los requisitos funcionales identificados en la Sección 17.1.1 pertenecen a esta categoría. La consideración del resto de requisitos del sistema se aplaza hasta el diseño de la arquitectura física, que se describe posteriormente.

17.3.1 Descomposición de primer nivel

El primer paso en el desarrollo de la arquitectura lógica es la identificación de las clases de objetos apropiadas que constituyen el sistema. Los requisitos funcionales del sistema sugieren cuatro subsistemas distintos:

(1) Subsistema controlador de la bomba: responsable del funcionamiento de la bomba.

(2) Subsistema de monitorización del entorno: responsable de la monitorización del entorno.

(3) Subsistema de consola del operador: responsable de la interfaz de los operadores.

(4) Subsistema de registro de datos: responsable de registrar datos operacionales y de entorno.

La Figura 17.4 muestra esta descomposición. El controlador de la bomba tiene cuatro operaciones: el monitor del entorno llama a las operaciones «no seguro» y «es seguro» para indicar al controlador de la bomba si el funcionamiento de la bomba es o no seguro (debido al nivel de metano en el entorno). Las operaciones de «petición estatus» y «establecer bomba» son invocadas desde la consola del operador. Como medida adicional de fiabilidad, el controlador de la bomba siempre comprobará antes de arrancar la bomba que el nivel de metano es bajo (llamando a «comprobar seguridad» del monitor de entorno). Si el controlador de la bomba encuentra que la bomba no puede ser arrancada (o que el agua no fluye cuando la bomba está teóricamente encendida), entonces genera una alarma de operador.

El monitor del entorno tiene una única operación, «comprobar seguridad», que es invocada por el controlador de la bomba.

La consola del operador tiene una operación de alarma que, además de por el controlador de la bomba, también es invocada por el monitor del entorno si cualquiera de sus lecturas es demasiado elevada. Además de recibir las llamadas de alarma, la consola del operador puede pedir el estatus de la bomba e intentar invalidar los sensores de agua inferior y superior, haciendo funcionar directamente a la bomba. Sin embargo, en este último caso todavía se realiza una comprobación de metano, con la utilización de una excepción que informaría al operador de que la bomba no puede ser encendida.

El registrador de datos tiene seis operaciones que simplemente registran datos, y que son llamadas por el controlador de la bomba y por el monitor del entorno.

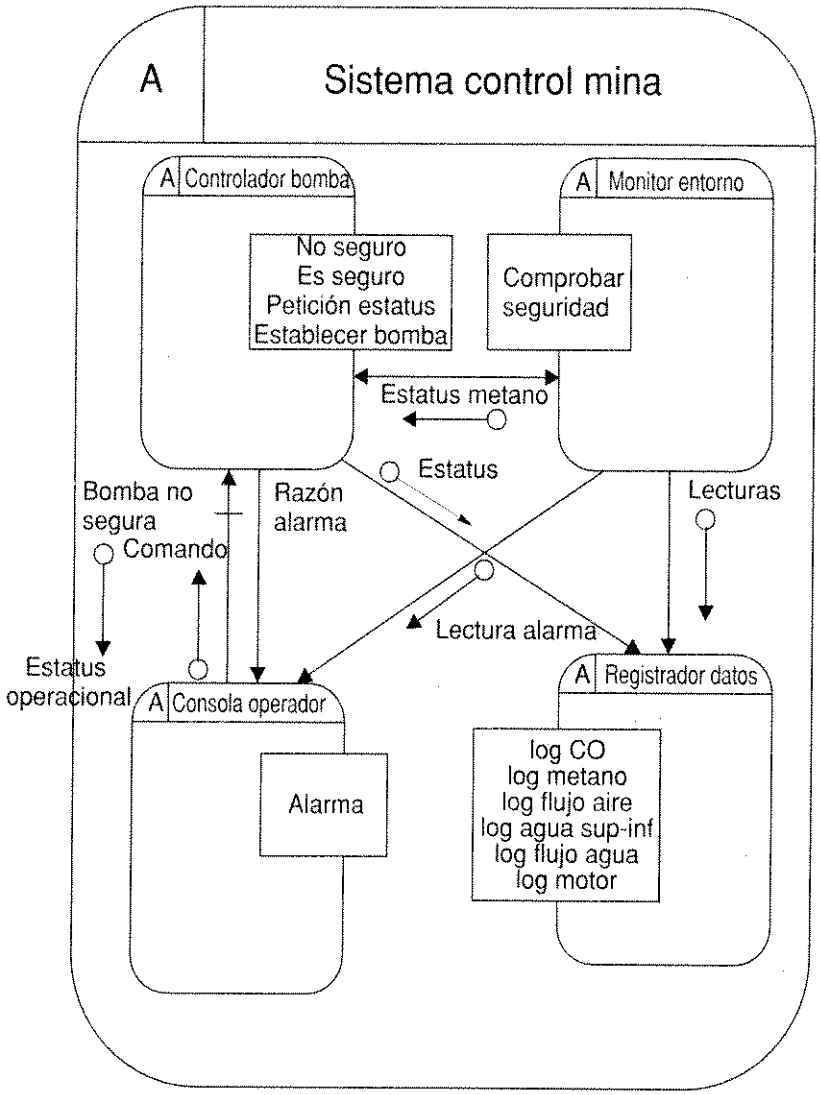


Figura 17.4. Descomposición jerárquica de primer nivel del sistema de control.

17.3.2 Controlador de la bomba

En la Figura 17.5 se muestra una descomposición aceptable del controlador de la bomba. El controlador de la bomba se descompone en tres objetos. El primer objeto controla el motor de la bomba. Es un objeto *protegido*, ya que únicamente responde a comandos, su funcionamiento necesita de exclusión mutua, y no llama espontáneamente a otros objetos. El objeto motor implementa todas las operaciones del controlador de la bomba. Dado que el sistema es de tiempo real, ninguna de las operaciones puede ser arbitrariamente bloqueada (aunque necesitan exclusión mutua). El objeto motor hará llamadas a todos sus objetos tíos.

Los otros dos objetos controlan los sensores de agua. El objeto sensor de flujo es un objeto *cíclico* que supervisa continuamente el flujo de agua desde la mina. El sensor de agua sup-inf es un objeto *activo* que gestiona las interrupciones de los sensores de agua superior e inferior. Se descompone en un objeto *protegido* y en uno *esporádico*, como se muestra en la Figura 17.6.

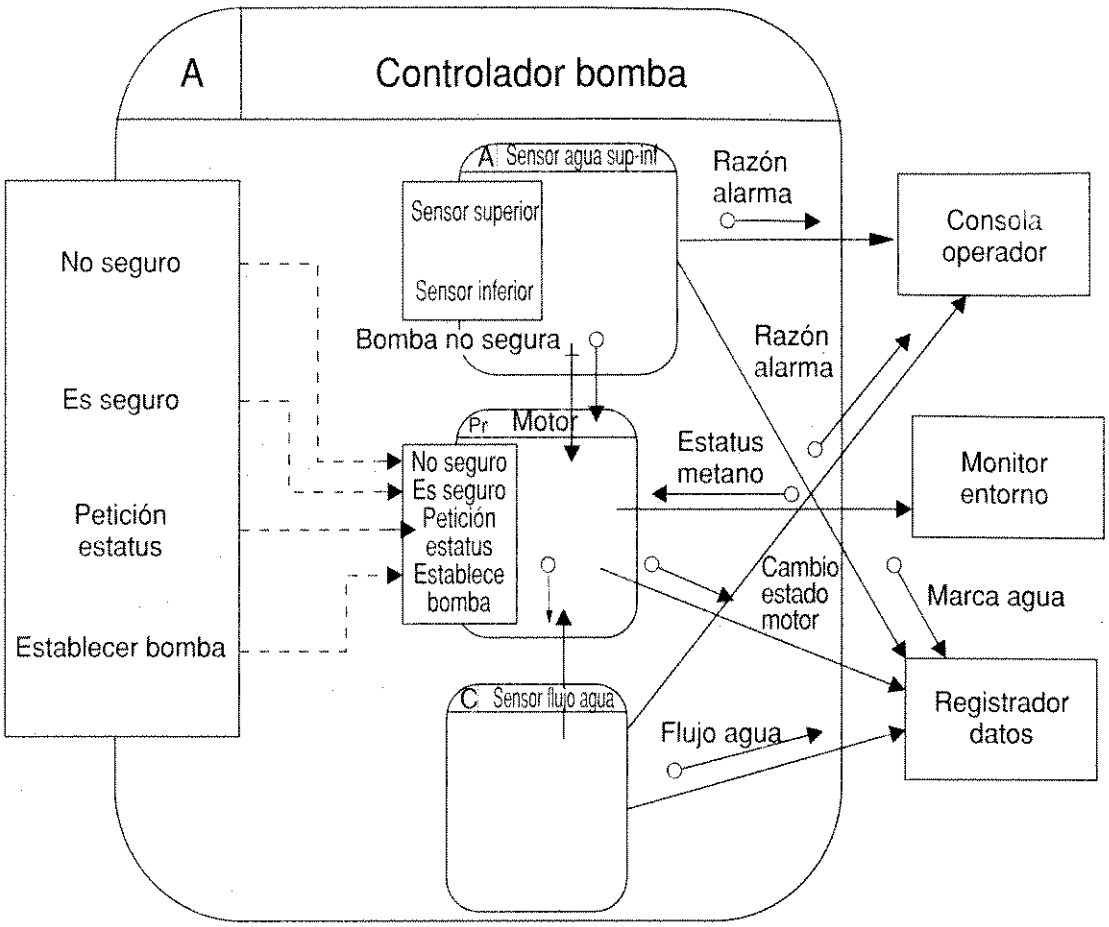


Figura 17.5. Descomposición jerárquica del objeto bomba.

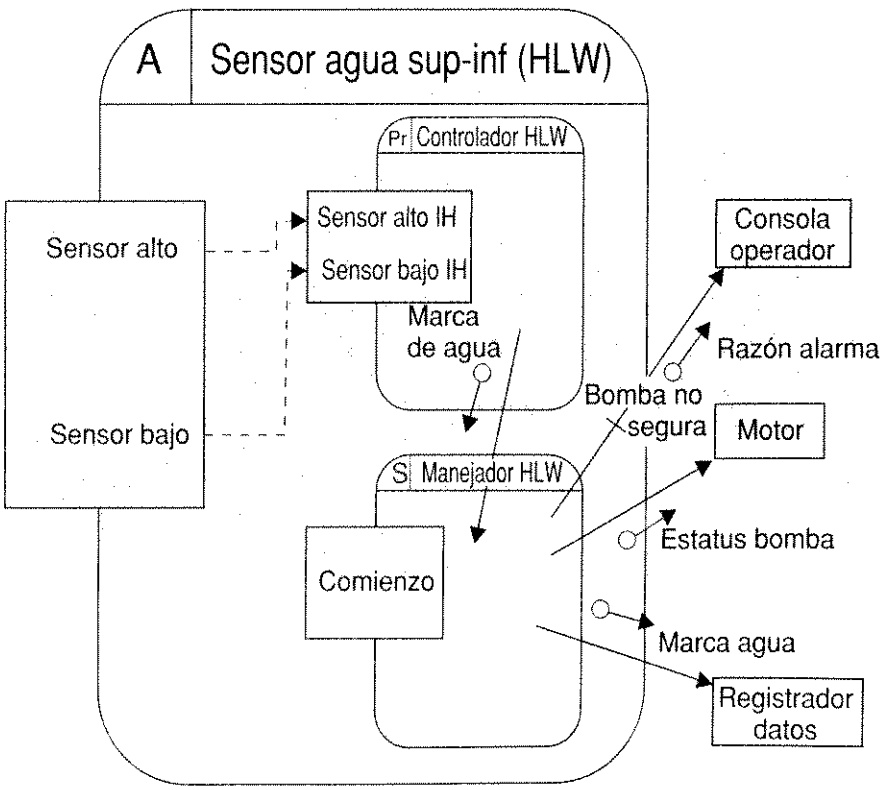


Figura 17.6. Descomposición del sensor de agua sup-inf.

17.3.3 El monitor del entorno

El monitor del entorno se descompone en cuatro objetos terminales, como se muestra en la Figura 17.7. Tres de ellos son objetos cíclicos que supervisan los niveles de CH_4 y de CO y el flujo de aire en el entorno de la mina; sólo el nivel de CH_4 puede ser pedido por otros objetos del sistema, por lo que se utiliza un objeto protegido para controlar el acceso al valor existente.

17.3.4 El registrador de datos y la consola del operador

Este caso de estudio no trata los detalles de registro de datos ni los de consola del operador. Sin embargo, existe el requisito de que sólo se puede retrasar a los hilos de tiempo real durante un tiempo limitado. Se supone, por tanto, que sus interfaces contienen objetos protegidos.

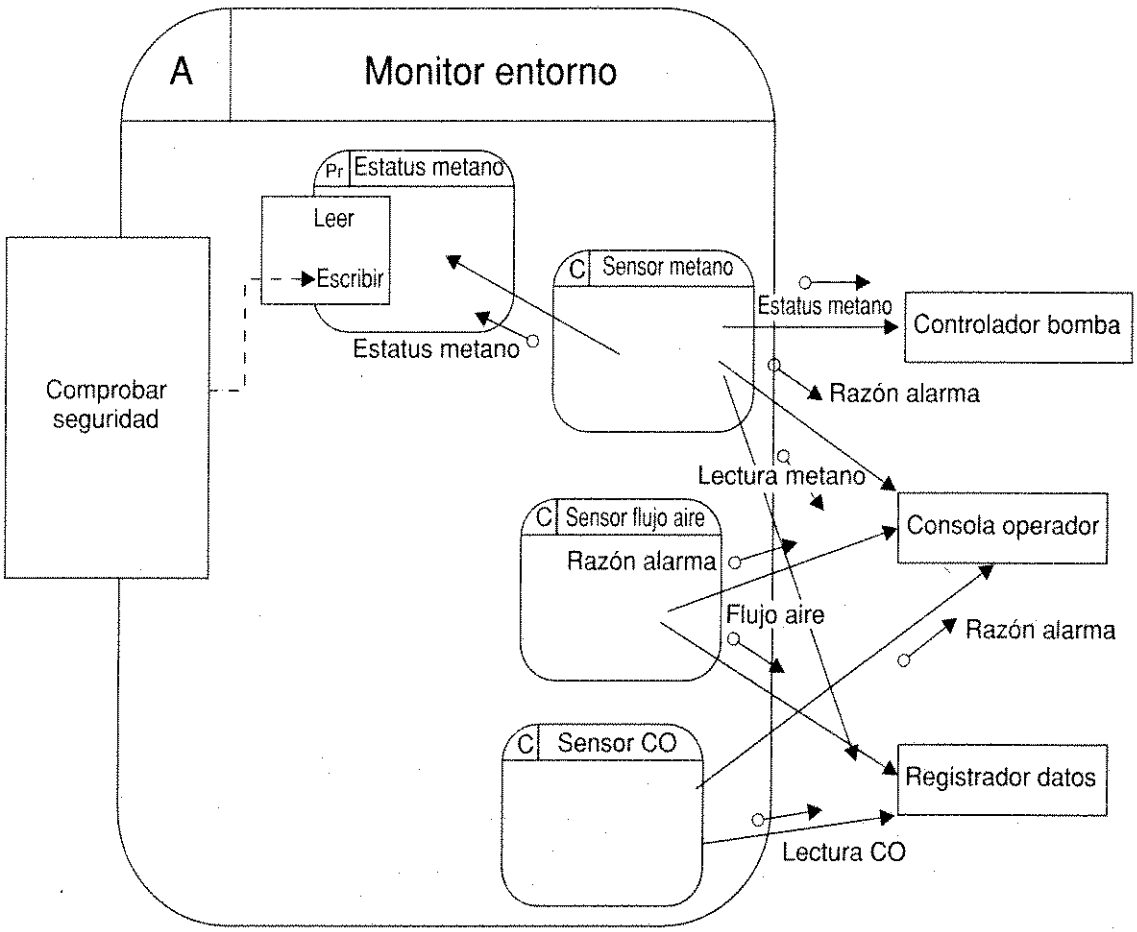


Figura 17.7. Descomposición jerárquica del monitor del entorno.

17.4 El diseño de la arquitectura física

HRT-HOOD permite el diseño de la arquitectura física:

- Permitiendo la asociación de atributos temporales a los objetos.

- Proporcionando un entorno en el que se pueda definir una propuesta de planificación y analizar los objetos terminales definidos.
- Proporcionando las abstracciones que permiten al diseñador expresar el manejo de los errores de tiempo.

Los requisitos no funcionales identificados en la Sección 17.1.2 se transforman en anotaciones sobre los métodos y los hilos. Para acometer el análisis descrito en el Capítulo 10, se utilizará una planificación de prioridad fija y una forma de análisis respuesta-tiempo. La tabla 17.2 resume los atributos temporales de los objetos que se introdujeron en la arquitectura lógica.

Tabla 17.2. Atributos de los objetos de diseño.

	Tipo	«Periodo»	Tiempo límite	Prioridad
Sensor CH_4	Periódico	80	30	10
Sensor CO	Periódico	100	60	8
Sensor flujo-aire	Periódico	100	100	7
Sensor flujo-agua	Periódico	1.000	40	9
Manejador HLW	Esporádico	6.000	200	6
Motor	Protegido			10
Controlador HLW	Protegido			11
Estatus CH_4	Protegido			10
Consola operador	Protegido			10
Registrador datos	Protegido			10

17.4.1 Análisis de planificación

Una vez que se ha desarrollado el código, éste debe ser analizado para obtener sus tiempos de ejecución *en el peor caso*. Como se indica en la Sección 15.8, estos valores pueden obtenerse por medio de una medida directa o por modelado del hardware. El código obtenido no es particularmente extenso, por lo que se asume razonablemente que un procesador de baja velocidad es adecuado. La Tabla 17.3 contiene algunos valores representativos de los tiempos de ejecución en el peor caso (en milisegundos) para los objetos del diseño. Los tiempos para cada hilo son: el tiempo empleado ejecutándose en otros objetos, el tiempo empleado ejecutando manejadores de excepciones, y el tiempo asociado a los cambios de contexto. Se introduce un «pseudoobjeto» esporádico, que representa el efecto del manejador de interrupciones (2ms es el tiempo máximo de ejecución del manejador).

El entorno de ejecución impone sus propios parámetros de importancia (que se dan en la Tabla 17.4). Obsérvese que la interrupción de reloj es de suficiente granularidad como para asegurar que no se produce inestabilidad en el lanzamiento de los hilos periódicos.

Tabla 17.3. Tiempos de ejecución en el peor caso.

	Tipo	WCET
Sensor CH_4	Periódico	12
Sensor CO	Periódico	10
Sensor flujo aire	Periódico	10
Sensor flujo agua	Periódico	10
Manejador HLW	Esporádico	20
Interrupción	Esporádico	2

Tabla 17.4. Sobrecargas.

	Símbolo	Tiempo
Periodo de reloj	T_{CLK}	20
Sobrecarga de reloj	CT^c	2
Coste de mover una única tarea	CT^s	1

El tiempo máximo de bloqueo, para todos los hilos, ocurre cuando la consola del operador realiza una llamada sobre el objeto motor. Se puede suponer que el hilo que hace la llamada es de baja prioridad. El tiempo de ejecución en el peor caso para esta operación protegida se supone que es de 3 ms.

Toda la información anterior puede ser sintetizada para proporcionar un análisis global de los tiempos de respuesta de todos los hilos del sistema. Este análisis aparece en la Tabla 17.5. La conclusión es que se cumplen todos los tiempos límite.

Tabla 17.5. Resultados del análisis.

	Tipo	T	B	C	D	R
Sensor CH_4	Periódico	80	3	12	30	25
Sensor CO	Periódico	100	3	10	60	47
Sensor Flujo aire	Periódico	100	3	10	100	57
Sensor Flujo agua	Periódico	1.000	3	10	40	35
Manejador HLW	Esporádico	6.000	3	20	200	79

17.5 Traducción a Ada

HRT-HOOD soporta una traducción sistemática a Ada. Para cada objeto terminal se generan dos paquetes: el primero simplemente contiene un conjunto de tipos de datos y variables que definen los atributos de tiempo real del objeto; el segundo contiene el código del propio objeto.

Potencialmente, cada uno de los objetos de la Figura 17.4 puede ser implementado en un procesador aparte. Sin embargo, para el objetivo de este ejemplo, se considera una implementación monoprocesador.

La descomposición del controlador de la bomba se muestra en la Figura 17.5, y el objeto sensor de agua sup-inf en la Figura 17.6. No es posible detallar el código de estos objetos.

17.5.1 El objeto controlador de la bomba

El motor

En primer lugar, se presentan los atributos de tiempo real del objeto motor. Para simplificar, sólo se muestra el atributo de prioridad máxima.

```
package Motor_AtrTR is
  Prioridad_Maxima: constant := 10;
end Motor_AtrTR;
```

La interfaz para el objeto motor es:

```
package Motor is -- PROTEGIDO
  type Estatus_Bomba is (On, Off);
  type Condicion_Bomba is (Activado, Desactivado);

  type Cambios_Estado_Motor is (Motor_Arrancado,
       Motor_Parado, Motor_Seguro, Motor_Inseguro);

  type Estatus_Funcionamiento is
    record
       Ps : Estatus_Bomba;
       Pc : Condicion_Bomba;
    end record;

  Bomba_No_Segura : exception;

  procedure No_Seguro;
  procedure Es_Seguro;

  function Peticion_Estatus return Estatus_Funcionamiento;
  procedure Establecer_Bomba(To : Estatus_Bomba);
end Motor;
```

El estado del motor se define por dos variables: una que indica si la bomba debería estar on u off, y otra que indica si la bomba está activada o desactivada. La bomba está desactivada cuando su

funcionamiento no es seguro. El tipo `Cambios_Estado_Motor` se utiliza para indicar los cambios de estado al registro de datos.

En la Figura 17.8 se muestra un diagrama de transición de estados del motor. La bomba sólo estará en funcionamiento efectivo en el estado «on-activado».

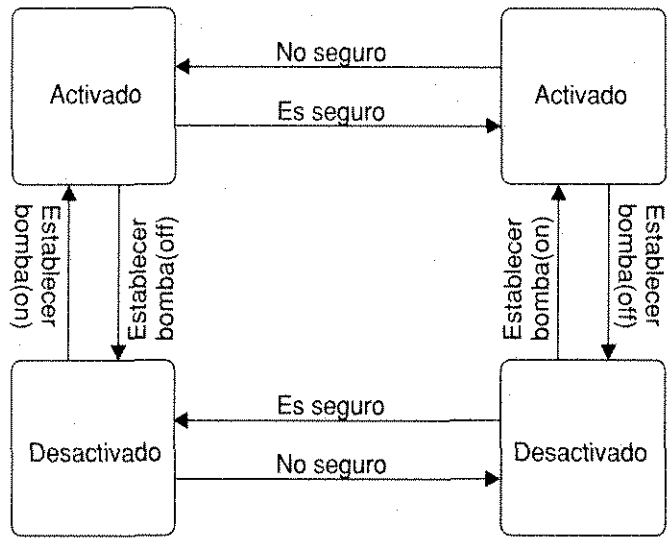


Figura 17.8. Diagrama de transición de estados del motor.

Consecuentemente, el cuerpo del paquete implementa los cambios de estado. Se utiliza un objeto protegido, ya que estos cambios deben realizarse de forma automática. En este capítulo todos los objetos generados para implementar restricciones de sincronización se denominan `Agentes`. En el código se supone que existe un paquete `Tipos_Registro_Dispositivo` que declara tipos registro de los dispositivos.

```
package Tipos_Registro_Dispositivo is
  Word : constant := 2;        -- dos bytes en una palabra
  One_Word : constant := 16;  -- 16 bits en una palabra
  -- tipos de los campos del registro
  type Error_Dispositivo is (Clear, Set);
  type Funcionamiento_Dispositivo is (Clear, Set);
  type Estatus_Interrupcion is (I_Desactivado, I_Activado);
  type Estatus_Dispositivo is (D_Desactivado, D_Activado);

  -- tipo registro
  type Rce is
    record
      Error_Bit : Error_Dispositivo;
      Funcionamiento : Funcionamiento_Dispositivo;
      Hecho : Boolean;
      Interrupcion : Estatus_Interrupcion;
```

```
      Dispositivo      : Estatus_Dispositivo;
   end record;
   -- representación bit del registro
   for Error_Dispositivo use (Clear => 0, Set => 1);
   for Funcionamiento_Dispositivo use (Clear => 0, Set => 1);
   for Estatus_Interrupcion use (I_Desactivado => 0,
                                 I_Activado => 1);
   for Estatus_Dispositivo use (D_Desactivado => 0,
                                D_Activado => 1);
   for Rce use
      record at mod Word;
         Error_Bit      at 0 range 15 ..15;
         Funcionamiento at 0 range 10 ..10;
         Hecho          at 0 range 7 ..7;
         Interrupcion   at 0 range 6 ..6;
         Dispositivo    at 0 range 0 ..0;
      end record;
   for Rce'Size use One_Word;
   for Rce'Alignment use Word;
   for Rce'Bit_order use Low_Order_First;
end Tipos_Registro_Dispositivo;
```

El cuerpo del paquete `Motor` contiene la implementación del tipo protegido `Agente`. Las operaciones externas llaman a subprogramas protegidos.

```
with Registrador_Datos;
with Estatus_Ch4; use Estatus_Ch4;
with Tipos_Registro_Dispositivo; use Tipos_Registro_Dispositivo;
with System; use System;
with Motor_AtrTR; use Motor_AtrTR;
with System.Storage_Elements; use System.Storage_Elements;
package body Motor is
   Control_Reg_Addr : constant Address := To_Address(16#Aa14#);
   PRce : Tipos_Registro_Dispositivo.Rce :=
      (Error_Bit => Clear, Funcionamiento => Set,
      Hecho => False, Interrupcion => I_Activado,
      Dispositivo => D_Activado);
   for PRce'Address use Control_Reg_Addr;

   protected Agente is
      pragma Priority(Motor_AtrTR.
                      Prioridad_Maxima);
```

```
   procedure No_Seguro;
   procedure Es_Seguro;
   function Peticion_Estatus return Estatus_Funcionamiento;
   procedure Establecer_Bomba(To : Estatus_Bomba);
private
   Estatus_Motor   : Estatus_Bomba    := Off;
   Condicion_Motor : Condicion_Bomba  := Desactivado;
end Agente;

procedure No_Seguro is
begin
   Agente.No_Seguro;
end No_Seguro;

procedure Es_Seguro is
begin
   Agente.Es_Seguro;
end Es_Seguro;

function Peticion_Estatus return Estatus_Bomba is
begin
   return Agente.Peticion_Estatus;
end Peticion_Estatus;

procedure Establecer_Bomba(To : Estatus_Bomba) is
begin
   Agente.Establecer_Bomba(To);
end Establecer_Bomba;

protected body Agente is
   procedure No_Seguro is
   begin
      if Estatus_Motor = On then
         PRce.Funcionamiento := Clear; -- parar motor
         Registrador_Datos.Motor_Log(Motor_Parado);
      end if;
      Condicion_Motor := Desactivado;
      Registrador_Datos.Motor_Log(Motor_Inseguro);
   end No_Seguro;

   procedure Es_Seguro is
```

```
      begin
         if Estatus_Motor = On then
            PRce.Funcionamiento := Set; -- arrancar motor
            Registrador_Datos.Motor_Log(Motor_Arrancado);
         end if;
         Condicion_Motor := Activado;
         Registrador_Datos.Motor_Log(Motor_Seguro);
      end Es_Seguro;

      function Peticion_Estatus return Estatus_Funcionamiento is
      begin
         return (Ps => Estatus_Motor, Pc => Condicion_Motor);
      end Peticion_Estatus;

      procedure Establecer_Bomba(To : Estatus_Bomba) is
      begin
         if To = On then
            if Estatus_Motor = Off then
               if Condicion_Motor = Desactivado then
                  raise Bomba_No_Segura;
               end if;
               if Estatus_Ch4.Leer = Motor_Seguro then
                  Estatus_Motor := On;
                  PRce.Funcionamiento := Set; -- arrancar motor
                  Registrador_Datos.Motor_Log(Motor_Arrancado);
               else
                  raise Bomba_No_Segura;
               end if;
            end if;
         else
            if Estatus_Motor = On then
               Estatus_Motor:= Off;
               if Condicion_Motor = Activado then
                  PRce.Funcionamiento := Clear; -- parar motor
                  Registrador_Datos.Motor_Log(Motor_Parado);
               end if;
            end if;
         end if;
      end Establecer_Bomba;
   end Agente;
end Motor;
```

Objeto controlador del sensor del flujo de agua

El sensor del flujo de agua es un objeto cíclico, y por tanto tiene los siguientes atributos de tiempo real.

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Sensor_Flujo_Agua_AtrTR is
  Periodo : Time_Span := Milliseconds(1000);
  Prioridad_Thread : constant Priority := 9;
end Sensor_Flujo_Agua_AtrTR;
```

No se proporciona interfaz del objeto, si bien se necesita una declaración de tipo para el registro de datos. El pragma es necesario para asegurar que se construye el cuerpo del paquete.

```
package Sensor_Flujo_Agua is -- C ´ ICLICO
  pragma Elaborate_Body;
  type Flujo_Agua is (Yes, No);
                    -- llama a Consola_Operador.Alarma
                    -- llama a Registrador_Datos.Flujo_Agua_Log
                    -- llama a Motor.Peticion_Estatus
end Sensor_Flujo_Agua;
```

El cuerpo contiene dos subprogramas: uno para inicializar el sensor `Inicializar`, y otro para el código que es ejecutado cada periodo (`Codigo_Periodico`). En cada invocación, la tarea comprueba que el agua fluye si la bomba está en funcionamiento, y que no lo hace si la bomba está apagada. Si se violan estas dos invariantes, se hacen sonar las alarmas. La tarea `Thread` implementa los atributos temporales periódicos adecuados.

```
with Ada.Real_Time; use Ada.Real_Time;
with Tipos_Registro_Dispositivo; use Tipos_Registro_Dispositivo;
with System; use System;
with Sensor_Flujo_Agua_AtrTR; use Sensor_Flujo_Agua_AtrTR;
with Motor; use Motor;
with Consola_Operador; use Consola_Operador;
with Registrador_Datos; use Registrador_Datos;
with System.Storage_Elements; use System.Storage_Elements;
package body Sensor_Flujo_Agua is
  Flujo : Flujo_Agua := No;
  Estatus_Bomba_Actual, Estatus_Bomba_Ultimo : Estatus_Bomba := Off;

  Control_Reg_Addr : constant Address := To_Address(16#Aa14#);
  WfRce : Tipos_Registro_Dispositivo.Rce;
  for WfRce'Address use Control_Reg_Addr;
```

```
   procedure Inicializar is
   begin
      -- activar dispositivo
      WfRce.Dispositivo := D_Activado;
   end Inicializar;

   procedure Codigo_Periodico is
   begin
      Estatus_Bomba_Actual := Motor.Peticion_Estatus;
      if (WfRce.Funcionamiento = Set) then
         Flujo := Yes;
      else
         Flujo := No;
      end if;
      if Estatus_Bomba_Actual = On and
            Estatus_Bomba_Ultimo = On and Flujo = No then
         Consola_Operador.Alarma(Fallo_Bomba);
      elsif Estatus_Bomba_Actual = Off and
            Estatus_Bomba_Ultimo = Off and Flujo = Yes then
         Consola_Operador.Alarma(Fallo_Bomba);
      end if;
      Estatus_Bomba_Ultimo := Estatus_Bomba_Actual;
      Registrador_Datos.Flujo_Agua_Log(Flujo);
   end Codigo_Periodico;

   task Thread is
      pragma Priority(Sensor_Flujo_Agua_AtrTR.Prioridad_Thread);
   end Thread;

   task body Thread is
      T: Time;
      Periodo : Time_Span := Sensor_Flujo_Agua_AtrTR.Periodo;
   begin
      T:= Clock;
      Inicializar;
      loop
         Codigo_Periodico;
         T := T + Periodo;
         delay until(T);
      end loop;
   end Thread;
end Sensor_Flujo_Agua;
```

El objeto controlador HLW

El objeto controlador HLW (High Low Water) es el encargado de manejar las interrupciones de los detectores de agua superior e inferior. Su objetivo es asignar las dos interrupciones a una llamada de un único objeto esporádico denominado «manejador HLW». Esto es así porque HRT-HOOD no permite que un objeto esporádico sea invocado por más de una operación de comienzo. Los manejadores de interrupciones son procedimientos dentro de un objeto protegido Ada.

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Controlador_Hlw_AtrTR is
  Prioridad_Maxima : constant Priority := 11;
end Controlador_Hlw_AtrTR;

with Controlador_Hlw_AtrTR; use Controlador_Hlw_AtrTR;
package Controlador_Hlw is -- PROTEGIDO
  procedure Sensor_Sup_Ih;
  procedure Sensor_Inf_Ih;
end Controlador_Hlw;

with Manejador_Hlw; use Manejador_Hlw;
with System; use System;
with Ada.Interrupts; use Ada.Interrupts;
with Ada.Interrupts.Names; use Ada.Interrupts.Names;
-- Ada.Interrupts.Names define Waterh_Interrupt y Waterl_Interrupt
package body Controlador_Hlw is
  protected Agente is
    pragma Priority(Controlador_Hlw_AtrTR.
                    Prioridad_Maxima);
    procedure Sensor_Sup_Ih;
    pragma Attach_Handler(Sensor_Sup_Ih, Waterh_Interrupt);
           -- asigna manejador de interrupción
    procedure Sensor_Inf_Ih;
    pragma Attach_Handler(Sensor_Inf_Ih, Waterl_Interrupt);
           -- asigna manejador de interrupción
  end Agente;

  procedure Sensor_Sup_Ih is
  begin
    Agente.Sensor_Sup_Ih;
  end Sensor_Sup_Ih;

  procedure Sensor_Inf_Ih is
```

```
  begin
    Agente.Sensor_Inf_Ih;
  end Sensor_Inf_Ih;

  protected body Agente is
    procedure Sensor_Sup_Ih is
    begin
      Manejador_Hlw.Comienzo(Superior);
    end Sensor_Sup_Ih;
    procedure Sensor_Inf_Ih is
    begin
      Manejador_Hlw.Comienzo(Inferior);
    end Sensor_Inf_Ih;
  end Agente;
end Controlador_Hlw;
```

Manejador HLW

El objeto manejador HLW controla la respuesta de la aplicación a la interrupción, solicitando a la bomba que arranque o que se pare. Contiene una tarea que espera a la interrupción por medio de una entrada del objeto protegido (`Esperar_Comienzo`). El controlador HLW señaliza la interrupción llamando a la operación `Comienzo`. (Una asignación más compleja permitiría la detección y gestión de sobrecargas en el dispositivo de interrupción.)

```
with System; use System;
package Manejador_Hlw_AtrTR is
  Prioridad_Maxima : constant Priority := 11;
  Prioridad_Thread : constant Priority := 6;
end Manejador_Hlw_AtrTR;

with Manejador_Hlw_AtrTR; use Manejador_Hlw_AtrTR;
package Manejador_Hlw is -- ESPORÁDICO
  type Marca_Agua is (Superior, Inferior);
  procedure Comienzo(Int : Marca_Agua);
end Manejador_Hlw;

with Tipos_Registro_Dispositivo; use Tipos_Registro_Dispositivo;
with System; use System;
with Motor; use Motor;
with Registrador_Datos;
with System.Storage_Elements; use System.Storage_Elements;
package body Manejador_Hlw is
  Hw_Control_Reg_Addr : constant Address := To_Address(16#Aa10#);
```

```
Lw_Control_Reg_Addr : constant Address := To_Address(16#Aa12#);
HwRce : Tipos_Registro_Dispositivo.Rce;
for HwRce'Address use Hw_Control_Reg_Addr;
LwRce : Tipos_Registro_Dispositivo.Rce;
for LwRce'Address use Lw_Control_Reg_Addr;

procedure Codigo_Esporadico(Int : Marca_Agua) is
begin
   if Int = Superior then
      Motor.Establecer_Bomba(On);
      Registrador_Datos.Agua_Sup_Inf_Log(Superior);
      LwRce.Interrupcion := I_Activado;
      HwRce.Interrupcion := I_Desactivado;
   else
      Motor.Establecer_Bomba(Off);
      Registrador_Datos.Agua_Sup_Inf_Log(Inferior);
      HwRce.Interrupcion := I_Activado;
      LwRce.Interrupcion := I_Desactivado;
   end if;
end Codigo_Esporadico;

procedure Inicializar is
begin
   HwRce.Dispositivo :=D_Activado;
   HwRce.Interrupcion := I_Activado;
   LwRce.Dispositivo :=D_Activado;
   LwRce.Interrupcion := I_Activado;
end Inicializar;

task Thread is
   pragma Priority(Manejador_Hlw_AtrTR.Prioridad_Thread);
end Thread;

protected Agente is
   pragma Priority(Manejador_Hlw_AtrTR.
                        Prioridad_Maxima);
   -- para la operación Comienzo
   procedure Comienzo(Int : Marca_Agua);
   entry Esperar_Comienzo(Int : out Marca_Agua);
private
   Comienzo_Abierto : Boolean := False;
   W : Marca_Agua;
end Agente;
```

```
  procedure Comienzo(Int : Marca_Agua) is
  begin
  Agente.Comienzo(Int);
  end Comienzo;

  protected body Agente is
     procedure Comienzo(Int : Marca_Agua) is
     begin
        W := Int;
        Comienzo_Abierto := True;
     end Comienzo;

     entry Esperar_Comienzo(Int : out Marca_Agua)
             when Comienzo_Abierto is
     begin
        Int := W;
        Comienzo_Abierto := False;
     end Esperar_Comienzo;
  end Agente;

  task body Thread is
     Int : Marca_Agua;
  begin
     Inicializar;
     loop
        Agente.Esperar_Comienzo(Int);
        Codigo_Esporadico(Int);
     end loop;
  end Thread;
end Manejador_Hlw;
```

17.5.2 Monitorización del entorno

En la Figura 17.7 se muestra la descomposición del subsistema de monitorización del entorno. A través del objeto protegido «estatus CH_4», el subprograma `comprobar_seguridad` permite que el controlador de la bomba pueda observar sin bloquear el estado actual del nivel de metano. El resto de componentes son actividades periódicas.

El objeto estatus CH_4

El objeto estatus CH_4 contiene información que indica si es o no seguro hacer que la bomba funcione.

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Estatus_Ch4_AtrTR is
  -- para objetos PROTEGIDOS
  Prioridad_Maxima : constant Priority := 10;
end Estatus_Ch4_AtrTR;

package Estatus_Ch4 is -- PROTEGIDO
  type Estatus_Metano is (Motor_Seguro, Motor_Inseguro);

  function Leer return Estatus_Metano;
  procedure Escribir (Estatus_Actual : Estatus_Metano);
end Estatus_Ch4;

with Estatus_Ch4_AtrTR; use Estatus_Ch4_AtrTR;
package body Estatus_Ch4 is
  protected Agente is
    pragma Priority(Estatus_Ch4_AtrTR.
                    Prioridad_Maxima);
    procedure Escribir (Estatus_Actual : Estatus_Metano);
    function Leer return Estatus_Metano;
  private
    Estatus_Entorno : Estatus_Metano := Motor_Inseguro;
  end Agente;

  function Leer return Estatus_Metano is
  begin
    return Agente.Leer;
  end Leer;
  procedure Escribir (Estatus_Actual : Estatus_Metano) is
  begin
    Agente.Escribir(Estatus_Actual);
  end Escribir;

  protected body Agente is
    procedure Escribir (Estatus_Actual : Estatus_Metano) is
    begin
      Estatus_Entorno := Estatus_Actual;
    end Escribir;

    function Leer return Estatus_Metano is
    begin
```

```
        return Estatus_Entorno;
      end Leer;
  end Agente;
end Estatus_Ch4;
```

El objeto controlador del sensor CH_4

La función del sensor CH_4 es medir el nivel de metano en el entorno. El requisito es que no debería sobrepasar un umbral. Inevitablemente, el sensor continuamente señalará «seguro» e «inseguro» en las cercanías del umbral. Para evitar esta inestabilidad, se utilizan unas cotas inferiores y superiores sobre el umbral. Dado que el ADC tarda un tiempo en producir el resultado, la conversión se pide al final de un periodo para ser utilizada al comienzo del siguiente.

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Sensor_Ch4_AtrTR is

   Periodo : Time_Span := Milliseconds(80);
   Prioridad_Thread : constant Priority := 10;

end Sensor_Ch4_AtrTR;

package Sensor_Ch4 is -- CYCLIC
   pragma Elaborate_Body;
   type Lectura_Ch4 is new Integer range 0 .. 1023;
   Ch4_Sup : constant Lectura_Ch4 := 400;
                        -- llama a Motor.Es_Seguro
                        -- llama a Motor.No_Seguro
                        -- llama a Consola_Operador.Alarma
                        -- llama a Registrador_Datos.Ch4_Log
end Sensor_Ch4;

with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
with Sensor_Ch4_AtrTR; use Sensor_Ch4_AtrTR;
with Tipos_Registro_Dispositivo; use Tipos_Registro_Dispositivo;
with Consola_Operador; use Consola_Operador;
with Motor; use Motor;
with Registrador_Datos; use Registrador_Datos;
with Estatus_Ch4; use Estatus_Ch4;
with System.Storage_Elements; use System.Storage_Elements;
package body Sensor_Ch4 is
   Ch4_Actual : Lectura_Ch4;
```

```
   Control_Reg_Addr : constant Address := To_Address(16#Aa18#);
   Data_Reg_Addr : constant Address := To_Address(16#Aa1a#);
   Ch4Rce : Tipos_Registro_Dispositivo.Rce;
   for Ch4Rce'Address use Control_Reg_Addr;
   -- define el registro de datos
   Ch4dbr : Lectura_Ch4;
   for Ch4dbr'Address use Data_Reg_Addr;
   Rango_Inestab : constant Lectura_Ch4 := 40;

   procedure Inicializar is
   begin
      -- activar dispositivo
      Ch4Rce.Dispositivo := D_Activado;
      Ch4Rce.Funcionamiento := Set;
   end Inicializar;

   procedure Codigo_Periodico is
   begin
      if not Ch4Rce.Hecho then
         Consola_Operador.Alarma(Ch4_Error_Dispositivo);
      else
         -- leer el valor del sensor del registro del dispositivo
         Ch4_Actual := Ch4dbr;
         if Ch4_Actual > Ch4_Sup then
            if Estatus_Ch4.Leer = Motor_Seguro then
               Motor.No_Seguro;
               Estatus_Ch4.Escribir(Motor_Inseguro);
               Consola_Operador.Alarma(Metano_Alto);
            end if;
         elsif (Ch4_Actual < (Ch4_Sup - Rango_Inestab)) and
               (Estatus_Ch4.Leer = Motor_Inseguro) then
            Motor.Es_Seguro;
            Estatus_Ch4.Escribir(Motor_Seguro);
         end if;
         Registrador_Datos.Ch4_Log(Ch4_Actual);
      end if;
      Ch4Rce.Funcionamiento := Set; -- comenzar conversión para iteración siguiente
   end Codigo_Periodico;

task Thread is
   pragma Priority(Sensor_Ch4_AtrTR.
                   Prioridad_Thread);
```

```
  end Thread;

  task body Thread is
    T: Time;
    Periodo : Time_Span := Sensor_Ch4_AtrTR.Periodo;
  begin
    T:= Clock + Periodo;
    Inicializar;
    loop
      delay until(T);
      Codigo_Periodico;
      T := T + Periodo;
    end loop;
  end Thread;
end Sensor_Ch4;
```

17.5.3 El objeto controlador del sensor del flujo de aire

El sensor del flujo de aire es otro objeto periódico que controla el flujo de aire en la mina.

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Sensor_Flujo_Aire_AtrTR is
  Periodo : Time_Span := Milliseconds(100);
  Prioridad_Thread : constant Priority := 7;
end Sensor_Flujo_Aire_AtrTR;

package Sensor_Flujo_Aire is -- C ´ ICLICO
  pragma Elaborate_Body;
  type Estatus_Flujo_Aire is (Flujo_Aire, No_Flujo_Aire);
                        -- llama a Registrador_Datos.Flujo_Aire_Log
                        -- llama a Consola_Operador.Alarma
end Sensor_Flujo_Aire;

with Tipos_Registro_Dispositivo; use Tipos_Registro_Dispositivo;
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
with Sensor_Flujo_Aire_AtrTR; use Sensor_Flujo_Aire_AtrTR;
with Consola_Operador; use Consola_Operador;
with Registrador_Datos; use Registrador_Datos;
with System.Storage_Elements; use System.Storage_Elements;
```

```
package body Sensor_Flujo_Aire is
   Lectura_Flujo_Aire : Boolean;

   Control_Reg_Addr : constant Address := To_Address(16#Aa120#);
   AfRce : Tipos_Registro_Dispositivo.Rce;
   for AfRce'Address use Control_Reg_Addr;

   task Thread is
      pragma Priority(Sensor_Flujo_Aire_AtrTR.
                      Prioridad_Thread_Inicial);
   end Thread;

   procedure Inicializar is
   begin
      -- activar dispositivo
      AfRce.Dispositivo := D_Activado;
   end Inicializar;

   procedure Codigo_Periodico is
   begin
      -- leer la indicación de flujo del registro del dispositivo
      -- (bit de funcionamiento a 1);
      Lectura_Flujo_Aire := AfRce.Funcionamiento = Set;
      if not Lectura_Flujo_Aire then
         Consola_Operador.Alarma(No_Flujo_Aire);
         Registrador_Datos.Flujo_Aire_Log(No_Flujo_Aire);
      else
         Registrador_Datos.Flujo_Aire_Log(Flujo_Aire);
      end if;
   end Codigo_Periodico;

   task body Thread is
      T: Time;
      Periodo : Time_Span := Sensor_Flujo_Aire_AtrTR.Periodo;
   begin
      T:= Clock;
      Inicializar;
      loop
         delay until(T);
         Codigo_Periodico;
         T := T + Periodo;
      end loop;
   end Thread;
end Sensor_Flujo_Aire;
```

17.5.4 El objeto controlador del sensor CO

La implementación del sensor CO es directa.

```
with System; use System;
with Ada.Real_Time; use Ada.Real_Time;
package Sensor_Co_AtrTR is

   Periodo : Time_Span := Milliseconds(100);
   Prioridad_Thread : constant Priority := 8;
end Sensor_Co_AtrTR;

   package Sensor_Co is -- C ´ ICLICO
   pragma Elaborate_Body;
   type Lectura_Co is new Integer range 0 .. 1023;
   Co_Sup : constant Lectura_Co := 600;
                        -- llama a Registrador_Datos.Co_log
                        -- llama a Consola_Operador.Alarma
end Sensor_Co;

with Ada.Real_Time; use Ada.Real_Time;
with System; use System;
with Tipos_Registro_Dispositivo; use Tipos_Registro_Dispositivo;
with Sensor_Co_AtrTR; use Sensor_Co_AtrTR;
with Consola_Operador; use Consola_Operador;
with Registrador_Datos; use Registrador_Datos;
with System.Storage_Elements; use System.Storage_Elements;
package body Sensor_Co is
   Co_Actual : Lectura_Co;

   Control_Reg_Addr : constant Address := To_Address(16#Aa1c#);
   Data_Reg_Addr : constant Address := To_Address(16#Aa1e#);
   CoRce : Tipos_Registro_Dispositivo.Rce;
   for CoRce'Addres use Control_Reg_Addr;
   -- define el registro de datos
   Codbr : Lectura_Co;
   for Codbr'Address use Data_Reg_Addr;

   procedure Inicializar is
   begin
      -- activar dispositivo
      CoRce.Dispositivo := D_Activado;
      CoRce.Funcionamiento := Set; -- comenzar la conversión
end Inicializar;
```

```
   procedure Codigo_Periodico is
   begin
      if not CoRce.Hecho then
         Consola_Operador.Alarma(Co_Error_Dispositivo);
      else
         -- leer el valor del sensor en el registro del dispositivo
         Co_Actual := Codbr;
         if Co_Actual > Co_Sup then
            Consola_Operador.Alarma(Co_Alto);
         end if;
         Registrador_Datos.Co_Log(Co_Actual);
      end if;
      CoRce.Funcionamiento := Set; -- comenzar la conversión
   end Codigo_Periodico;

   task Thread is
      pragma Priority(Sensor_Co_AtrTR.Prioridad_Thread);
   end Thread;

   task body Thread is
      T : Time;
      Periodo : Time_Span := Sensor_Co_AtrTR.Periodo;
   begin
      T := Clock + Periodo;
      Inicializar;
      loop
         delay until(T);
         Codigo_Periodico;
         T := T + Periodo;
      end loop;
   end Thread;
end Sensor_Co;
```

17.5.5 Registrador de datos

Sólo se muestra la interfaz del objeto «registrador de datos».

```
with Sensor_Co; use Sensor_Co;
with Sensor_Ch4; use Sensor_Ch4;
with Sensor_Flujo_Aire; use Sensor_Flujo_Aire;
with Manejador_Hlw; use Manejador_Hlw;
with Sensor_Flujo_Agua; use Sensor_Flujo_Agua;
```

```
with Motor; use Motor;
package Registrador_Datos is -- ACTIVO
  procedure Co_Log(Lectura : Lectura_Co);
  procedure Ch4_Log(Lectura : Lectura_Ch4);
  procedure Flujo_Aire_Log(Lectura : Estatus_Flujo_Aire);
  procedure Agua_Sup_Inf_Log(Marca : Marca_Agua);
  procedure Flujo_Agua_Log(Lectura : Flujo_Agua);
  procedure Motor_Log(Estado : Cambios_Estado_Motor);
end Registrador_Datos;
```

17.5.6 Consola del operador

Sólo se muestra la interfaz del objeto «consola del operador».

```
package Consola_Operador is -- ACTIVO
  type Razon_Alarma is (Metano_Alto, Co_Alto, No_Flujo_Aire,
                Ch4_Error_Dispositivo, Co_Error_Dispositivo,
                 Fallo_Bomba, Error_Desconocido);
  procedure Alarma(Razon: Razon_Alarma;
                 Name : String := "Desconocido";
                 Details : String:= "");

  -- llama a Peticion_Estatus en Controlador_Bomba
  -- llama a Establecer_Bomba en Controlador_Bomba
end Consola_Operador;
```

17.6 Tolerancia a fallos y distribución

En el Capítulo 5 se identificaron cuatro fuentes de defectos que pueden hacer fallar un sistema software embebido.

(1) Una especificación inadecuada.

(2) Defectos causados por errores de diseño de los componentes software.

(3) Defectos causados por fallos en uno o más componentes procesadores del sistema embebido.

(4) Defectos causados por interferencias permanentes o transitorias en el subsistema de comunicaciones.

Este libro se ha centrado en las tres últimas fuentes, que ahora se discuten en relación con el caso de estudio. La aproximación de Ada a la tolerancia a fallos en el software es utilizar el manejo de excepciones como un entorno sobre el que se puede construir la recuperación de errores.

17.6.1 Errores de diseño

El alcance de la tolerancia a fallos por errores de diseño en este caso es limitado, ya que el ejemplo ha sido necesariamente simplificado. La metodología de diseño HRT-HOOD, junto con las utilidades de abstracción de datos de Ada, contribuyen a prevenir la incorporación de defectos en el sistema durante las fases de diseño e implementación. En una aplicación real, esto se acompañaría de una fase exhaustiva de prueba con el objeto de eliminar los defectos existentes. También se pueden utilizar simulaciones.

Cualquier defecto residual de diseño provocará la aparición de errores imprevistos. Aunque para la recuperación de este tipo de errores es ideal disponer de recuperación de errores hacia atrás o de programación con *N*-versiones, no hay mucho campo de acción en el ejemplo para variar el diseño. Si en el ejemplo se supone que todos los errores imprevistos generan una excepción, cada una de las tareas u operaciones podría ser protegida con un manejador de excepciones «when other». Por ejemplo, podría modificarse el hilo de flujo de agua para informar al operador de los errores imprevistos.

```
   task Thread;

   task body Thread is
      T: Time;
      Periodo : Time_Span := Sensor_Flujo_Agua_AtrTR.Periodo;
      use Ada.Exceptions;
   begin
      T:= Clock;
      Inicializar;
      loop
         Codigo_Periodico;
         T := T + Periodo;
         delay until(T);
      end loop;
   exception
      when E: others =>
         Consola_Operador.Alarma(Error_Desconocido, Exception_Name(E),
                                 Exception_Information(E) );
         Motor.No_Seguro; -- Controlador_Bomba.No_Seguro
         Estatus_Ch4.Escribir(Motor_Inseguro);
         -- intenta y apaga el motor antes de terminar
   end Thread;
end Sensor_Flujo_Agua;
```

A pesar de la gravedad de una inundación de la mina, los requisitos indican que el fuego es más peligroso; por tanto, el manejo de errores siempre tiende a asegurar que la bomba se encuentra apagada (seguridad de fallo).

Debe advertirse que todas las interacciones que manipulan la bomba y el estatus del metano deben tener la forma de acciones atómicas. El cógigo anterior permite a otra tarea determinar que la situación del motor es insegura, a pesar de que el estatus del metano indique que el funcionamiento de la bomba es seguro (véase el Ejercicio 17.2).

17.6.2 Fallo de procesador y comunicaciones

En general, si el sistema de control de una mina se implementa en un único procesador y falla cualquier parte de éste, entonces todo el sistema estaría en peligro. Por tanto, o se aplica alguna forma de redundancia de hardware, o se introduce distribución. Los sistemas de control como el de la mina son, por naturaleza, distribuidos. La descomposición de nivel superior de la Figura 17.4 muestra cuatro componentes que, claramente, podrían ser ejecutados en procesadores distintos (véase el Ejercicio 17.4).

En el Capítulo 14 se comentó que Ada no define una semántica de fallos para programas que fallan parcialmente. Sin embargo, la implementación subyacente generará la excepción `Error_Comunicacion` cuando no pueda ponerse en contacto con una partición remota.

Se ha supuesto que todos los fallos transitorios en las comunicaciones serán enmascarados por la implementación subyacente del sistema distribuido. Si ocurre un fallo permanente, la implementación debe generar una excepción apropiada, que será manejada por la aplicación. Por ejemplo, si la llamada remota a `Es_Seguro` genera una excepción, la bomba debería desactivarse.

17.6.3 Otros fallos hardware

En lo anterior se supone que sólo el procesador y los subsistemas de comunicación pueden fallar. Obviamente, es igualmente probable que fallen los sensores, bien por deterioro o bien por avería. En el ejemplo presentado en este capítulo no se ha intentado incrementar la fiabilidad de los sensores, ya que las técnicas de redundacia de hardware sólo han sido tratadas de pasada en este libro. Una posibilidad sería replicar cada sensor, siendo controlada cada una de las réplicas por una tarea diferente. Las tareas tendrían que estar en comunicación para comparar resultados. Dado que estos resultados, inevitablemente, serían algo diferentes, se necesitaría algún algoritmo de contraste.

Resumen

Este caso de estudio ha sido incluido para ilustrar algunos de los puntos tratados en el libro. Sin embargo, una única aplicación relativamente pequeña no puede servir para poner en práctica todos los conceptos importantes tratados. Está claro que ciertos temas de envergadura y complejidad no se pueden tratar en este contexto.

Aun así, es de esperar que el caso de estudio haya ayudado a consolidar la comprensión del lector sobre una serie de temas, como por ejemplo:

- Diseño y descomposición *de arriba a abajo*.
- Concurrencia y modelo Ada de comunicación entre procesos.
- Técnicas de recuperación de errores hacia adelante y diseño tolerante a fallos.
- Procesos periódicos y esporádicos.
- Asignación de prioridades y análisis de planificación.
- Programación distribuida.

Lecturas complementarias

El caso de estudio presentado en este capítulo es tratado también por otros autores en las siguientes referencias:

Burns, A., y Lister, A. M. (1991), «A Framework for Building Dependable Systems», *Computer Journal*, 34(2), 173-181.

Joseph, M. (ed.) (1996), *Real-Time Systems: Specification, Verification and Analysis*, Englewood Cliffs, NJ: Prentice Hall.

Shrivastava, S. K., Mancini, L., y Randell, B. (1987), *On The Duality of Fault Tolerant Structures*, Berlin: Springer-Verlag, pp. 19-37.

Sloman, M., y Kramer, J. (1987), *Distributed Systems and Computer Networks*, Englewood Cliffs, NJ: Prentice Hall.

Ejercicios

17.1 Si el agua penetra en la mina aproximadamente a la misma velocidad a la que la bomba la desaloja, la interrupción de agua en el nivel superior puede generarse muchas veces. ¿Afectará esto al comportamiento del software?

17.2 Señale qué interacciones de tareas del diseño del sistema de control de la mina deberían ser acciones atómicas. ¿Cómo podría modificarse la solución para soportar estas acciones?

17.3 Todas las tareas periódicas del capítulo tienen una estructura similar. ¿Pueden ser representadas como instanciaciones de una tarea genérica (encapsulada en un paquete) o de una tarea parametrizada?

17.4 Modifique la solución dada en este capítulo de forma que pueda ser ejecutada en un sistema distribuido. Suponga que todos los objetos de alto nivel de la Figura 17.4 son particiones activas.

17.5 ¿Hasta qué punto se pueden analizar las propiedades temporales de la solución al Ejercicio 17.4?

17.6 ¿Puede el registrador de datos determinar el orden real en el que han ocurrido los eventos? Si no es posible, ¿cómo podría modificarse el código para dar una ordenación global válida? ¿Cuáles serían las implicaciones para una implementación distribuida?

17.7 En el análisis del sistema de control de la mina, ¿cuáles serían las consecuencias de hacer funcionar el reloj a 100 ms? ¿Y a 10 ms?

17.8 Realice un análisis de perceptibidad del conjunto de tareas del control de la mina. Para cada una de las tareas, considere en cuánto puede incrementarse su tiempo de cómputo antes de que el conjunto de tareas se convierta en no planificable. Exprese este valor como un porcentaje del valor original de C.

17.9 Si los tiempos límite de los sensores de CO y de flujo de aire fueran los dos de 50, el sistema sería no planificable. ¿Cómo podría explotarse el hecho de que ambos sensores tengan el mismo periodo para obtener un sistema planificable?

Capítulo 18

Conclusiones

La característica diferenciadora de los sistemas de tiempo real es que la corrección no está únicamente en función de los resultados lógicos de la ejecución del programa, sino también del tiempo en el que se producen estos resultados. Esta característica hace que el estudio de los sistemas de tiempo real esté completamente separado de otras áreas de la computación. La importancia de muchos sistemas de tiempo real plantea también responsabilidades específicas en el realizador. A medida que se incrustan más computadores en aplicaciones de ingeniería, mayor es el riesgo de catástrofe humana, ecológica o económica. Estos riesgos surgen del problema de la incapacidad de probar (o al menos demostrar convincentemente) que todas las restricciones funcionales se satisfacen en todas las situaciones.

Los sistemas de tiempo real pueden ser clasificados atendiendo a diferentes criterios. El primero es el grado el con que la aplicación puede tolerar retrasos en las respuestas del sistema. Los que tienen alguna flexibilidad son denominados sistemas de tiempo real *no estrictos*; los de rigidez temporal se llaman *estrictos*. Un tiempo límite de tres horas puede ser estricto, pero fácilmente obtenible; uno de tres microsegundos (estricto o no) plantea una considerable dificultad al desarrollador. En los casos en los que los tiempos límites, o tiempos de respuesta, son muy cortos, se dice que el sistema es *auténtico* tiempo-real (*real* real-time).

Un sistema que no es de tiempo real puede esperar casi indefinidamente a procesadores u otros recursos. Mientras dicho sistema posea *viveza*, se ejecutará de forma apropiada. Éste no es el caso de los sistemas de tiempo real. Como el tiempo es limitado y los procesadores no son infinitamente rápidos, se debe considerar un programa de tiempo real que se ejecuta en un sistema con recursos limitados. Es necesario, por tanto, planificar la utilización de estos recursos ante peticiones que compiten desde diferentes partes del programa; un empeño lejos de ser sencillo.

Otras características de un sistema de tiempo real moderno típico son las siguientes:

- A veces están distribuidos geográficamente.
- Pueden contener componentes software muy grandes y complejos.

- Deben interactuar con entidades concurrentes del mundo real.
- Pueden contener elementos de proceso que están sujetos a restricciones de coste, tamaño y peso.

De la naturaleza de la mayoría de los sistemas de tiempo real se deduce que hay un requisito estricto de alta fiabilidad. Esto se puede formular también como un requisito de fiabilidad y seguridad. A veces existe un relación casi simbiótica entre el sistema del computador y su entorno inmediato. Uno no puede funcionar sin el otro, como ocurre en un avión filoguiado. Proporcionar niveles altos de fiabilidad requiere un hardware y un software tolerantes a fallos. Existe una necesidad de tolerancia ante la pérdida de funcionalidad o ante tiempos límites incumplidos (incluso para sistemas de tiempo real estricto).

La combinación de requisitos temporales, recursos limitados, entidades de entorno concurrente y requisitos de alta fiabilidad (junto con proceso distribuido) presenta nuevos problemas al ingeniero de sistemas. La ingeniería de sistemas de tiempo real se reconoce ahora como una disciplina distinta. Tiene su propio cuerpo de conocimiento y fundamento teórico. A partir de una comprensión de la ciencia de los sistemas de tiempo real grandes, surgirá lo siguiente:

- Técnicas de especificación que pueden capturar requisitos temporales y de tolerancia a fallos.
- Métodos de diseño que tengan en sus fundamentos requisitos temporales y que puedan tratar con formas de proporcionar tolerancia a fallos y distribución.
- Lenguajes de programación y sistemas operativos que puedan ser utilizados para implementar esos diseños.

Este libro se ha ocupado de las características y requisitos de los sistemas de tiempo real, técnicas de tolerancia a fallos, modelos de concurrencia, características del lenguaje relacionadas con el tiempo, control de recursos, distribución y técnicas de programación de bajo nivel. Todo el tiempo se ha proporcionado atención a las funcionalidades proporcionadas por los lenguajes de tiempo real actuales, en particular Ada, Java (y sus extensiones para tiempo real), C (aumentado con las extensiones POSIX para tiempo real e hilos), y occam2. La Tabla 18.1 resume las funcionalidades proporcionadas por estos lenguajes.

Ada es todavía el lenguaje más apropiado para sistemas de alta integridad y para los que tienen restricciones de tiempo real estrictas. Java para tiempo real rivaliza ahora con Ada en cuanto a poder expresivo. Sin embargo, todavía está por ver si las implementaciones se pueden hacer lo suficientemente eficientes para su utilización efectiva. Efectivamente, este puede ser el caso de Java para tiempo real: el lenguaje es apropiado, pero la de máquina virtual Java para tiempo real no lo es. Consecuentemente, los programas necesitarán ser compilados más que interpretados, o se necesitará una máquina virtual gestionada por hardware. Quizás donde Java para tiempo real gana terreno a Ada es su soporte para sistemas de tiempo real no estrictos más dinámicos. Es en este dominio de aplicación en el que el lenguaje encontrará su uso inicial, particularmente en aquellas aplicaciones en las que la portabilidad es esencial.

El lenguaje C, aumentado con un sistema operativo de tiempo real (con o sin conformidad POSIX) continuará utilizándose en aquellos sistemas pequeños embebidos cuyos recursos estén restringidos. El uso de occam2, en comparación, continuará reduciéndose. Académicamente tiene algunas propiedades interesantes, pero su falta de soporte para la programación a gran escala le condenará a ser poco más que un lenguaje esotérico.

Una tendencia de los últimos cinco años, que continuará previsiblemente en el futuro, es el uso de más de un lenguaje para una única aplicación. Esto es un resultado del requisito de utilizar código de legado, y el reconocimiento de que no es apropiado utilizar el mismo lenguaje para todos los casos de aplicaciones. Por ejemplo, una aplicación de tiempo real con un importante componente de interfaz de usuario significativa podría ser escrita en una mezcla de Ada y Java. Un sistema de tiempo real «inteligente» podría requerir una componente basada en reglas, para la que el lenguaje más apropiado podría ser Prolog.

Para dar un ejemplo actual de un sistema de tiempo real, se ha descrito un caso de estudio. Por necesidad, se ha tratado de un sistema relativamente pequeño. Muchos de los retos a los que se enfrenta la computación de tiempo real se manifiestan, sin embargo, sólo en las grandes aplicaciones complejas. Con el fin de dar una idea de la clase de sistemas que se utilizarán en el futuro próximo, se considera la Estación Espacial Internacional. La función principal de sus sistemas de computador es el soporte de la misión y la vida. Otras actividades son: control de vuelo (particularmente del vehículo de transferencia orbital), monitorización externa, control y monitorización de los experimentos, y gestión de la base de datos de la misión. Un aspecto particular del software en placa es la interfaz que presenta al personal de vuelo.

El entorno de ejecución en placa para la estación espacial tiene las siguientes características pertinentes:

- Es grande y complejo (es decir, hay una gran variedad de actividades que han de ser computerizadas).
- Debe tener una ejecución ininterrumpida.
- Tendrá una vida operativa larga (quizás hasta 30 años).
- Experimentará cambios evolutivos de software (sin detención).
- Debe tener una ejecución altamente fiable.
- Tendrá componentes con tiempos límite de tiempo real estricto y no estricto.
- El sistema distribuido puede contener procesadores heterogéneos.

Para enfrentarse a los retos de esta clase de aplicaciones, se debe desarrollar la ciencia de los sistemas de tiempo real por sí misma. Existen todavía muchos temas de investigación por explorar. Incluso el estado actual de los conocimientos, que ha sido el foco de atención en este libro, casi nunca se pone en práctica. En una búsqueda de la «siguiente generación» de sistemas de tiempo real, Stankovic identificó varios temas de investigación (Stankovic, 1988). Aunque se han reali-

Tabla 18.1. Resumen de las funcionalidades proporcionadas por Ada, Java para tiempo real, C/POSIX y occam2.

	Ada	**Java para tiempo real**	**C/POSIX**	**occam2**
Soporte para programación de lo grande	Paquetes genéricos	Paquetes de interfaz	Ficheros	Ninguno
Soporte para programación concurrente	Tareas, rendezvous; tipos protegidos	Hilos, métodos; sentencias sincronizadas	Procesos; hilos; mutexes	Procesos; rendezvous
Soporte para ejecución en un entorno distribuido	RPC, particiones; objetos distribuidos	Métodos remotos; objetos distribuidos	Por definir	Procesos distribuidos
Funcionalidades para programación tolerante a fallos	Excepciones; ATC	Excepciones, ATC; Eventos	Señales	Ninguna
Funcionalidades de tiempo real	Relojes y retardo	Relojes y temporizadores	Relojes y temporizadores	Relojes y retardo
Funcionalidades de planificación	Modelo de prioridad coherente; prioridades dinámicas	Modelo de prioridad coherente; prioridades dinámicas	Modelo de prioridad coherente; prioridades dinámicas	Modelo de prioridad coherente; prioridades estáticas
Modelo de manejo de dispositivos	Memoria compartida	Memoria compartida limitada	Ninguno definido	Paso de mensajes
Entorno de ejecución	Tareas restringidas; ATAC	Sin subconjuntos	Perfiles	Transputer

zado muchos progresos desde 1988, los siguientes temas de investigación son todavía cruciales para el desarrollo de la disciplina.

- Técnicas de especificación y verificación que pueden tratar las necesidades de tiempo real con un gran número de componentes interaccionando.
- Metodologías de diseño que consideren propiedades temporales desde el comienzo del proceso de diseño.
- Lenguajes de programación con construcciones explícitas para expresar comportamientos relacionados con el tiempo (en particular, relacionados con la computación distribuida).
- Algoritmos de planificación que puedan manejar estructuras de procesos y restricciones de recursos complejas, requisitos temporales de granularidad variable y garantías probabilísticas.

- Modelos de colas para tiempo real.
- Soporte de ejecución, o funciones del sistema operativo, diseñadas para tratar con el uso de recursos tolerantes a fallos.
- Soporte de herramientas para predecir los tiempos de ejecución de los casos peor y promedio para el software en procesadores modernos complejos.
- Integración de métricas de prestaciones de los casos peor y promedio.
- Arquitecturas y protocolos de comunicación para manejar eficientemente mensajes que requieren una entrega oportuna a lo largo de Internet.
- Soporte de arquitectura para tolerancia a fallos y reconfiguración dinámica.
- Soporte integrado para componentes de inteligencia artificial (por ejemplo, *machine learning*).
- Soporte de lenguaje de programación y sistema operativo para acciones atómicas, bloques de recuperación, protocolos de conversación o comunicación de grupos.
- Lenguajes de programación con soporte explícito para la gestión del cambio (por ejemplo, capacidad para hacer actualizaciones en sistemas sin interrupción).
- Máquinas virtuales de tiempo real que soporten aplicaciones de tiempo real «escritas una vez y ejecutadas en cualquier parte» con restricciones temporales estrictas.
- Arquitecturas de tiempo real reflexivas (que se automodifiquen) que permitan adaptar su comportamiento en respuesta a entornos que cambian.

Esperamos que algunos lectores de este libro sean capaces de contribuir a estos temas de investigación.

Apéndice A

Especificación de Java para tiempo real

Este apéndice contiene la especificación de las clases de Java para tiempo real que se discuten en este libro. Estas especificaciones respetan las presentadas en Bollella et al. (2000). Sin embargo, hay que tener en cuenta que Java para tiempo real es una especificación en evolución que, en el momento de escribir este libro, no ha sido corroborada por una implementación. Por eso, el lector deberá tomar como referencia el sitio web que mantiene el Grupo de Expertos en Tiempo Real para Java (The Real-Time for Java Expert Group), `http://www.rtj.org/`, para el estado actual del proyecto.

La especificación viene dada en forma de encabezados de las clases en orden alfabético. Tenga en cuenta que también se da la especificación de la clases `Thread` y `ThreadGroup`, que no forman parte de la especificación de Java para tiempo real.

A.1 AbsoluteTime

```
package javax.realtime;
public class AbsoluteTime extends HighResolutionTime
{
  // constructores
  public AbsoluteTime();
  public AbsoluteTime(AbsoluteTime time);
  public AbsoluteTime(java.util.Date date);
  public AbsoluteTime(long millis, int nanos);

  // métodos
  public AbsoluteTime absolute(Clock clock, AbsoluteTime destination);
  public AbsoluteTime add(long millis, int nanos);
```

```
  public AbsoluteTime add(long millis, int nanos,
                          AbsoluteTime destination);
  public final AbsoluteTime add(RelativeTime time);
  public AbsoluteTime add(RelativeTime time,
                          AbsoluteTime destination);
  public java.util.Date getDate();
  public void set(java.util.Date date);
  public final RelativeTime subtract(AbsoluteTime time);
  public RelativeTime subtract(AbsoluteTime time,
                               RelativeTime destination);
  public final AbsoluteTime subtract(RelativeTime time);
  public AbsoluteTime subtract(RelativeTime time,
                               AbsoluteTime destination);
  public java.lang.String toString();
}
```

A.2 AperiodicParameters

```
package javax.realtime;
public class AperiodicParameters extends ReleaseParameters
{
  // constructores
  public AperiodicParameters(RelativeTime cost, RelativeTime deadline,
        AsyncEventHandler overrunHandler,
        AsyncEventHandler missHandler);
}
```

A.3 AsyncEvent

```
package javax.realtime;
public class AsyncEvent
{
  // constructores
  public AsyncEvent();

  // métodos
  public synchronized void addHandler(AsyncEventHandler handler);
  public void bindTo(java.lang.String happening);
```

```
  public ReleaseParameters createReleaseParameters();
  public synchronized void fire();
  public boolean handledBy(AsyncEventHandler target);
  public synchronized void removeHandler(AsyncEventHandler handler);
  public void setHandler(AsyncEventHandler handler);
}
```

A.4 AsyncEventHandler

```
package javax.realtime;
public abstract class AsyncEventHandler implements Schedulable
{
  //constructores
  public AsyncEventHandler();
  public AsyncEventHandler(boolean nonheap);
  public AsyncEventHandler(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group);
  public AsyncEventHandler(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group,
         boolean nonheap);

  //métodos
  public void addToFeasibility();
  protected final synchronized int getAndClearPendingFireCount();
  protected synchronized int getAndDecrementPendingFireCount();
  protected synchronized int getAndIncrementPendingFireCount();
  public MemoryArea getMemoryArea();
  public MemoryParameters getMemoryParameters();
  public ProcessingGroupParameters getProcessingGroupParameters();
  public ReleaseParameters getReleaseParameters();
  public Scheduler getScheduler();
  public SchedulingParameters getSchedulingParameters();
  public abstract void handleAsyncEvent();
  public void removeFromFeasibility();
  public final void run();
  public void setMemoryParameters(MemoryParameters memory);
  public void setProcessingGroupParameters(
              ProcessingGroupParameters parameters);
```

```
  public void setReleaseParameters(ReleaseParameters parameters);
  public void setScheduler(Scheduler scheduler);
  public void setSchedulingParameters(
              SchedulingParameters parameters);
}
```

A.5 AsynchronouslyInterruptedException

```
package javax.realtime;
public class AsynchronouslyInterruptedException
       extends java.lang.InterruptedException
{
  // constructores
  public AsynchronouslyInterruptedException();

  // métodos
  public synchronized boolean disable();
  public boolean doInterruptible (Interruptible logic);
  public synchronized boolean enable();•734 AP ´ENDICE
  public synchronized boolean fire();
  public static AsynchronouslyInterruptedException getGeneric();
  public boolean happened (boolean propagate);
  public boolean isEnabled();
  public void propagate();
}
```

A.6 BoundAsyncEventHandler

```
package javax,realtime;
public abstract class BoundAsyncEventHandler
                extends AsyncEventHandler
{
  // constructores
  public BoundAsyncEventHandler();
  public BoundAsyncEventHandler(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group,
         boolean nonheap);
}
```

A.7 Clock

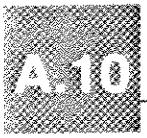
```
package javax.realtime;
public abstract class Clock
{
  // constrcutores
  public Clock();

  // métodos
  public static Clock getRealtimeClock();
  public abstract RelativeTime getResolution();
  public AbsoluteTime getTime();
  public abstract void getTime(AbsoluteTime time);
  public abstract void setResolution(RelativeTime resolution);
}
```

A.8 HighResolutionTime

```
package javax.realtime;
public abstract class HighResolutionTime
       implements java.lang.Comparable
{
  // métodos
  public abstract AbsoluteTime absolute(Clock clock,
                  AbsoluteTime destination);
  public int compareTo(HighResolutionTime time);
  public int compareTo(java.lang.Object object);
  public boolean equals(HighResolutionTime time);
  public boolean equals(java.lang.Object object);
  public final long getMilliseconds();
  public final int getNanoseconds();
  public int hashCode();
  public void set(HighResolutionTime time);
  public void set(long millis);
  public void set(long millis, int nanos);
}
```

ImmortalMemory

```
package javax.realtime;
public final class ImmortalMemory extends MemoryArea
{
  // métodos
  public static ImmortalMemory instance();
}
```



ImportanceParameters

```
package javax.realtime;
public class ImportanceParameters extends PriorityParameters
{
  // constructores
  public ImportanceParameters(int priority, int importance);

  // métodos
  public int getImportance();
  public void setImportance(int importance);
  public java.lang.String toString();
}
```

Interruptible

```
package javax.realtime;
public interface Interruptible
{
  public void interruptAction(
              AsynchronouslyInterruptedException exception);
  public void run(AsynchronouslyInterruptedException exception)
         throws AsynchronouslyInterruptedException;
}
```

A.12 LTMemory

```
public class LTMemory extends ScopedMemory
{
  // constructores
  public LTMemory(long initialSizeInBytes, long maxSizeInBytes);
}
```

A.13 MemoryArea

```
public abstract class MemoryArea
{
  // constructores
  protected MemoryArea(long sizeInBytes);

  // métodos
  public void enter(java.lang.Runnable logic);
  public static MemoryArea getMemoryArea(java.lang.Object object);
  public long memoryConsumed();
  public long memoryRemaining();
  public synchronized java.lang.Object newArray(
         java.lang.class type, int number)
         throws IllegalAccessException, InstantiationException,
         OutOfMemoryError;
  public synchronized java.lang.Object newInstance(
         java.lang.class type)
         throws IllegalAccessException, InstantiationException,
         OutOfMemoryError;
  public long size();
}
```

A.14 MemoryParameters

```
public class MemoryParameters
{
  // atributos
```

```
  public static final long NO_MAX;

  // constructores
  public MemoryParameters(long maxMemoryArea, long maxImmortal)
         throws IllegalArgumentException;
  public MemoryParameters(long maxMemoryArea, long maxImmortal,
                          long allocationRate)
         throws IllegalArgumentException;

  // métodos
  public long getAllocationRate();
  public long getMaxImmortal();
  public long getMaxMemoryArea();
  public void setAllocationRate(long rate);
  public boolean setMaxImmortal(long maximum);
  public boolean setMaxMemoryArea(long maximum);
}
```

A.15 MonitorControl

```
package javax.realtime;
public abstract class MonitorControl
{
  // constructores
  public MonitorControl();

  // métodos
  public static void setMonitorControl(MonitorControl policy);
  public static void setMonitorControl(java.lang.Object monitor,
                                       MonitorControl policy);
}
```

A.16 NoHeapRealtimeThread

```
package javax.realtime;
public class NoHeapRealtimeThread extends RealtimeThread
{
  // constructores
```

```
  public NoHeapRealtimeThread(SchedulingParameters scheduling,
         MemoryArea area) throws IllegalArgumentException;

  public NoHeapRealtimeThread(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryArea area)
         throws IllegalArgumentException;

  public NoHeapRealtimeThread(SchedulingParameters scheduling,
         ReleaseParameters release, MemoryParameters memory,
         MemoryArea area, ProcessingGroupParameters group,
         java.lang.Runnable logic)
              throws IllegalArgumentException;
}
```

A.17 OneShotTimer

```
package javax.realtime;
public class OneShotTimer extends Timer
{
  // constructores
  public OneShotTimer(HighResolutionTime time,
                      AsyncEventHandler handler);
  public OneShotTimer(HighResolutionTime start, Clock clock,
                      AsyncEventHandler handler);
}
```

A.18 PeriodicParameters

```
package javax.realtime;
public class PeriodicParameters extends ReleaseParameters
{
  // constructores
  public PeriodicParameters(HighResolutionTime start,
         RelativeTime period, RelativeTime cost,
         RelativeTime deadline, AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

  // métodos
```

```
  public RelativeTime getPeriod();
  public HighResolutionTime getStart();
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);
}
```

A.19 PeriodicTimer

```
package javax.realtime;
public class PeriodicTimer extends Timer
{
  // constructores
  public PeriodicTimer(HighResolutionTime start,
                       RelativeTime interval,
                       AsyncEventHandler handler);
  public PeriodicTimer(HighResolutionTime start,
                       RelativeTime interval,
                       Clock clock, AsyncEventHandler handler);

  // métodos
  public ReleaseParameters createReleaseParameters();
  public void fire();
  public AbsoluteTime getFireTime();•AP ´ENDICE 739
  public RelativeTime getInterval();
  public void setInterval(RelativeTime interval);
}
```

A.20 POSIXSignalHandler

```
package javax.realtime;
public final class POSIXSignalHandler
{
  // atributos
  public static final int SIGABRT;
  public static final int SIGALRM;
  public static final int SIGBUS;
  public static final int SIGCANCEL;
  public static final int SIGCHLD;
```

```
public static final int SIGCLD;
public static final int SIGCONT;
public static final int SIGEMPT;
public static final int SIGFPE;
public static final int SIGFREEZE;
public static final int SIGHUP;
public static final int SIGILL;
public static final int SIGINT;
public static final int SIGIO;
public static final int SIGIOT;
public static final int SIGKILL;
public static final int SIGLOST;
public static final int SIGLWP;
public static final int SIGPIPE;
public static final int SIGPOLL;
public static final int SIGPROF;
public static final int SIGPWF;
public static final int SIGQUIT;
public static final int SIGSEGV;
public static final int SIGSTOP;
public static final int SIGSYS;
public static final int SIGTERM;
public static final int SIGTHAW;
public static final int SIGTRAP;
public static final int SIGTSTP;
public static final int SIGTTIN;
public static final int SIGTTOU;
public static final int SIGURG;
public static final int SIGUSR1;
public static final int SIGUSR2;
public static final int SIGVTALRM;
public static final int SIGWINCH;
public static final int SIGXCPU;
public static final int SIGXFSZ;

// métodos
public static synchronized void addHandler(int signal,
       AsyncEventHandler handler);
public static synchronized void removeHandler(int signal
       AsyncEventHandler handler);
```

```
    public static synchronized void setHandler(int signal,
            AsyncEventHandler handler);
}
```

A.21 PriorityCeilingEmulation

```
package javax.realtime;
public class PriorityCeilingEmulation extends MonitorControl
{
  // constructores
  public PriorityCeilingEmulation(int ceiling);

  // métodos
  public int getDefaultCeiling();
}
```

A.22 PriorityInheritance

```
package javax.realtime;
public class PriorityInheritance extends MonitorControl
{
  // constructores
  public PriorityInheritance();

  // métodos
  public static PriorityInheritance instance();
}
```

A.23 PriorityParameters

```
package javax.realtime;
public class PriorityParameters extends SchedulingParameters
{
  // constructores
  public PriorityParameters(int priority);
```

```
  // métodos
  public int getPriority();
  public void setPriority(int priority) throws IllegalArgumentException;
  public java.lang.String toString();
}
```

A.24 PriorityScheduler

```
package javax.realtime;
public class PriorityScheduler extends Scheduler
{
  // constructores
  public PriorityScheduler() ;

  // métodos
  protected void addToFeasibility(Schedulable schedulable);
  boolean changeIfFeasible(Schedulable schedulable,
          ReleaseParameters release, MemoryParameters memory);
  public void fireSchedulable(Schedulable schedulable);
  public int getMaxPriority();
  public static int getMaxPriority(java.lang.Thread thread);
  public int getMinPriority();
  public static int getMinPriority(java.lang.Thread thread);
  public int getNormPriority();
  public static int getNormPriority(java.lang.Thread thread);
  public java.lang.String getPolicyName();
  public static PriorityScheduler instance();
  public boolean isFeasible();
  protected void removeFromFeasibility(Schedulable schedulable);
}
```

A.25 ProcessingGroupParameters

```
package javax.realtime;
public class ProcessingGroupParameters
{
  // constructores
```

```
  public ProcessingGroupParameters(HighResolutionTime start,
         RelativeTime period, RelativeTime cost,
         RelativeTime deadline, AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

  // métodos
  public RelativeTime getCost();
  public AsyncEventHandler getCostOverrunHandler();
  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler();
  public RelativeTime getPeriod();
  public HighResolutionTime getStart();
  public void setCost(RelativeTime cost);
  public void setCostOverrunHandler(AsyncEventHandler handler);
  public void setDeadline(RelativeTime deadline);
  public void setDeadlineMissHandler(AsyncEventHandler handler);
  public void setPeriod(RelativeTime period);
  public void setStart(HighResolutionTime start);
}
```

RationalTime

```
package javax.realtime;
public class RationalTime extends RelativeTime
{
  // constructores
  public RationalTime(int frequency);
  public RationalTime(int frequency, long millis, int nanos)
         throws IllegalArgumentException;
  public RationalTime( int frequency, RelativeTime interval)
         throws IllegalArgumentException;

  // métodos
  public AbsoluteTime absolute(Clock clock, AbsoluteTime destination);
  public void addInterarrivalTo(AbsoluteTime destination);
  public int getFrequency();
  public RelativeTime getInterarrivalTime(RelativeTime destination);
  public void set(long millis, int nanos)
         throws IllegalArgumentException;
```

```
  public void setFrequency(int frequency)
         throws ArithmeticException;
}
```

A.27 RawMemoryAccess

```
package javax.realtime;
public class RawMemoryAccess
{
  // constructores
  protected RawMemoryAccess(long base, long size);
  protected RawMemoryAccess(RawMemoryAccess memory, long base,
                            long size);

  // métodos
  public static RawMemoryAccess create(java.lang.Object type,
         long size) throws SecurityException, OffsetOutOfBoundsException,
         SizeOutOfBoundsException, UnsupportedPhysicalMemoryException;

  public static RawMemoryAccess create(java.lang.Object type,
                                       long base, long size)
         throws SecurityException, OffsetOutOfBoundsException,
         SizeOutOfBoundsException, UnsupportedPhysicalMemoryException;
  public byte getByte(long offset) throws SizeOutOfBoundsException,
         OffsetOutOfBoundsException;
  public void getBytes(long offset, byte[] bytes, int low,
                       int number) throws
              SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public int getInt(long offset) throws SizeOutOfBoundsException,
         OffsetOutOfBoundsException;
  public void getInts(long offset, int[] ints, int low,
                      int number) throws
                      SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public long getLong(long offset) throws SizeOutOfBoundsException,
         OffsetOutOfBoundsException;
  public void getLongs(long offset, long[] longs, int low,
                       int number) throws
         SizeOutOfBoundsException, OffsetOutOfBoundsException;
  public long getMapped Address();
```

```
    public short getshort(long offset) throws SizeOutOfBoundsException,
           OffsetOutOfBoundsException;
    public void getshorts(long offset, short[] shorts, int low,
                          int number) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public long map();
    public long map(long base);
    public long map(long base, long size);
    public void setByte(long offset, byte value) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public void setBytes(long offset, byte[] bytes, int low,
                         int number) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public void setInt(long offset, int value) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public void setInts(long offset, int[] ints, int low,
                        int number) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public void setLong(long offset, long value) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public void setLongs(long offset, long[] longs, int low,
                         int number) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public void setShort(long offset, short value) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public void setShorts(long offset, short[] shorts, int low,
                          int number) throws
                SizeOutOfBoundsException, OffsetOutOfBoundsException;
    public void unmap();
}
```

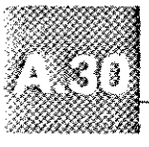
A.28 Realtime Security

```
package javax.realtime;
public class RealtimeSecurity
{
  // constructores
  public RealtimeSecurity();

  // métodos
```

```
  public void checkAccessPhysical() throws
         SecurityException;
  public void checkAccessPhysicalRange(long base, long size) throws
         SecurityException;
  public void checkSetFactory() throws
         SecurityException;
  public void checkSetScheduler() throws
         SecurityException;
}
```

Realtime System

```
package javax.realtime;
public class RealtimeSystem
{
  // atributos
  public static final byte BIG_ENDIAN;
  public static final byte BYTE_ORDER;
  public static final byte LITTLE_ENDIAN;

  // métodos
  public static GarbageCollector currentGC();
  public int getConcurrentLocksUsed();
  public int getMaximumConcurrentLocks();
  public static RealtimeSecurity getSecurityManager();
  public void setMaximumConcurrentLocks(int number);
  public void setMaximumConcurrentLocks(int number, boolean hard);
  public static voidgetSecurityManager(RealtimeSecurity manager) throws
         SecurityException;
}
```



RealtimeThread

```
package javax.realtime;
public class RealtimeThread extends java.lang.Thread
       implements Schedulable
{
  // constructores
```

```
public RealtimeThread();
public RealtimeThread(SchedulingParameters scheduling);
public RealtimeThread(SchedulingParameters scheduling,
                      ReleaseParameters release);
public RealtimeThread(SchedulingParameters scheduling,
       ReleaseParameters release, MemoryParameters memory,
       MemoryArea area, ProcessingGroupParameters group,
       java.lang.Runnable logic);

// métodos
public void addToFeasibility();
public static RealtimeThread currentRealtimeThread();
public synchronized void deschedulePeriodic();
public MemoryArea getMemoryArea();•AP ´ENDICE 745
public MemoryParameters getMemoryParameters();
public ProcessingGroupParameters getProcessingGroupParameters();
public ReleaseParameters getReleaseParameters();
public Scheduler getScheduler();
public SchedulingParameters getSchedulingParameters();
public synchronized void interrupt();
public void removeFromFeasibility();
public synchronized void schedulePeriodic();
public void setMemoryParameters(MemoryParameters parameters);
public void setProcessingGroupParameters(
            ProcessingGroupParameters parameters);
public void setReleaseParameters(ReleaseParameters parameters);
public void setScheduler(Scheduler scheduler)
       throws IllegalThreadStateException;
public void setSchedulingParameters(
            SchedulingParameters scheduling);
public static void sleep(Clock clock, HighResolutionTime time)
       throws InterruptedException;
public static void sleep(HighResolutionTime time)
       throws InterruptedException;
public boolean waitForNextPeriod()
       throws IllegalThreadStateException ;
}
```

A.31 RelativeTime

```
package javax.realtime;
public class RelativeTime extends HighResolutionTime
{
  // constructores
  public RelativeTime();
  public RelativeTime(long millis, int nanos);
  public RelativeTime(RelativeTime time);

  // métodos
  public AbsoluteTime absolute(Clock clock,
                               AbsoluteTime destination);
  public RelativeTime add(long millis, int nanos);
  public RelativeTime add(long millis, int nanos,
                          RelativeTime destination);
  public final RelativeTime add(RelativeTime time);
  public RelativeTime add(RelativeTime time, RelativeTime destination);
  public void addInterarrivalTo(AbsoluteTime destination);
  public RelativeTime getInterarrivalTime(RelativeTime destination);
  public final RelativeTime subtract(RelativeTime time);
  public RelativeTime subtract(RelativeTime time,
                               RelativeTime destination);
  public java.lang.String toString();
}
```

A.32 ReleaseParameters

```
package javax.realtime;
public abstract class ReleaseParameters
{
  // constructores
  protected ReleaseParameters(RelativeTime cost, RelativeTime deadline,
            AsyncEventHandler overrunHandler,
            AsyncEventHandler missHandler);

  // métodos
  public RelativeTime getCost();
```

```
  public AsyncEventHandler getCostOverrunHandler();
  public RelativeTime getDeadline();
  public AsyncEventHandler getDeadlineMissHandler();
  public void setCost(RelativeTime cost);
  public void setCostOverrunHandler(AsyncEventHandler handler);
  public void setDeadline(RelativeTime deadline);
  public void setDeadlineMissHandler(AsyncEventHandler handler);
}
```

A.33 Schedulable

```
package javax.realtime;
public interface Schedulable extends java.lang.Runnable
{
  // métodos
  public void addToFeasibility();
  public MemoryParameters getMemoryParameters();
  public ReleaseParameters getReleaseParameters();
  public Scheduler getScheduler();
  public SchedulingParameters getSchedulingParameters();
  public void removeFromFeasibility();
  public void setMemoryParameters(MemoryParameters memory);
  public void setReleaseParameters(ReleaseParameters release);
  public void setScheduler(Scheduler scheduler);
  public void setSchedulingParameters(SchedulingParameters scheduling);
}
```

A.34 Scheduler

```
package javax.realtime;
public abstract class Scheduler
{
  // constructores
  public Scheduler();
  // métodos
  protected abstract void addToFeasibility(Schedulable schedulable);
  public boolean changeIfFeasible(Schedulable schedulable,
                  ReleaseParameters release, MemoryParameters memory);
```

```
  public static Scheduler getDefaultScheduler();
  public abstract java.lang.String getPolicyName();
  public abstract boolean isFeasible();
  protected abstract void removeFromFeasibility(Schedulable schedulable);
  public static void setDefaultScheduler(Scheduler scheduler);
}
```

A.35 SchedulingParameters

```
package javax.realtime;
public abstract class SchedulingParameters
{
  // constructores
  public SchedulingParameters();
}
```

A.36 ScopedMemory

```
package javax.realtime;
public abstract class ScopedMemory extends MemoryArea
{
  // constructores
  public ScopedMemory(long size);

  // métodos
  public void enter(java.lang.Runnable logic);
  public int getMaximumSize();
  public MemoryArea getOuterScope();
  public java.lang.Object getPortal();
  public void setPortal(java.lang.Object object);
}
```

A.37 ScopedPhysicalMemory

```
package javax.realtime;
public class ScopedPhysicalMemory extends ScopedMemory;
```

```
{
  // constructores
  protected ScopedPhysicalMemory(long base, long size);
  protected ScopedPhysicalMemory(ScopedPhysicalMemory memory,
                                 long base, long size);

  // métodos
  public static ScopedPhysicalMemory create(java.lang.Object type,
                long base, long size) throws SecurityException,
                OffsetOutOfBoundsException, SizeOutOfBoundsException,
                UnsupportedPhysicalMemoryException;
  public static void setFactory(PhysicalMemoryFactory factory);
}
```

A.38 SporadicParameters

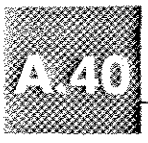
```
package javax.realtime;
public class SporadicParameters extends AperiodicParameters
{
  //constructores
  public SporadicParameters(RelativeTime minInterarrival,
         RelativeTime cost, RelativeTime deadline,
         AsyncEventHandler overrunHandler,
         AsyncEventHandler missHandler);

  // métodos
  public RelativeTime getMinimumInterarrival();
  public void setMinimumInterarrival(RelativeTime minimum);
}
```

A.39 Thread

```
// Not in package javax.realtime;
public class Thread extends Object implements Runnable
{
  // atributos
  static int MAX_PRIORITY;
  static int MIN_PRIORITY;
```

```
static int NORM_PRIORITY;

// constructores
public Thread();
public Thread(Runnable target);
public Thread(Runnable target, String name);
public Thread(String name);
public Thread(ThreadGroup group, String name);
       throws SecurityException, IllegalThreadStateException
public Thread(ThreadGroup group, Runnable target);
       throws SecurityException, IllegalThreadStateException
public Thread(ThreadGroup group, Runnable target, String name);
       throws SecurityException, IllegalThreadStateException

// métodos
static int activeCount();
void checkAccess() throws SecurityException;
int countStackFrames()
    throws IllegalThreadStateException;
public static Thread currentThread();
public void destroy() throws SecurityException;
static void dumpStack();
static int enumerate(Thread[] tarray)
       throws SecurityException;
ClassLoader getContextClassLoader() throws SecurityException;
String getName();
int getPriority();
ThreadGroup getThreadGroup();.
void interrupt() throws SecurityException;
static boolean interrupted();
public final native boolean isAlive();
public final boolean isDaemon();
boolean isInterrupted();
public final void join() throws InterruptedException;
public final void join(long millis)
       throws InterruptedException;
public final void join(long millis, int nanos)
       throws InterruptedException;
public void resume()
       throws SecurityException; // DEPRECATED
public void run();
```

```
  public void setContextClassLoader(ClassLoader cl)
         throws SecurityException;
  public final void setDaemon()
         throws SecurityException, IllegalThreadStateException;
  public void setName(String name)
         throws SecurityException;
  public void setPriority(int newPriority)
         throws class IllegalArgumentException, InterruptedException;
  public static void sleep(long millis)
         throws InterruptedException;
  public static void sleep(long millis, int nanos)
         throws class IllegalArgumentException, InterruptedException;
  public native synchronized void start()
         throws IllegalThreadStateException;
  public final void stop()
         throws SecurityException; // DEPRECATED
  public final synchronized void stop(Throwable o);
         throws SecurityException, NullPointerException; // DEPRECATED
  public void suspend()
         throws SecurityException; // DEPRECATED
  public String toString();
  public static void yield();
}
```



A.40 ThreadGroup

```
// Not in package javax.realtime;
public class ThreadGroup {
  // constructores
  public ThreadGroup(String name)
         throws SecurityException;
  public ThreadGroup(ThreadGroup parent, String name)
         throws SecurityException, NullPointerException;

  // métodos
  public int activeCount();
  public int activeGroupCount();
  public boolean allowThreadSuspension(boolean b); // DEPRECATED
  public final void checkAccess()
         throws SecurityException;
```

```
  public final void destroy()
         throws IllegalThreadStateException, SecurityException;
  public int enumerate(Thread[] list)
         throws SecurityException;
  public int enumerate(Thread[] list, boolean recurse)
         throws SecurityException;
  public int enumerate(ThreadGroup[] list)
         throws SecurityException;
  public int enumerate(ThreadGroup[] list, boolean recurse)
         public int getMaxPriority()
  public String getName()
  public final ThreadGroup getParent()
         throws SecurityException;
  public void interrupt()
         throws SecurityException;
  public final boolean isDaemon();
  public synchronized boolean isDestroyed();
  public void list()
  public final boolean parentOf(ThreadGroup g);
  public void resume()
         throws SecurityException; // DEPRECATED
  public final void setDaemon(boolean daemon);
               throws SecurityException;
  public void setMaxPriority(int pri)
         throws SecurityException;
  public final void stop()
               throws SecurityException; // DEPRECATED
  public void suspend()
         throws SecurityException; // DEPRECATED
  public String toString();
  public void uncaughtException(Thread t, Throwable e)
}
```

A.41 Timed

```
package javax.realtime;
public class Timed extends AsynchronouslyInterruptedException
                   implements java.io.Serializable
{
  // constructores
```

```
  public Timed(HighResolutionTime time) throws
              IllegalArgumentException;
  // métodos
  public boolean doInterruptible(Interruptible logic);
  public void resetTime(HighResolutionTime time);
}
```

A.42 Timer

```
package javax.realtime;
public abstract class Timer extends AsyncEvent
{
  // constructores
  protected Timer(HighResolutionTime time, Clock clock,
                  AsyncEventHandler handler);
  // métodos
  public ReleaseParameters createReleaseParameters();
  public void disable();
  public void enable();
  public Clock getClock();
  public AbsoluteTime getFireTime();
  public void reschedule(HighResolutionTime time);
  public void start();
}
```

A.43 VTMemory

```
package javax.realtime;
public class VTMemory extends ScopedMemory
{
  // constructores
  VTMemory(int initial, int maximum);
}
```

Índice

B

C

D

E

F

G

H

I

J

R

S

T