Sistemas de tiempo real

Nuestro tratamiento de los temas relativos a los sistemas operativos se ha centrado principalmente en los sistemas informáticos de propósito general (por ejemplo, los sistemas de sobremesa y sistemas servidores). En este capítulo, vamos a volver nuestra atención a los sistemas informáticos de tiempo real. Los requisitos de los sistemas de tiempo de real difieren de los de muchos de los sistemas que hemos descrito hasta el momento, principalmente porque los sistemas de tiempo real deben producir los resultados dentro de ciertos límites de tiempo. En este capítulo, vamos a proporcionar una panorámica de los sistemas informáticos en tiempo real y a describir cómo deben construirse para satisfacer los estrictos requisitos de tiempo de estos sistemas.

OBJETIVOS DEL CAPÍTULO

- Explicar los requisitos de temporización de los sistemas de tiempo real.
- Distinguir entre sistemas de tiempo real estrictos y no estrictos.
- Explicar las características que definen a los sistemas de tiempo real.
- Describir los algoritmos de planificación para los sistemas de tiempo real estrictos.

19.1 Introducción

Un **sistema de tiempo real** es un sistema informático que no sólo requiere que los resultados calculados sean "correctos", sino que también esos resultados se produzcan dentro de un período de tiempo especificado. Los resultados producidos después de que ese período de tiempo haya transcurrido pueden no tener (incluso aunque sean correctos) ningún valor real. Como ilustración, considere un robot autónomo que se encarga de distribuir el correo en un complejo de oficinas. Si el sistema de control de visión identifica una pared *después* de que el robot haya impactado contra ella, el sistema no habrá satisfecho los requisitos previstos, incluso aunque haya identificado correctamente la pared. Compare este requisito de temporización con las demandas mucho menos estrictas de otros sistemas. En un sistema informático de sobremesa interactivo, resulta deseable proporcionar un rápido tiempo de respuesta al usuario interactivo, pero no es obligatorio hacerlo. Algunos sistemas (como los sistemas de procesamiento por lotes) pueden que no tengan, incluso, ningún requisito de temporización en absoluto.

Los sistemas de tiempo real que se ejecutan sobre hardware informático tradicional se utilizan en un amplio rango de aplicaciones. Además, muchos sistemas de tiempo real están integrados en "dispositivos especializados", como electrodomésticos normales, (por ejemplo, hornos de microondas y lavavajillas), dispositivos digitales de consumo (por ejemplo, cámaras y reproductores MP3) y dispositivos de comunicaciones (por ejemplo, teléfonos móviles y dispositivos de mano Black-berry). También están presentes en otros aparatos de mayor tamaño, como automóviles o

aeroplanos. Un **sistema integrado** es un dispositivo informático que forma parte de un sistema de mayor tamaño, en el que la presencia de ese dispositivo informático a menudo no resulta obvia para el usuario.

Como ejemplo, considere un sistema integrado para el control de un lavavajillas. El sistema integrado puede ofrecer diversas opciones para planificar la operación del lavavajillas: la temperatura del agua, el tipo de lavado (ligero o en profundidad) o incluso un temporizador que indique cuándo debe comenzar a funcionar el lavavajillas. Lo más probable es que el usuario del lavavajillas no sea consciente de que existe una computadora integrada en el electrodoméstico. Como ejemplo adicional, considere un sistema integrado que controla el sistema antibloqueo de los frenos en un automóvil. Cada rueda del automóvil tiene un sensor que detecta el grado de deslizamiento y de tracción en cada momento y cada sensor envía continuamente sus datos al controlador del sistema. Tomando los resultados de estos sensores, el controlador le dice al mecanismo de frenado de cada rueda cuanta presión de frenado debe aplicar. De nuevo, para el usuario (en este caso, el conductor del vehículo), la presencia de un sistema informático integrado puede no resultar aparente. Sin embargo, es importante destacar, que no todos los sistemas integrados son sistemas de tiempo real. Por ejemplo, un sistema integrado que controle un horno doméstico puede no tener ningún requisito de tiempo real en absoluto.

Algunos sistemas de tiempo real se identifican como **sistemas de seguridad crítica**. En un sistema de seguridad crítica, la operación incorrecta (usualmente debido a que no se cumple con un requisito de temporización) provoca algún tipo de "catástrofe". Como ejemplos de sistemas de seguridad crítica podemos citar los sistemas armamentísticos, los sistemas antibloqueo de los frenos, los sistemas de gestión de vuelo, los sistemas integrados que tengan relación con la salud, como por ejemplo los marcapasos. En estos escenarios, el sistema de tiempo real *debe* responder a los sucesos con los requisitos de temporización especificados; en caso contrario, podrían producirse graves daños o incluso algo peor. Sin embargo, una mayoría significativa de los sistemas integrados no son sistemas de seguridad crítica, incluyendo por ejemplo las máquinas fax, los hornos microondas, los relojes de pulsera y los dispositivos de red como conmutadores y encaminadores. Para estos dispositivos, si no se cumplen los requisitos de temporización, el peor de los resultados posibles es, en muchas ocasiones, un usuario insatisfecho.

La informática de tiempo de real puede clasificarse en dos tipos distintos: estricta y no estricta. Un **sistema de tiempo real estricto** tiene los requisitos más fuertes, que garantizan que las tareas de tiempo real críticas se completen dentro del período especificado. Los sistemas de seguridad crítica son, normalmente, sistemas de tiempo real estrictos. Los **sistemas de tiempo real no estrictos** no tienen requisitos tan fuertes, y se limitan a garantizar que las tareas de tiempo real críticas tengan prioridad sobre otras tareas y que retengan dicha prioridad hasta complementarse. Muchos sistemas operativos comerciales, así como Linux, proporcionan soporte de tiempo real no estricto.

19.2 Características del sistema

En esta sección, vamos a explorar las características de los sistemas de tiempo real y a analizar diversas cuestiones relacionadas con el diseño de sistemas operativos de tiempo real tanto estrictos como no estrictos.

Entre las características típicas de muchos sistemas operativos de tiempo real podemos citar:

- Tienen un único propósito.
- Son de pequeño tamaño.
- Son de bajo coste y se producen en masa.
- Tienen requisitos de temporización específicos.

A continuación, vamos a examinar cada una de estas características.

A diferencia de las máquinas de tipo PC, que se pueden utilizar para muchas cosas, un sistema en tiempo real sirve normalmente a un único propósito, como por ejemplo controlar el sistema

antibloqueo de los frenos o reproducir música en un reproductor MP3. ¡Resulta bastante poco probable que un sistema de tiempo de real que controle el sistema de navegación de una aeronave se utilice también para reproducir un DVD! El diseño de un sistema operativo de tiempo real refleja su naturaleza concentrada en un único propósito y es a menudo bastante simple.

Existen muchos sistemas de tiempo real en entornos en los que el espacio físico está restringido. Considere la cantidad de espacio disponible en un reloj de pulsera o en un horno microondas: es considerablemente menor que el que hay disponible en una computadora de sobremesa. Debido a las restricciones de espacio, la mayoría de los sistemas de tiempo real carecen tanto de la potencia de procesamiento de la CPU como de la cantidad de memoria disponibles en las computadoras de sobremesa estándar. Mientras que la mayoría de los sistemas contemporáneos de sobremesa y de servidor utilizan procesadores de 32 o 64 bits, muchos sistemas de tiempo real se ejecutan sobre procesadores de 8 o 16 bits. De forma similar, un PC de sobremesa puede tener varios gigabytes de memoria física, mientras que un sistema de tiempo real puede tener menos de un megabyte. Denominamos **huella** de un sistema a la cantidad de memoria necesaria para ejecutar un sistema operativo y sus aplicaciones. Puesto que la cantidad de memoria es limitada, la mayoría de los sistemas operativos deben tener huellas de pequeño tamaño.

A continuación, pensemos en dónde se implementan los sistemas en tiempo real. A menudo los podemos encontrar en electrodomésticos y dispositivos de consumo. Dispositivos como cámaras digitales, hornos de microondas y termostatos se fabrican en masa dentro de entornos de mercado en los que el coste es un factor fundamental. Por tanto, los microprocesadores para los sistemas de tiempo real también tienen que poder fabricarse masivamente y a bajo coste.

Una técnica para reducir el coste de un controlador integrado consiste en emplear una técnica alternativa para organizar los componentes del sistema informático. En lugar de organizar la computadora de acuerdo con la estructura mostrada en la Figura 19.1, en la que una serie de buses proporcionan el mecanismo de interconexión con los componentes individuales, muchos controladores para sistemas integrados utilizan una estrategia conocida con el nombre de **sistema en un chip** (SOC, system-on-chip). En este tipo de sistemas, la CPU, la memoria (incluyendo la caché), la unidad de gestión de memoria (MMU, memory-management-unit) y los puertos periféricos asociados, como los puertos USB, están contenidos en un único circuito integrado. Esta estrategia SOC suele ser menos costosa que la organización orientada a bus de la Figura 19.1.

Fijémonos ahora en la característica final de los sistemas de tiempo real que hemos identificado anteriormente: los requisitos específicos de temporización. Se trata, de hecho, de la característica definitoria de este tipo de sistemas. Correspondientemente, la característica definitoria de los sistemas operativos de tiempo real tanto estrictos como no estrictos consiste en dar soporte a los requisitos de temporización de las tareas en tiempo real, y en el resto del capítulo nos vamos a centrar en esta cuestión. Los sistemas operativos de tiempo real cumplen con los requisitos de temporización utilizando algoritmos de planificación que proporcionan a los procesos de tiempo real las más altas prioridades de planificación. Además, los planificadores deben garantizar que

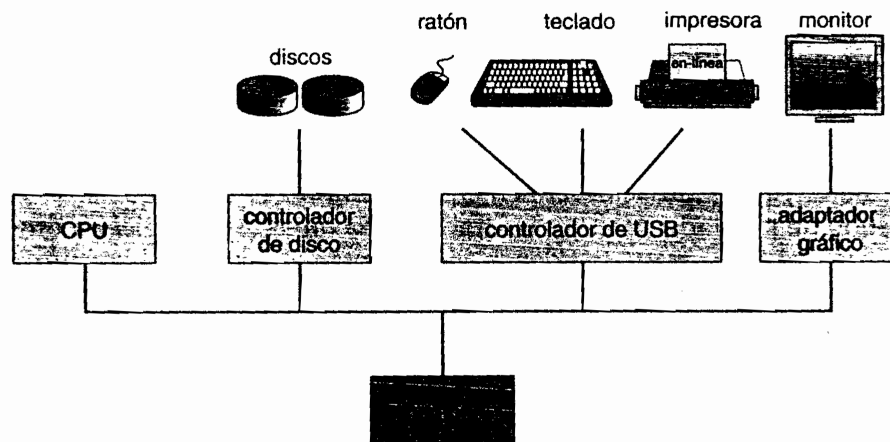


Figura 19.1 Organización orientada a bus.

la prioridad de una tarea de tiempo real no se degrade con el tiempo. Una segunda técnica, en cierto modo relacionada con la anterior, que se utiliza para satisfacer los requisitos de temporización consiste en minimizar el tiempo de respuesta a sucesos como las interrupciones.

19.3 Características de un kernel de tiempo real

En esta sección, vamos a analizar las características necesarias para diseñar un sistema operativo que permita procesos en tiempo real. Sin embargo, antes de comenzar, vamos a analizar lo que normalmente *no* se necesita para un sistema en tiempo real. Comenzaremos examinando varias de las características proporcionadas en muchos de los sistemas operativos que hemos analizado hasta ahora en el texto, incluyendo Linux, UNIX y las diversas versiones de Windows. Estos sistemas proporcionan normalmente soporte para las siguientes características:

- Una diversidad de dispositivos periféricos, tales como pantallas gráficas, unidades de CD y de DVD.
- Mecanismos de protección y seguridad.
- Múltiples usuarios.

Soportar estas características requiere a menudo un *kernel* sofisticado y de gran tamaño, por ejemplo, Windows XP tiene más de 40 millones de líneas de código fuente. Por contraste, un sistema operativo de tiempo real típico suele tener un diseño muy simple, que a menudo está escrito en miles, en lugar de en millones, de líneas de código fuente. No cabe razonablemente esperar que estos sistemas simples puedan soportar las características que hemos enumerado anteriormente.

¿Pero por qué los sistemas de tiempo real no proporcionan estas características, que son cruciales para los sistemas estándar de sobremesa y de servidor? Existen varias razones, pero las fundamentales son tres. En primer lugar, porque la mayoría de los sistemas en tiempo real sirven a un único propósito, por lo que no requieren muchas de las características que podemos encontrar en un PC de sobremesa. Considere un reloj de pulsera digital: obviamente, no necesita disponer de una unidad de disco o de DVD, ni tampoco una memoria virtual. Además, un sistema de tiempo real no incluye la noción de usuario: el sistema simplemente permite un número pequeño de tareas, que a menudo están a la espera de una serie de entradas que tienen que recibir de dispositivos hardware (sensores, aparatos de visión, etc.). En segundo lugar, las características soportadas por los sistemas operativos estándar de sobremesa son imposibles de proporcionar sin utilizar procesadores rápidos y grandes cantidades de memoria. Tanto unos como otra no están disponibles en los sistemas de tiempo de real debido a restricciones de espacio, como hemos explicado anteriormente. Además, muchos sistemas de tiempo real carecen del suficiente espacio como para soportar unidades de disco periféricas o pantallas gráficas, aunque algunos sistemas admiten sistemas de archivos utilizando memoria no volátil (NVRAM). En tercer lugar, soportar las características que resultan comunes en los entornos informáticos estándar de sobremesa incrementaría enormemente el coste de los sistemas de tiempo real, lo que podría hacer que dichos sistemas no resultaran factibles desde un punto de vista económico.

Hay una serie de consideraciones adicionales que entran en juego cuando consideramos el tema de la memoria virtual en un sistema de tiempo real. Proporcionar las características de memoria virtual que se describen en el Capítulo 9 requiere que el sistema incluya una unidad de gestión de memoria (MMU) para traducir las direcciones lógicas a direcciones físicas. Sin embargo, las unidades MMU tienden a incrementar el coste y el consumo de potencia del sistema. Además, el tiempo requerido para traducir las direcciones lógicas a las direcciones físicas, especialmente en el caso de que se produzca un fallo de búfer de traducción (TLB, translation look-aside buffer), puede ser prohibitivo en un entorno de tiempo real. En el resto de esta sección, vamos a examinar varias técnicas que se utilizan para traducir direcciones en los sistemas de tiempo real.

La Figura 19.2 ilustra tres diferentes estrategias para la gestión de traducción de direcciones que los diseñadores de sistemas operativos en tiempo real pueden utilizar. En este escenario, la

http://libreria-universitaria.blogspot.com

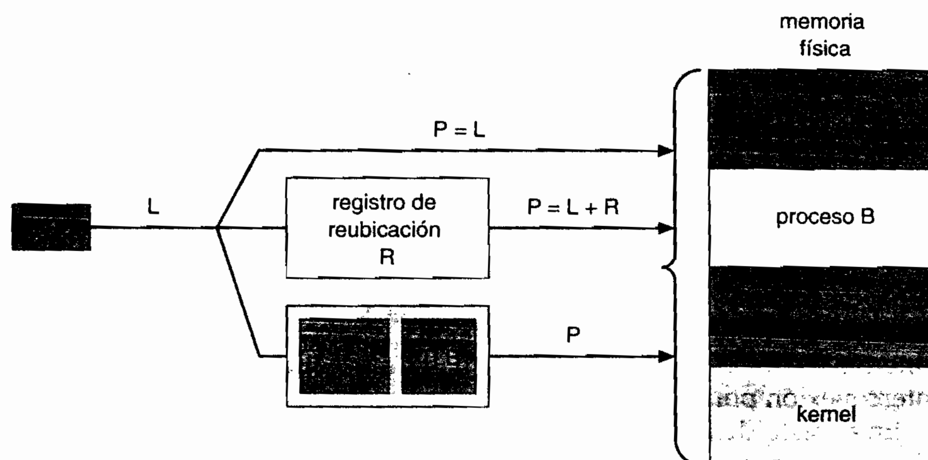


Figura 19.2 Traducción de direcciones en los sistemas de tiempo real.

CPU genera la dirección lógica *L* que debe hacerse corresponder con la dirección física *P*. La primera técnica consiste en puentear las direcciones lógicas y hacer que la CPU genere directamente las direcciones físicas. Esta técnica (denominada **modo de direccionamiento real**) no emplea mecanismos de memoria virtual y equivale, en la práctica, a decir que *P* equivale a *L*. Un problema con el modo de direccionamiento real es la ausencia de protección de memoria entre los procesos. El modo de direccionamiento real puede también requerir que los programadores especifiquen la ubicación física donde deben cargarse sus programas en memoria. Sin embargo, la ventaja de esta técnica es que el sistema es bastante rápido, ya que no se pierde ningún tiempo en traducir las direcciones. El modo de direccionamiento real resulta bastante común en los sistemas integrados con restricciones de tiempo real estrictas. De hecho, algunos sistemas operativos de tiempo real que se ejecutan sobre microprocesadores que contienen una MMU suelen desactivar la MMU para incrementar la velocidad haciendo directamente referencia a las direcciones físicas.

Una segunda estrategia para traducir las direcciones consiste en utilizar una técnica similar a la del registro de reubicación dinámica mostrado en la Figura 8.4. En este escenario, se configura un registro de reubicación *R* con la ubicación de memoria en la que se ha cargado un programa. La dirección física *P* se genera sumando el contenido del registro de reubicación *R* con *L*. Algunos sistemas de tiempo real configuran la MMU para que funcione de esta forma. La ventaja obvia de esta estrategia es que la MMU puede traducir fácilmente las direcciones lógicas a direcciones físicas utilizando la ecuación $P = L + R$. Sin embargo, este sistema continuará sufriendo la desventaja de que no existe protección de memoria entre los procesos.

La última técnica consiste en que el sistema de tiempo real proporcione una funcionalidad completa de memoria virtual, como se describe en el Capítulo 9. En este caso, la traducción de direcciones tiene lugar mediante tablas de páginas y un búfer de traducción o TLD. Además de permitir cargar un programa en cualquier posición de memoria, esta estrategia también proporciona protección de memoria entre los procesos. Para los sistemas que no tengan unidades de disco conectadas, puede que no sea posible implementar los mecanismos de paginación bajo demanda y de intercambio. Sin embargo, los sistemas pueden proporcionar dichas características utilizando memorias flash NVRAM. Los sistemas LynxOS y OnCore Systems son ejemplos de sistemas operativos de tiempo real que proporcionan soporte completo de memoria virtual.

19.4 Implementación de sistemas operativos de tiempo real

Teniendo presentes las muchas posibles variaciones, vamos a identificar ahora las características necesarias para implementar un sistema operativo en tiempo real. Esta lista no pretende ser exhaustiva, algunos sistemas proporcionan más características de las que a continuación enumeramos, mientras que otros sistemas proporcionan menos.

- Planificación apropiativa basada en prioridades.
- *Kernel* apropiativo.
- Latencia minimizada.

Una característica notable que hemos omitido de esta lista es el soporte para la conexión por red. Sin embargo, decidir si deben soportarse protocolos de red, como por ejemplo TCP/IP, resulta bastante simple: si el sistema de tiempo real debe conectarse a una red, el sistema operativo deberá proporcionar capacidades de interconexión por red. Por ejemplo, un sistema que recopile datos de tiempo real y los transmita a un servidor, deberá obviamente incluir características de interconexión por red. Alternativamente, un sistema integrado autocontenido que no requiera ninguna interacción con otros sistemas informáticos no tendrá ningún requisito obvio para la interconexión por red.

En el resto de esta sección, examinaremos los requisitos básicos enumerados anteriormente e identificaremos cómo se les puede implementar en un sistema operativo de tiempo real.

19.4.1 Planificación basada en prioridades

La característica más importante de un sistema operativo de tiempo real consiste en responder inmediatamente a un proceso de tiempo real, en cuanto dicho proceso necesite la CPU. Como resultado, el planificador para un sistema de tiempo real debe permitir un algoritmo basado en prioridades con apropiación. Recuerde que los algoritmos de planificación basados en prioridades asignan a cada proceso una prioridad, dependiendo de su importancia, a las tareas más importantes se les asignan prioridades más altas que a aquellas que se considera que son menos importantes. Si el planificador también permite un mecanismo de apropiación, un proceso que se esté ejecutando actualmente en la CPU será desalojado si otro proceso de mayor prioridad pasa a estar disponible para ejecución.

Los algoritmos de planificación apropiativos basados en prioridades se han tratado en detalle en el Capítulo 5, donde presentamos ejemplos de las características de planificación de tiempo real no estricto de los sistemas operativos Solaris, Windows XP y Linux. Cada uno de estos sistemas asigna a los procesos de tiempo real la más alta prioridad de planificación. Por ejemplo, Windows XP tiene 32 niveles diferentes de prioridad; los niveles más altos (valores de prioridad 16 a 31) están reservados para los procesos de tiempo real. Solaris y Linux tienen esquemas de asignación de prioridades similares.

Observe sin embargo, que proporcionar un planificador apropiativo basado en prioridades sólo garantiza una funcionalidad de tiempo real no estricta. Los sistemas de tiempo real estrictos deben además garantizar que las tareas de tiempo real reciban servicio de acuerdo con sus requisitos de temporización, y para poder proporcionar tales garantías pueden necesitar características de planificación adicionales. En la Sección 19.5 analizamos los algoritmos de planificación adecuados para los sistemas de tiempo real estrictos.

19.4.2 Kernels apropiativos

Los *kernels* no apropiativos no permiten desalojar un proceso que se esté ejecutando en modo *kernel;* el proceso en modo *kernel* continuará ejecutándose hasta que salga del modo *kernel,* se bloquee o voluntariamente ceda el control de la CPU. Por el contrario, un *kernel* apropiativo permitirá desalojar una tarea que se esté ejecutando en modo *kernel.* El diseño de un *kernel* apropiativo puede resultar bastante difícil, y las aplicaciones tradicionales orientadas al usuario, como hojas de cálculo, procesadores de texto y exploradores web normalmente no requieren esos rápidos tiempos de respuesta. Como resultado, algunos sistemas operativos de sobremesa convencionales (como Windows XP) son no apropiativos.

Sin embargo, para satisfacer los requisitos de temporización de los sistemas de tiempo real (y en particular de los sistemas de tiempo real estrictos) resulta obligatorio utilizar un *kernel* apropiativo. En caso contrario, una tarea de tiempo real podría tener que esperar un período de tiempo arbitrariamente largo, mientras hubiera otra tarea activa en el *kernel.*

Existen diversas estrategias para hacer que un *kernel* sea apropiativo. Una de ellas consiste en insertar **puntos de desalojo** en las llamadas al sistema de larga duración. En cada punto de desalojo se comprueba si existe la necesidad de ejecutar un proceso de alta prioridad. En caso de que exista, se produce un cambio de contexto. Después, cuando el proceso de alta prioridad termine de ejecutarse, el proceso interrumpido continuará con su llamada al sistema. Los puntos de desalojo sólo pueden colocarse en puntos *seguros* del *kernel*, es decir, sólo en aquellos puntos donde las estructuras del *kernel* no estén siendo modificadas. Una segunda estrategia para hacer que un *kernel* sea apropiativo consiste en utilizar mecanismos de sincronización, como los que hemos descrito en el Capítulo 6. Con este método, el *kernel* siempre puede ser apropiativo, porque cualquier dato del *kernel* que se esté actualizando estará protegido frente a posibles modificaciones por parte del proceso de alta prioridad.

19.4.3 Minimización de la latencia

Consideremos la naturaleza dirigida por sucesos de un sistema de tiempo real: normalmente, el sistema está esperando a que se produzca un suceso en tiempo real. Los sucesos pueden producirse en el software (por ejemplo, cuando caduca un contador) o en el hardware (como por ejemplo cuando un vehículo con control remoto detecta que se está aproximando a un rectángulo). Cuando se produce un suceso, el sistema debe responder a él y darle servicio lo más rápidamente posible. Denominamos **latencia del suceso** a la cantidad de tiempo que transcurre desde el momento que tiene lugar el suceso hasta el momento en el que se le da servicio (Figura 19.3).

Normalmente, los diferentes sucesos tienen distintos requisitos de latencia. Por ejemplo, el requisito de latencia para un sistema antibloqueo de frenos puede ser de entre 3 y 5 milisegundos, lo que quiere decir que desde el momento en que una rueda detecta por primera vez que está deslizándose, el sistema que controla el mecanismo de antibloqueo de los frenos tiene entre 3 y 5 milisegundos para responder a esa situación y controlarla. Cualquier respuesta que requiera más tiempo puede provocar que el vehículo quede fuera de control. Por contraste, un sistema integrado que controle el radar en una aeronave puede tolerar un período de latencia de varios segundos.

Hay dos tipos de latencias que afectan al rendimiento de los sistemas de tiempo real:

1. Latencia de interrupción.
2. Latencia de despacho.

La **latencia de interrupción** se refiere al período de tiempo que transcurre entre la llegada de una interrupción a la CPU y el instante en que comienza la rutina de servicio de dicha interrupción. Cuando se produce una interrupción, el sistema operativo debe primero completar la instrucción que está ejecutando y determinar el tipo de interrupción que ha ocurrido. Luego, debe guardar el estado del proceso actual antes de dar servicio a la interrupción utilizando la rutina de servicio de interrupción (ISR, interrupt service routine) específica. El tiempo total requerido para

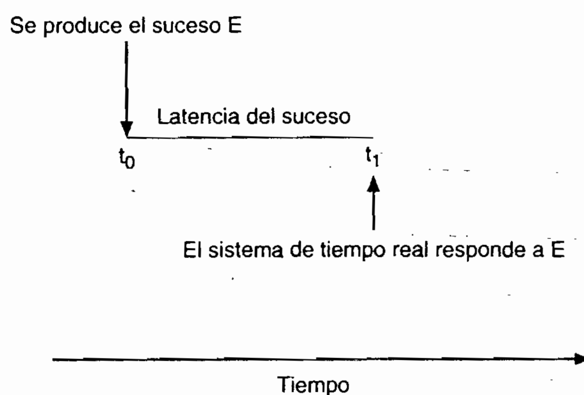


Figura 19.3 Latencia de un suceso.

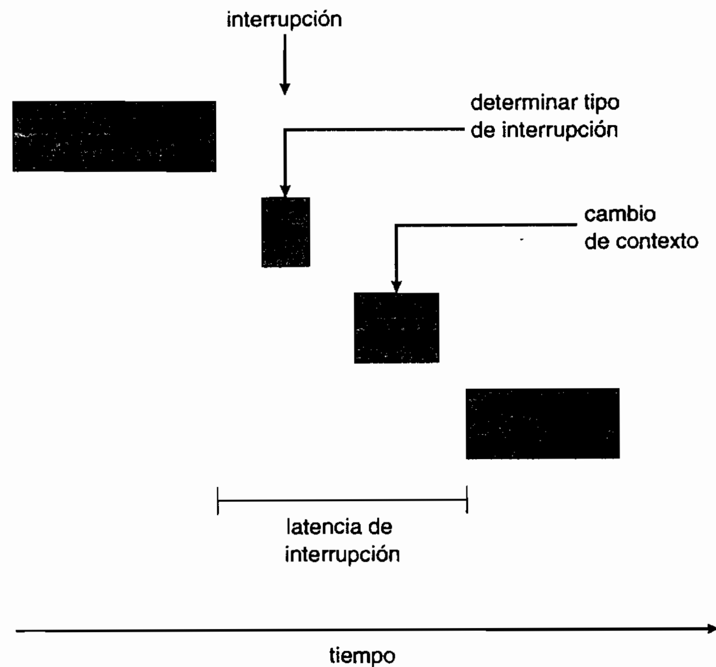


Figura 19.4 Latencia de interrupción.

realizar estas tareas es la latencia de interrupción (Figura 19.4). Obviamente, para los sistemas operativos de tiempo real resulta crucial minimizar la latencia de interrupción, con el fin de garantizar que las tareas de tiempo real reciban una atención inmediata.

Un factor importante que contribuye a la latencia de interrupción es la cantidad de tiempo que las interrupciones pueden estar desactivadas mientras se actualizan las estructuras de datos del *kernel*. Los sistemas operativos de tiempo real requieren que las interrupciones estén desactivadas durante períodos muy cortos de tiempo. Sin embargo, para los sistemas de tiempo real estrictos, no sólo es necesario minimizar la latencia de interrupción, sino que de hecho es preciso acotarla, con el fin de garantizar el comportamiento determinista que los *kernels* de tiempo real estricto deben exhibir.

La cantidad de tiempo requerida para que el despachador de planificación detenga un proceso e inicie otro se conoce como **latencia de despacho**. Para proporcionar a las tareas de tiempo real un acceso inmediato a la CPU, es obligatorio que el sistema operativo de tiempo real minimice esta latencia. La técnica más efectiva para tener una latencia de despacho bajo consiste en proporcionar un *kernel* apropiativo.

En la Figura 19.5 se muestra un diagrama que ilustra el concepto de latencia de despacho. La **fase de conflicto** de la latencia de despacho tiene dos componentes:

1. El desalojo de cualquier proceso que se esté ejecutando en el *kernel*.
2. La liberación por parte de los procesos de baja prioridad de los recursos necesarios para el proceso de alta prioridad.

Como ejemplo, en Solaris, la latencia de despacho con el mecanismo de apropiación desactivado es de más de 100 milisegundos. Si se activa el mecanismo de apropiación, esta latencia se reduce a menos de un milisegundo.

Una cuestión que puede afectar a la latencia de despacho es la que surge cuando un proceso de mayor prioridad necesita leer o modificar datos del *kernel* a los que esté actualmente accediendo un proceso de menor prioridad (o una cadena de procesos de menor prioridad). Puesto que los datos del *kernel* están normalmente protegidos mediante un cerrojo, el proceso de mayor prioridad tendrá que esperar a que otro de menor prioridad finalice con el recurso. Esta situación se complica aún más si el proceso de menor prioridad es desalojado en favor de otro proceso que

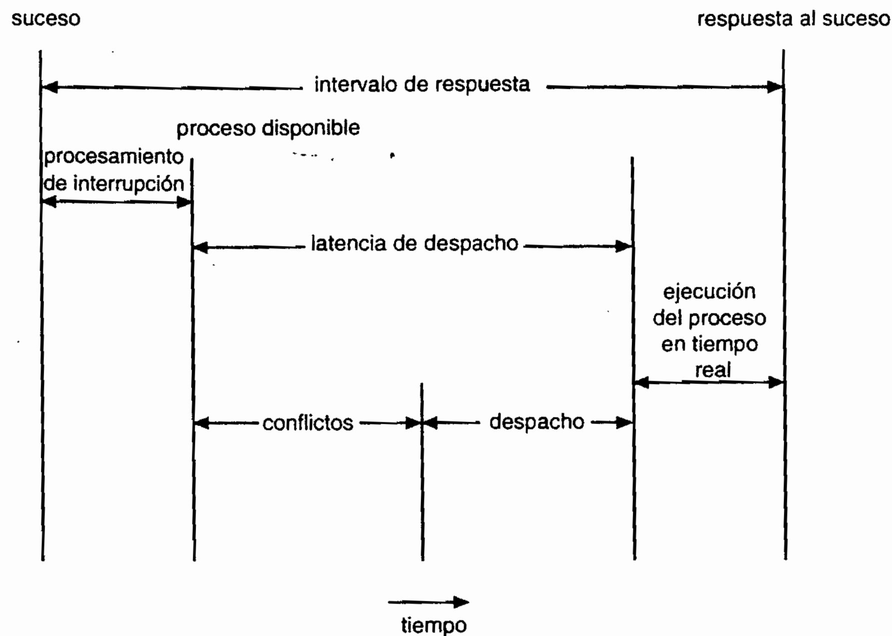


Figura 19.5 Latencia de despacho.

tenga una prioridad mayor. Como ejemplo, vamos a suponer que tenemos tres procesos *L*, *M* y *H*, cuyas prioridades siguen el orden $L < M < H$. Supongamos que el proceso *H* requiere el recurso *R*, al que está actualmente accediendo el proceso *L*. Normalmente, el proceso *H* esperaría a que *L* finalizará de utilizar el recurso *R*. Sin embargo, suponga ahora que el proceso *M* pasa a ser ejecutable desalojando así al proceso *L*. Indirectamente, un proceso con una menor prioridad (el proceso *M*) ha afectado al tiempo que el proceso *H* debe esperar para que *L* libere el recurso *R*.

Este problema, conocido como **inversión de prioridades**, puede resolverse utilizando el denominado **protocolo de herencia de prioridades**. De acuerdo con este protocolo, todos los procesos que estén accediendo a recursos que necesite otro proceso de mayor prioridad heredarán esa mayor prioridad hasta que hayan terminado con los recursos en cuestión. Una vez que han terminado, sus prioridades revierten a los valores originales. En el ejemplo anterior, un protocolo de herencia de prioridad permitiría al proceso *L* heredar temporalmente la prioridad del proceso *H*, evitando así que el proceso *M* se apropiara de la CPU. Cuando el proceso *L* hubiera terminado de utilizar el recurso *R*, renunciaría a su prioridad heredada de *H* y asumiría su prioridad original. Como el recurso *R* está ahora disponible, es el proceso *H* (y no *M*) el que se ejecutará a continuación.

19.5 Planificación de la CPU en tiempo real

Nuestro tratamiento de la planificación hasta ahora se ha centrado principalmente en los sistemas de tiempo no estrictos. Como hemos mencionado, sin embargo, el mecanismo de planificación para dichos sistemas no proporciona ninguna garantía sobre cuándo se planificará para ejecución un proceso crítico; el sistema tan sólo garantiza que se dará a ese proceso preferencia sobre otros procesos no críticos. Los sistemas de tiempo real estrictos tienen requisitos más fuertes: es necesario dar servicio a una tarea de acuerdo con el plazo prefijado; proporcionar ese servicio después de que pase el plazo equivale, a todos los efectos, a no proporcionar el servicio en absoluto.

Vamos a considerar ahora el tema de la planificación para los sistemas de tiempo real estrictos. Sin embargo, antes de continuar con los detalles de los algoritmos de planificación individuales, debemos definir ciertas características de los procesos que hay que planificar. En primer lugar, los procesos se consideran **periódicos**. Es decir, esos procesos requieren la CPU a intervalos constantes (períodos). Cada proceso periódico tiene un tiempo de procesamiento fijo *t* una vez que

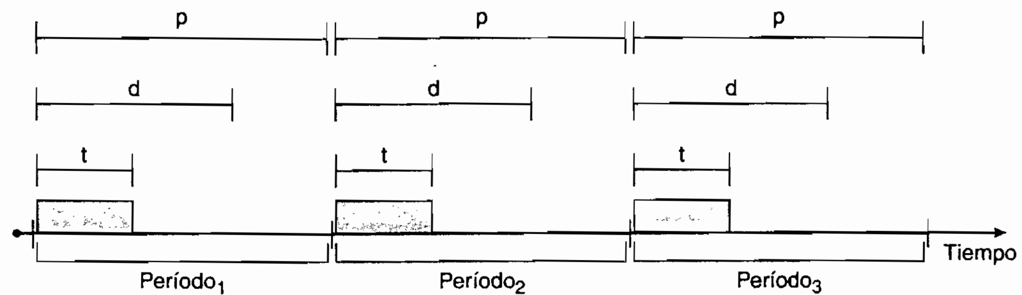


Figura 19.6 Tarea periódica.

adquiere la CPU, un plazo d antes del cual debe ser servido por la CPU y un período p. La relación del tiempo de procesamiento, el plazo y el período puede expresarse como $0 \leq t \leq d \leq p$. La **tasa** de una tarea periódica es $1/p$. La Figura 19.6 ilustra la ejecución de un proceso periódico a lo largo del tiempo. Los planificadores pueden aprovechar esta relación y asignar las prioridades de acuerdo con el plazo de un proceso periódico o con su tasa requerida.

Lo que resulta inusual acerca de esta forma de planificación es que un proceso puede tener que anunciar al planificador sus requisitos relativos al plazo de servicio. Entonces, utilizando una técnica conocida con el nombre de algoritmo de **control de admisión**, el planificador puede, o bien admitir el proceso, garantizando que se completará en el tiempo prefijado, o rechazar la solicitud como imposible, si no puede garantizar que se vaya a dar servicio a la tarea antes de alcanzar el plazo.

En las siguientes secciones, vamos a explorar los algoritmos de planificación dirigidos a tratar de satisfacer los requisitos relativos al plazo en los sistemas de tiempo real estrictos.

19.5.1 Planificación por prioridad monótona en tasa

El algoritmo de **planificación por prioridad monótona en tasa** (rate-monotonic scheduling) planifica las tareas periódicas utilizando una política de prioridad estática con apropiación. Si se está ejecutando un proceso de menor prioridad y otro proceso de mayor prioridad pasa a estar disponible para ejecución, este proceso desalojará al proceso de menor prioridad. Al entrar en el sistema, a cada tarea periódica se le asigna una prioridad que está en función inversa a su período: cuanto más corto sea el período, mayor será la prioridad y cuanto más largo sea el período, menor será la prioridad. La razón que subyace a esta política consiste en asignar una mayor prioridad a aquellas tareas que requieren la CPU más a menudo. Además, la planificación por prioridad monótona en tasa presupone que el tiempo de procesamiento de un proceso periódico es igual en cada ráfaga de la CPU, es decir, que cada vez que un proceso adquiere la CPU, la duración de su correspondiente ráfaga de CPU es la misma.

Vamos a considerar un ejemplo. Tenemos dos procesos P_1 y P_2. Los períodos para P_1 y P_2 son 50 y 100, respectivamente; es decir, $p_1 = 50$ y $p_2 = 100$. Los tiempos de procesamiento son $t_1 = 20$ para P_1 y $t_2 = 35$ para P_2. El plazo para cada proceso requiere que el proceso complete su ráfaga de CPU antes de que comience el siguiente período.

Primero debemos preguntarnos si es posible planificar estas tareas de manera que cada una de ellas satisfaga sus requisitos de plazo. Si medimos la utilización de CPU por parte de un proceso P_i como la relación entre su ráfaga de procesamiento y su período (t_i / p_i) la utilización de la CPU por parte de P_1 será 20 / 50 = 0,40 y la de P_2 será 35 / 100 = 0,35, dando una utilización total de la CPU del 75 por ciento. Por tanto, parece que sí que podemos planificar estas tareas de tal forma que ambas cumplan con sus plazos y sigan dejando a la CPU una serie de ciclos disponibles.

En primer lugar, vamos a suponer que asignamos a P_2 una mayor prioridad que a P_1. La ejecución de P_1 y P_2 se muestra en la Figura 19.7. Como podemos ver, P_2 comienza primero su ejecución y se completa en el instante 35. En este punto, comienza P_1, que completa su ráfaga de CPU en el instante 55. Sin embargo, el primer plazo para P_1 era en el instante 50, por lo que el planificador ha hecho que P_1 no sea capaz de completarse antes del plazo.

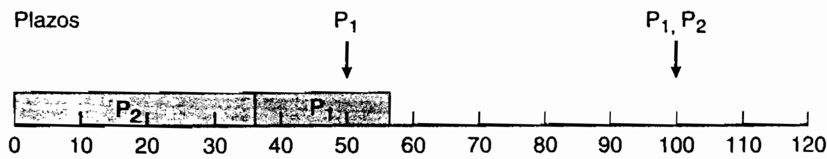


Figura 19.7 Planificación de tareas cuando P_2 tiene una prioridad mayor que P_1.

Ahora vamos a suponer que utilizamos una planificación por prioridad monótona en tasa, en la que asignamos a P_1 una mayor prioridad que a P_2, ya que el período de P_1 es más corto que el de P_2. La ejecución de estos procesos se muestra en la Figura 19.8. P_1 comienza primero y completa su ráfaga de CPU en el instante 20, cumpliendo así con su primer plazo. P_2 comienza a ejecutarse en este punto y se ejecuta hasta el instante 50. En este momento, es desalojado por P_1, aunque todavía faltan 5 milisegundos para completar su ráfaga. P_1 completa su ráfaga de CPU en el instante 70, momento en el que el planificador reanuda la ejecución de P_2. P_2 completa su ráfaga de CPU en el instante 75, cumpliendo también con su primer plazo. El sistema quedará inactivo hasta el instante 100, cuando se vuelva a planificar la ejecución de P_1.

La planificación por prioridad monótona en tasa se considera óptima, en el sentido de que si un conjunto procesos no puede planificarse utilizando este algoritmo, no podrá ser planificado por ningún otro algoritmo que asigne prioridades estáticas. Examinemos a continuación un conjunto de procesos que no pueden planificarse utilizando el algoritmo monótono con respecto a la tasa. Supongamos que el proceso P_1 tiene un período de $p_1 = 50$ y una ráfaga de CPU de $t_1 = 25$. Para P_2, los valores correspondientes son $p_2 = 80$ y $t_2 = 35$. La planificación por prioridad monótona en tasa asignaría al proceso P_1 una prioridad más alta, ya que es el proceso que tiene el período más corto. La utilización total de la CPU por parte de los dos procesos será $(25 / 50) + (35 / 80) = 0{,}94$ y, por tanto, parece lógico que pudieran planificarse los dos procesos y seguir dejando un 6 por ciento de tiempo de CPU disponible. El diagrama de Gantt que muestra la planificación de los procesos P_1 y P_2 es el que aparece en la Figura 19.9. Inicialmente, P_1 se ejecuta hasta que completa su ráfaga de CPU en el instante 25. Entonces empieza a ejecutarse el proceso P_2 y se ejecuta hasta el instante 50, siendo desalojado por el proceso P_1. En este punto, P_2 sigue necesitando 10 milisegundos para terminar su ráfaga de CPU. El proceso P_1 se ejecuta hasta el instante 75; sin embargo, P_2 no podrá completar su ráfaga de CPU antes de su plazo, en el instante 80.

Por tanto, a pesar de ser óptima, la planificación por prioridad monótona en tasa tiene una limitación: la utilización de la CPU está acotada y no siempre es posible maximizar por completo los recursos de CPU. El caso peor de utilización de la CPU para la planificación de N procesos es:

$$2\,(2^{1/n} - 1)$$

Con un proceso en el sistema, la utilización de la CPU es del 100 por ciento, pero esa tasa de utilización cae aproximadamente al 69 por ciento a medida que el número de procesos se aproxima a infinito. Con dos procesos, la tasa utilización de la CPU está acotada entorno al 83 por ciento. La utilización combinada de la CPU para los dos procesos planificados en las Figuras 19.7 y 19.8 es del 75 por ciento, por lo que está garantizado que el algoritmo de planificación por prioridad monótona en tasa puede planificarlos de modo que sean capaces de cumplir con sus plazos. Sin embargo, para los procesos planificados de la Figura 19.9, la tasa combinada de utilización de la CPU es de aproximadamente el 94 por ciento, por tanto, la planificación por prioridad monótona en tasa no puede garantizar que puedan planificarse esos procesos para cumplir con sus plazos.

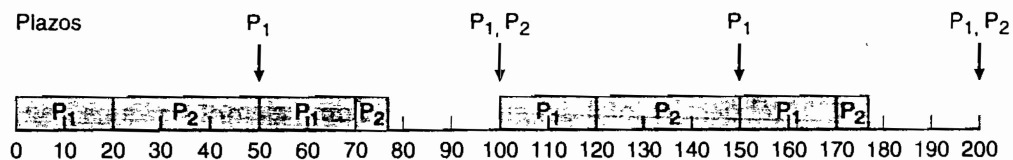


Figura 19.8 Planificación por prioridad monótona en tasa.

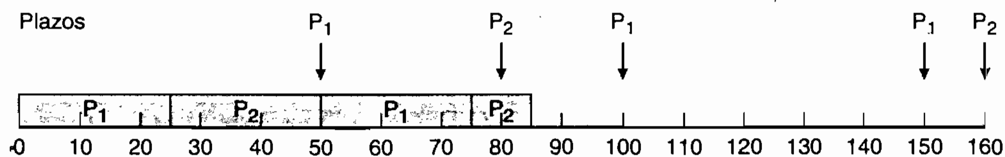


Figura 19.9 Problema de no cumplimiento de plazos con la planificación por prioridad monótona en tasa.

19.5.2 Planificación por prioridad en finalización de plazo

El algoritmo de planificación por prioridad en finalización de plazo (EDF, earliest-deadline-first) asigna dinámicamente las prioridades de acuerdo con la finalización de los plazos. Cuanto más próximo esté un plazo, mayor será la prioridad, y cuanto más lejano esté el plazo, menor será la prioridad. Con una política EDF, cuando un proceso pasa a ser ejecutable, debe anunciar sus requisitos de plazos al sistema. Puede que sea necesario ajustar las prioridades para reflejar el siguiente plazo de ese proceso que acaba de pasar a ser ejecutable. Observe que este algoritmo difiere del algoritmo por prioridad monótona en tasa en que las prioridades eran fijas.

Para ilustrar la planificación EDF, vamos a planificar de nuevo los procesos mostrados en la Figura 19.9, en la que no habíamos sido capaces de cumplir con los plazos empleando una planificación por prioridad monótona en tasa. Recuerde que P_1 tiene los valores $p_1 = 50$ y $t_1 = 25$ y que P_2 tiene los valores $p_2 = 80$ y $t_2 = 35$. La planificación EDF de estos procesos se muestra en la Figura 19.10. El proceso P_1 tiene su finalización de plazo, por lo que su prioridad inicial es más alta que la del proceso P_2. El proceso P_2 comienza a ejecutarse al final de la ráfaga de CPU de P_1. Sin embargo, mientras que el algoritmo de planificación por prioridad monótona en tasa permitía a P_1 desalojar a P_2 al principio de su siguiente período, en el instante 50, el algoritmo de planificación EDF permite que el proceso P_2 continúe ejecutándose. P_2 tiene ahora una mayor prioridad que P_1 porque su siguiente plazo (en el instante 80) está más próximo que el de P_1 (en el instante 100). De este modo, tanto P_1 como P_2 pueden cumplir con su primer plazo. El proceso P_1 comienza a ejecutarse de nuevo en el instante 60 y completa su segunda ráfaga de CPU en el instante 85, cumpliendo también con su segundo plazo en el instante 100. P_2 comenzará a ejecutarse en este punto, para ser desalojado por P_1 al comienzo de su siguiente período en el instante 100. P_2 es desalojado porque P_1 tiene una finalización de plazo anterior (instante 150) que P_2 (instante 160). En el instante 125, P_1 completa su ráfaga de CPU y P_2 reanuda su ejecución, finalizando en el instante 145 y cumpliendo también con su plazo. El sistema estará inactivo hasta el instante 150, en el que se planifica de nuevo la ejecución de P_1.

A diferencia del algoritmo por prioridad monótona en tasa, la planificación EDF no requiere que los procesos sean periódicos, ni tampoco que necesiten una cantidad constante de tiempo de CPU por cada ráfaga de ejecución. El único requisito es que cada proceso indique al planificador su próximo plazo en el momento de pasar a ser ejecutable. El atractivo de la planificación EDF es que resulta óptima desde el punto de vista teórico. En teoría, permite planificar los procesos de modo que cada uno de ellos pueda cumplir con sus requisitos relativos a los plazos de ejecución, y la utilización de la CPU será del 100 por ciento. Sin embargo, en la práctica, es imposible conseguir este nivel de utilización de la CPU debido al coste de los cambios de contexto entre un proceso y el coste del tratamiento de las interrupciones.

19.5.3 Planificación con cuota proporcional

El algoritmo de planificación con cuota proporcional funciona asignando T cuotas entre todas las aplicaciones. Cada aplicación puede recibir N cuotas de tiempo, garantizando así que la aplicación tenga un N/T del tiempo total del procesador. Como ejemplo, vamos a suponer que hay un total de $T = 100$ cuotas que hay que dividir entre tres procesos, A, B y C. Asignamos 50 cuotas a A, 15 cuotas a B y 20 cuotas a C. Este esquema asegura que A tendrá un 50 por ciento del tiempo total de procesador, B tendrá un 15 por ciento y C tendrá un 20 por ciento.

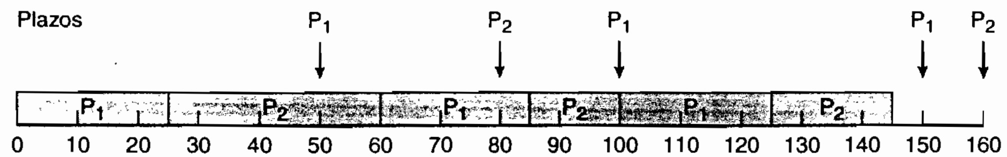


Figura 19.10 Planificación por prioridad en finalización de plazo.

El algoritmo de planificación con cuota proporcional debe trabajar en conjunción con una política de control de admisión, para garantizar que cada aplicación reciba su cuota de tiempo asignada. La política de control de admisión sólo admitirá a un cliente que solicite un número concreto de cuotas si hay suficientes cuotas disponibles. En nuestro ejemplo, hemos asignado 50 + 15 + 20 = 75 cuotas del total de 100. Si un nuevo proceso *D* solicitará 30 cuotas, el controlador de admisión denegaría a *D* la entrada en el sistema.

19.5.4 Planificación en Pthread

El estándar POSIX también proporciona extensiones para informática en tiempo real: POSIX.1b. En esta sección, vamos a analizar parte de la API Pthread de POSIX relacionada con la planificación de hebras en tiempo real. Pthreads define dos clases de planificación para las hebras en tiempo real:

- SCHED_FIFO
- SCHED_RR

SCHED_FIFO planifica las hebras de acuerdo con una política en que el primero en llegar es el primero en ser servido, usando una cola FIFO, como se indica en la Sección 5.3.1. Sin embargo, no existe ningún reparto de franjas temporales entre las hebras de igual prioridad. Por tanto, la hebra de tiempo real de prioridad más alta situada al principio de la cola FIFO recibirá la CPU y continuará ejecutándose hasta terminar, o hasta quedar bloqueada. SCHED_RR (donde RR quiere decir *round-robin*, es decir, planificación por turnos) es similar a SCHED_FIFO salvo porque incluye una distribución de franjas temporales entre hebras de igual prioridad. Pthreads proporciona una clase adicional de planificación (SCHED_OTHER) pero su implementación no está definida y es específica del sistema, pudiendo por tanto comportarse de manera diferente en distintos sistemas.

La API Pthread especifica las siguientes dos funciones para consultar y establecer la política de planificación:

- `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
- `pthread_attr_getsched_policy(pthread_attr_t *attr, int policy)`

El primer parámetro de ambas funciones es un puntero al conjunto de atributos de la hebra. El segundo parámetro puede ser un puntero a un entero que defina la política de planificación actual [para `pthread_attr_getsched_policy()`] o un valor entero (SCHED_FIFO, SCHED_RR o SCHED_OTHER) para la función `pthread_attr_getsched_policy()`. Ambas funciones devuelven valores distintos de cero en caso de que se produzca un error.

En la Figura 19.11 se muestra un programa Pthread que usa esta API. Este programa determina en primer lugar la política de planificación actual y después selecciona el algoritmo de planificación SCHED_OTHER.

19.6 VxWorks 5.x

En esta sección, vamos a describir VxWorks, un popular sistema operativo de tiempo real que proporciona soporte de tiempo real estricto. VxWorks, desarrollado comercialmente por Wind River Systems, se utiliza ampliamente en vehículos, dispositivos de consumo e industriales y en equipos de red como conmutadores y encaminadores. VxWorks también se emplea para controlar los dos robots, *Spirit* y *Opportunity*, que comenzaron a explorar el planeta Marte en 2004.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* obtener atributos predeterminados */
   pthread_attr_init(&attr);

   /* obtener la política de planificación actual */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Imposible obtener política.\n");
   else {
      if (policy == SCHED_OTHER)
        printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR)
        printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
        printf("SCHED_FIFO\n");
   }

   /* establecer la política de planificación - FIFO, RR, u OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)
     fprintf(stderr, "Imposible establecer política.\n");

   /* crear las hebras */
   for (i = 0; i < NUM_THREADS; i++)
     pthread_create(&tid[i],&attr,runner,NULL);

   /* terminar cada una de las hebras */
   for (i = 0; i < NUM_THREADS; i++)
     pthread_join(tid[i], NULL);
}

/* Cada hebra tomará el control en esta función */
void *runner(void *param)
{

   /* realizar alguna tarea ... */

   pthread exit(0);
}
```

Figura 19.11 API de planificación de Pthread.

La organización de VxWorks se muestra en la Figura 19.12. VxWorks está centrado en torno al *microkernel Wind*. Recuerde de nuestras explicaciones en la Sección 2.7.3, que los *microkernels* están diseñados de modo que el *kernel* del sistema operativo proporcione un número mínimo de características; las utilidades adicionales, como la conexión por red, los sistemas de archivos y los gráficos, se proporcionan en bibliotecas situadas fuera del *kernel*. Esta técnica ofrece muchas ventajas

http://libreria-universitaria.blogspot.com

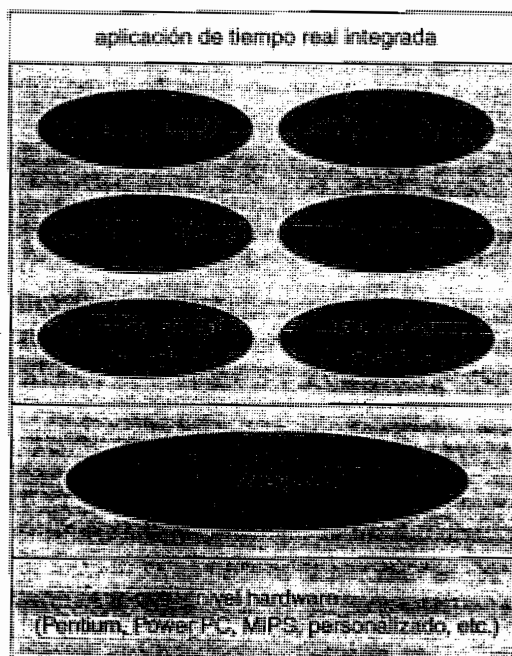


Figura 19.12 La organización de VxWorks.

incluyendo la minimización del tamaño del *kernel*, que constituye una característica deseable para todo sistema integrado que requiera una huella de memoria pequeña.

El *microkernel* Wind soporta las siguientes características básicas:

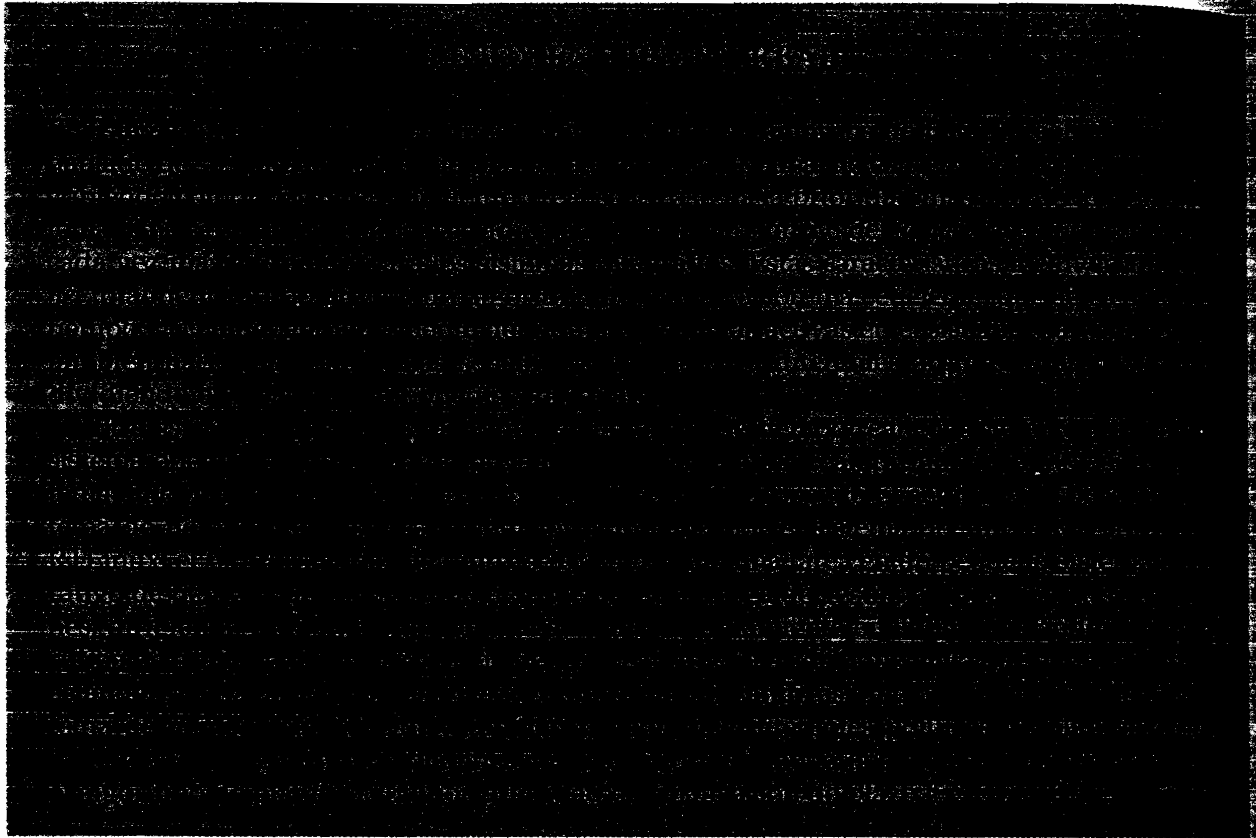
- **Procesos.** El *microkernel* Wind proporciona soporte para procesos individuales y hebras (utilizando la API Pthread). Sin embargo, de forma similar a Linux, VxWorks no distingue entre procesos y hebras haciendo referencia a ambos con el nombre de **tareas**.

- **Planificación.** Wind proporciona dos modelos separados de planificación: planificación por turnos apropiativa y no apropiativa con 256 niveles diferentes de prioridad. El planificador también soporta la API POSIX para hebras de tiempo real que se analiza en la Sección 19.5.4.

- **Interrupciones.** El *microkernel* Wind también gestiona interrupciones. Para cumplir con los requisitos de tiempo real estrictos, los tiempos de latencia de despacho y de interrupción están acotados.

- **Comunicación interprocesos.** El *microkernel* Wind proporciona mecanismos tanto de memoria compartida como de paso de mensajes para la comunicación entre las distintas tareas. Wind también permite que las tareas se comuniquen utilizando una técnica denominada ***pipes***, que son un mecanismo que se comporta del mismo modo que una cola FIFO, pero permite a las tareas comunicarse en un archivo especial, el *pipe*. Para proteger los datos compartidos por las distintas tareas, VxWorks proporciona semáforos y cerrojos mútex, con un protocolo de herencia de prioridades con el fin de evitar el fenómeno de inversión de prioridad.

Fuera del *microkernel*, VxWorks incluye varias bibliotecas de componentes que proporcionan soporte para POSIX, Java, redes TCP/IP, etc. Todos los componentes son opcionales, permitiendo al diseñador de un sistema integrado personalizar el sistema de acuerdo con sus necesidades específicas. Por ejemplo, si no se requiere la conexión por red, puede excluirse la biblioteca TCP/IP de la imagen del sistema operativo. Esta estrategia permite al diseñador del sistema operativo incluir sólo las características requeridas, minimizando así el tamaño (o huella de memoria) del sistema operativo.



VxWorks adopta un enfoque interesante en lo que respecta a la gestión de memoria, permitiendo dos niveles de memoria virtual. El primer nivel, que es bastante simple, permite controlar la caché por separado para cada página. Esta política hace que las aplicaciones puedan especificar ciertas páginas como no almacenables en caché. Cuando hay una serie de datos que están siendo compartidos por distintas tareas que se ejecutan en una arquitectura multiprocesador, resulta posible almacenar los datos compartidos en cachés separadas que sean locales a cada procesador individual. A menos que una arquitectura soporte una política de coherencia de caché para garantizar que los mismos datos que residan en dos cachés no puedan ser distintos, dichos datos compartidos no deben almacenarse en caché y deben residir, en su lugar, únicamente en memoria principal, para que todas las tareas puedan mantener una vista coherente de los datos.

El segundo nivel de memoria virtual requiere el componente opcional de memoria virtual VxVMI (Figura 19.12), junto con un soporte de una unidad de gestión de memoria (MMU) por parte del procesador. Cargando este componente opcional en los sistemas con una MMU, VxWorks permite a las tareas marcar ciertas áreas de datos como *privadas*. Un área de datos marcada como privada sólo podrá ser accedida por la tarea a la que pertenezca. Además, VxWorks permite declarar como de sólo lectura las páginas que contienen el código del *kernel* junto con el vector de interrupción. Esta característica resulta muy útil, ya que VxWorks no distingue entre los modos de usuario y de *kernel*; todas las aplicaciones se ejecutan en modo *kernel*, lo que proporciona a la aplicación acceso al espacio de direcciones completo del sistema.

19.7 Resumen

Un sistema de tiempo real es un sistema informático que requiere que se obtengan los resultados antes de un cierto plazo; los resultados obtenidos después cumplirse el plazo son completamente inútiles. En los dispositivos de consumo e industriales están integrados muchos sistemas de tiempo real. Existen dos tipos de sistema de tiempo real: estrictos y no estrictos. Los sistemas de tiempo real no estrictos son los menos restrictivos, limitándose a asignar a las tareas de tiempo

real una prioridad de planificación mayor que a las otras tareas. Los sistemas de tiempo real estrictos deben garantizar que se preste servicio a las tareas de tiempo real dentro de los plazos definidos. Además de por los requisitos de temporización estrictos, los sistemas de tiempo pueden caracterizarse también por el hecho de que se utilizan para un único propósito y se ejecutan sobre dispositivos de pequeño tamaño y bajo coste.

Para cumplir con los requisitos de temporización, los sistemas operativos de tiempo real deben emplear diversas técnicas. El planificador para un sistema de tiempo real debe poder trabajar con un algoritmo basado en prioridades con apropiación. Además, el sistema operativo debe permitir que las tareas que se ejecutan en el *kernel* sean desalojadas en favor de las tareas de tiempo real con mayor prioridad. Los sistemas operativos de tiempo real también tratan de resolver los problemas específicos de temporización minimizando la latencia de interrupción como la de despacho.

Entre los algoritmos de planificación de tiempo real podemos citar el algoritmo de planificación por prioridad monótona en tasa y el algoritmo por prioridad en finalización de plazo. La planificación por prioridad monótona en tasa asigna una mayor prioridad a las tareas que requieren la CPU más a menudo, asignando la prioridad más baja a aquellas que hacen un uso más infrecuente de la CPU. La planificación por prioridad en finalización de plazo asigna la prioridad de acuerdo con los plazos previstos: cuanto más próximo esté un plazo, mayor será la prioridad del proceso correspondiente. La planificación con cuota proporcional utiliza la técnica de dividir el tiempo de procesador en una serie cuotas y asignar un número de cuotas a cada proceso, garantizando así a cada proceso su cuota proporcional de tiempo de CPU. La API Pthread proporciona también diversas funciones para la planificación de hebras en tiempo real.

Ejercicios

19.1 Indique si resulta más apropiada la planificación de tiempo real estricta o no estricta en los siguientes entornos:

a. Termostato en una vivienda.

b. Sistema de control para una central de energía nuclear.

c. Sistema de ahorro en un vehículo.

d. Sistema de aterrizaje en un avión comercial.

19.2 Indique de qué manera se podría resolver el problema de inversión de prioridades en un sistema de tiempo real. Indique también si las soluciones podrían implementarse dentro del contexto de un planificador con cuota proporcional.

19.3 El *kernel* de Linux 2.6 puede construirse sin sistema de memoria virtual. Explique por qué esta característica puede resultar atractiva para los diseñadores de sistemas en tiempo real.

19.4 ¿En qué circunstancias es la planificación por prioridad monótona en tasa peor que la planificación por prioridad de finalización de plazo, a la hora de satisfacer los plazos asociados con cada proceso?

19.5 Considere dos procesos, P_1 y P_2, donde $p_1 = 50$, $t_1 = 25$, $p_2 = 75$ y $t_2 = 30$.

a. ¿Pueden planificarse estos dos procesos utilizando un algoritmo de planificación monótona en tasa? Ilustre su respuesta utilizando un diagrama de Gantt.

b. Ilustre la planificación de estos dos procesos utilizando el algoritmo EDF de planificación por prioridad en finalización de plazo.

19.6 ¿Cuáles son los diversos componentes de las latencias de interrupción y de despacho?

19.7 Explique por qué las latencias de interrupción y de despacho deben estar acotadas en un sistema de tiempo real estricto.

Notas bibliográficas

Los algoritmos de planificación para los sistemas de tiempo real estrictos, como la planificación monótona en tasa y la planificación por prioridad en finalización de plazo, fueron presentados en Liu y Layland [1973]. Otros algoritmos de planificación, asi como una serie de extensiones a los algoritmos anteriores se presentan en Jensen et al. [1985], Lehoczky et al. [1989], Audsley et al. [1991], Mok [1983] y Stoica et al. [1996]. Mok [1983] describía un algoritmo de asignación dinámica de prioridades denominado planificación con prioridad de la menor laxitud. Stoica et al [1996] analiza el algoritmo de cuota proporcional. Puede encontrar información útil relativa a diversos sistemas operativos populares que se utilizan en sistemas integrados en *http://rtlinux.org*, *http://windriver.com* y *http://qnx.com*. El tema de las líneas futuras de investigación y los problemas más importantes abordados por esas investigaciones en el campo de los sistemas integrados se analiza en un artículo de Stankovic [1996].