CAPÍTULO 3

Procesos

Los primeros sistemas informáticos sólo permitían que se ejecutara un programa a la vez. Este programa tenía el control completo del sistema y tenía acceso a todos los recursos del mismo. Por el contrario, los sistemas informáticos actuales permiten que se carguen en memoria múltiples programas y se ejecuten concurrentemente. Esta evolución requiere un mayor control y aislamiento de los distintos programas y estas necesidades dieron lugar al concepto de **proceso**, que es un programa en ejecución. Un proceso es la unidad de trabajo en los sistemas modernos de tiempo compartido.

Cuanto más complejo es el sistema operativo, más se espera que haga en nombre de sus usuarios. Aunque su principal cometido es ejecutar programas de usuario, también tiene que ocuparse de diversas tareas del sistema que, por uno u otro motivo, no están incluidas dentro del *kernel*. Por tanto, un sistema está formado por una colección de procesos: procesos del sistema operativo que ejecutan código del sistema y procesos de usuario que ejecutan código de usuario. Potencialmente, todos estos procesos pueden ejecutarse concurrentemente, multiplexando la CPU (o las distintas CPU) entre ellos. Cambiando la asignación de la CPU entre los distintos procesos, el sistema operativo puede incrementar la productividad de la computadora.

OBJETIVOS DEL CAPÍTULO

- Presentar el concepto de proceso (un programa en ejecución), en el que se basa todo el funcionamiento de un sistema informático.
- Describir los diversos mecanismos relacionados con los procesos, incluyendo los de planificación, creación y finalización de procesos, y los mecanismos de comunicación.
- Describir los mecanismos de comunicación en los sistemas cliente-servidor.

3.1 Concepto de proceso

Una pregunta que surge cuando se estudian los sistemas operativos es cómo llamar a las diversas actividades de la CPU. Los sistemas de procesamiento por lotes ejecutan *trabajos*, mientras que un sistema de tiempo compartido tiene *programas de usuario* o *tareas*. Incluso en un sistema monousuario, como Microsoft Windows, el usuario puede ejecutar varios programas al mismo tiempo: un procesador de textos, un explorador web y un programa de correo electrónico. Incluso aunque el usuario pueda ejecutar sólo un programa cada vez, el sistema operativo puede tener que dar soporte a sus propias actividades internas programadas, como los mecanismos de gestión de la memoria. En muchos aspectos, todas estas actividades son similares, por lo que a todas ellas las denominamos *procesos*.

En este texto, los términos *trabajo* y *proceso* se usan indistintamente. Aunque personalmente preferimos el término *proceso*, gran parte de la teoría y terminología de los sistemas operativos se

desarrolló durante una época en que la principal actividad de los sistemas operativos era el procesamiento de trabajos por lotes. Podría resultar confuso, por tanto, evitar la utilización de aquellos términos comúnmente aceptados que incluyen la palabra *trabajo* (como por ejemplo *planificación de trabajos*) simplemente porque el término *proceso* haya sustituido a *trabajo*.

3.1.1 El proceso

Informalmente, como hemos indicado antes, un proceso es un programa en ejecución. Hay que resaltar que un proceso es algo más que el código de un programa (al que en ocasiones se denomina **sección de texto**). Además del código, un proceso incluye también la actividad actual, que queda representada por el valor del **contador de programa** y por los contenidos de los registros del procesador. Generalmente, un proceso incluye también la **pila** del proceso, que contiene datos temporales (como los parámetros de las funciones, las direcciones de retorno y las variables locales), y una **sección de datos**, que contiene las variables globales. El proceso puede incluir, asimismo, un **cúmulo de memoria**, que es la memoria que se asigna dinámicamente al proceso en tiempo de ejecución. En la Figura 3.1 se muestra la estructura de un proceso en memoria.

Insistamos en que un programa, por sí mismo, no es un proceso; un programa es una entidad *pasiva*, un archivo que contiene una lista de instrucciones almacenadas en disco (a menudo denominado **archivo ejecutable**), mientras que un proceso es una entidad *activa*, con un contador de programa que especifica la siguiente instrucción que hay que ejecutar y un conjunto de recursos asociados. Un programa se convierte en un proceso cuando se carga en memoria un archivo ejecutable. Dos técnicas habituales para cargar archivos ejecutables son: hacer doble clic sobre un icono que represente el archivo ejecutable e introducir el nombre del archivo ejecutable en la línea de comandos (como por ejemplo, `prog.exe` o `a.out`.)

Aunque puede haber dos procesos asociados con el mismo programa, esos procesos se consideran dos secuencias de ejecución separadas. Por ejemplo, varios usuarios pueden estar ejecutando copias diferentes del programa de correo, o el mismo usuario puede invocar muchas copias del explorador web. Cada una de estas copias es un proceso distinto y, aunque las secciones de texto sean equivalentes, las secciones de datos, del cúmulo (*heap*) de memoria y de la pila variarán de unos procesos a otros. También es habitual que un proceso cree muchos otros procesos a medida que se ejecuta. En la Sección 3.4 se explican estas cuestiones.

3.1.2 Estado del proceso

A medida que se ejecuta un proceso, el proceso va cambiando de **estado**. El estado de un proceso se define, en parte, según la actividad actual de dicho proceso. Cada proceso puede estar en uno de los estados siguientes:

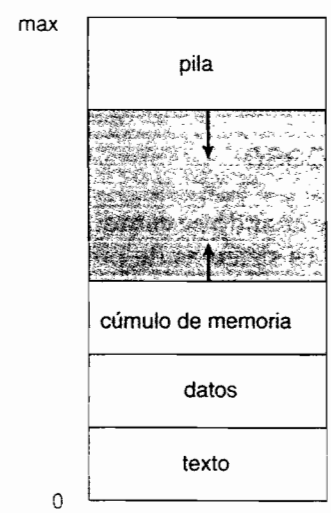

Figura 3.1 Proceso en memoria.

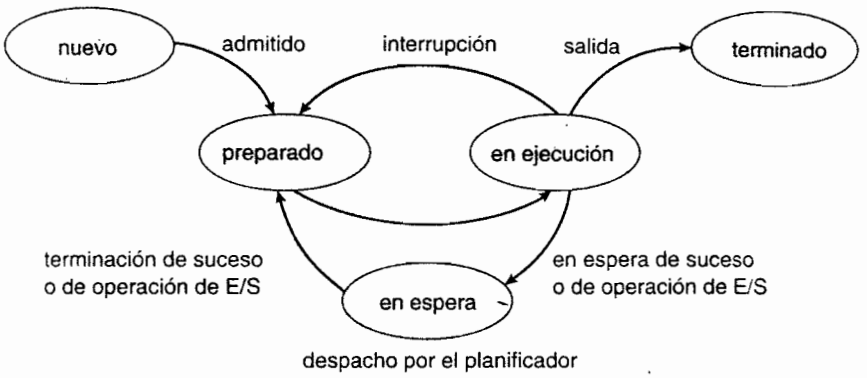

Figura 3.2 Diagrama de estados de un proceso.

- **Nuevo**. El proceso está siendo creado.
- **En ejecución**. Se están ejecutando las instrucciones.
- **En espera**. El proceso está esperando a que se produzca un suceso (como la terminación de una operación de E/S o la recepción de una señal).
- **Preparado**. El proceso está a la espera de que le asignen a un procesador.
- **Terminado**. Ha terminado la ejecución del proceso.

Estos nombres son arbitrarios y varían de un sistema operativo a otro. Sin embargo, los estados que representan se encuentran en todos los sistemas. Determinados sistemas operativos definen los estados de los procesos de forma más específica. Es importante darse cuenta de que sólo puede haber un proceso *ejecutándose* en cualquier procesador en cada instante concreto. Sin embargo, puede haber muchos procesos *preparados* y *en espera*. En la Figura 3.2 se muestra el diagrama de estados de un proceso genérico.

3.1.3 Bloque de control de proceso

Cada proceso se representa en el sistema operativo mediante un **bloque de control de proceso** (PCB, process control block), también denominado *bloque de control de tarea* (véase la Figura 3.3). Un bloque de control de proceso contiene muchos elementos de información asociados con un proceso específico, entre los que se incluyen:

- **Estado del proceso**. El estado puede ser: nuevo, preparado, en ejecución, en espera, detenido, etc.
- **Contador de programa**. El contador indica la dirección de la siguiente instrucción que va a ejecutar dicho proceso.
- **Registros de la CPU**. Los registros varían en cuanto a número y tipo, dependiendo de la arquitectura de la computadora. Incluyen los acumuladores, registros de índice, punteros de pila y registros de propósito general, además de toda la información de los indicadores de estado. Esta información de estado debe guardarse junto con el contador de programa cuando se produce una interrupción, para que luego el proceso pueda continuar ejecutándose correctamente (Figura 3.4).
- **Información de planificación de la CPU**. Esta información incluye la prioridad del proceso, los punteros a las colas de planificación y cualesquiera otros parámetros de planificación que se requieran. El Capítulo 5 describe los mecanismos de planificación de procesos.
- **Información de gestión de memoria**. Incluye información acerca del valor de los registros base y límite, las tablas de páginas o las tablas de segmentos, dependiendo del mecanismo de gestión de memoria utilizado por el sistema operativo (Capítulo 8).

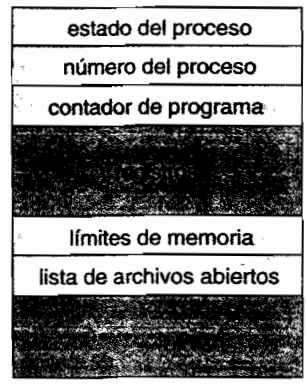

Figura 3.3 Bloque de control de proceso (PCB).

- **Información contable.** Esta información incluye la cantidad de CPU y de tiempo real empleados, los límites de tiempo asignados, los números de cuenta, el número de trabajo o de proceso, etc.
- **Información del estado de E/S.** Esta información incluye la lista de los dispositivos de E/S asignados al proceso, una lista de los archivos abiertos, etc.

En resumen, el PCB sirve simplemente como repositorio de cualquier información que pueda variar de un proceso a otro.

3.1.4 Hebras

El modelo de proceso que hemos visto hasta ahora implicaba que un proceso es un programa que tiene una sola **hebra** de ejecución. Por ejemplo, cuando un proceso está ejecutando un procesador de textos, se ejecuta una sola hebra de instrucciones. Esta única hebra de control permite al proceso realizar sólo una tarea cada vez. Por ejemplo, el usuario no puede escribir simultáneamente

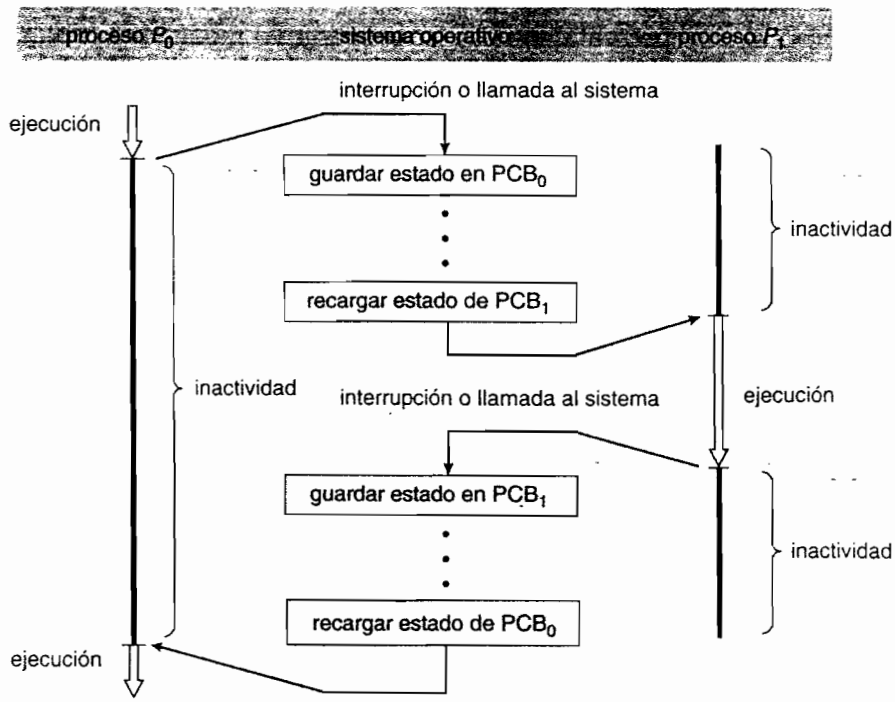

Figura 3.4 Diagrama que muestra la conmutación de la CPU de un proceso a otro.

caracteres y pasar el corrector ortográfico dentro del mismo proceso. Muchos sistemas operativos modernos han ampliado el concepto de proceso para permitir que un proceso tenga múltiples hebras de ejecución y, por tanto, pueda llevar a cabo más de una tarea al mismo tiempo. El Capítulo 4 se ocupa en detalle del análisis de los procesos multihebra.

3.2 Planificación de procesos

El objetivo de la multiprogramación es tener en ejecución varios procesos al mismo tiempo con el fin de maximizar la utilización de la CPU. El objetivo de los sistemas de tiempo compartido es conmutar la CPU entre los distintos procesos con tanta frecuencia que los usuarios puedan interactuar con cada programa mientras éste se ejecuta. Para conseguir estos objetivos, el **planificador de pro-**

REPRESENTACIÓN DE UN PROCESO EN LINUX

El bloque de control de un proceso en el sistema operativo Linux se representa mediante la estructura C `task_struct`. Esta estructura contiene toda la información necesaria para representar un proceso, incluyendo el estado del proceso, la información de planificación y de gestión de memoria, la lista de archivos abiertos y los punteros al padre del proceso y a cualquiera de sus hijos.

```
pid_t pid; /* identificador del proceso */
long state; /* estado del proceso */
unsigned int time_slice /* información de planificación */
struct task_struct *parent; /* padre del proceso */
struct list_head children; /* hijos del proceso */
struct files_struct *files; /* lista de archivos abiertos */
struct mm_struct *mm; /* espacio de direcciones de este proceso */
```

En esta estructura, por ejemplo, el estado de un proceso se representa mediante el campo `long state`. Dentro del *kernel* de Linux, todos los procesos activos se representan mediante una lista doblemente enlazada de objetos `task_struct`, y el *kernel* mantiene un puntero al proceso que se esté ejecutando en el sistema en ese momento. Esto se muestra en la Figura 3.5.

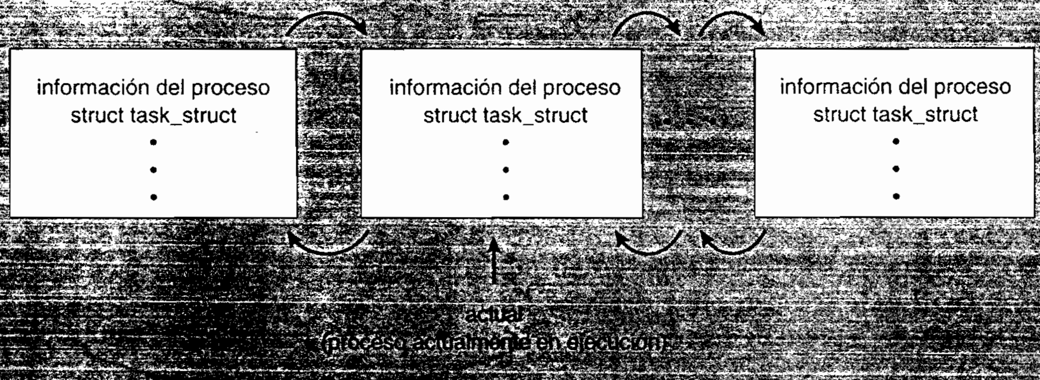

Figura 3.5 Procesos activos en Linux.

Como ilustración del modo en que el *kernel* manipula uno de los campos de `task_struct` para un proceso determinado, vamos a suponer que el sistema desea cambiar el estado del proceso actualmente en ejecución al valor `new_state`. Si `current` es un puntero al proce-

cesos selecciona un proceso disponible (posiblemente de entre un conjunto de varios procesos disponibles) para ejecutar el programa en la CPU. En los sistemas de un solo procesador, nunca habrá más de un proceso en ejecución: si hay más procesos, tendrán que esperar hasta que la CPU esté libre y se pueda asignar a otro proceso.

3.2.1 Colas de planificación

A medida que los procesos entran en el sistema, se colocan en una **cola de trabajos** que contiene todos los procesos del sistema. Los procesos que residen en la memoria principal y están preparados y en espera de ejecutarse se mantienen en una lista denominada **cola de procesos preparados**. Generalmente, esta cola se almacena en forma de lista enlazada. La cabecera de la cola de procesos preparados contiene punteros al primer y último bloques de control de procesos (PCB) de la lista. Cada PCB incluye un campo de puntero que apunta al siguiente PCB de la cola de procesos preparados.

El sistema también incluye otras colas. Cuando se asigna la CPU a un proceso, éste se ejecuta durante un rato y finalmente termina, es interrumpido o espera a que se produzca un determinado suceso, como la terminación de una solicitud de E/S. Suponga que el proceso hace una solicitud de E/S a un dispositivo compartido, como por ejemplo un disco. Dado que hay muchos procesos en el sistema, el disco puede estar ocupado con la solicitud de E/S de algún otro proceso. Por tanto, nuestro proceso puede tener que esperar para poder acceder al disco. La lista de procesos en espera de un determinado dispositivo de E/S se denomina **cola del dispositivo**. Cada dispositivo tiene su propia cola (Figura 3.6).

Una representación que habitualmente se emplea para explicar la planificación de procesos es el **diagrama de colas**, como el mostrado en la Figura 3.7, donde cada rectángulo representa una cola. Hay dos tipos de colas: la cola de procesos preparados y un conjunto de colas de dispositivo. Los círculos representan los recursos que dan servicio a las colas y las flechas indican el flujo de procesos en el sistema.

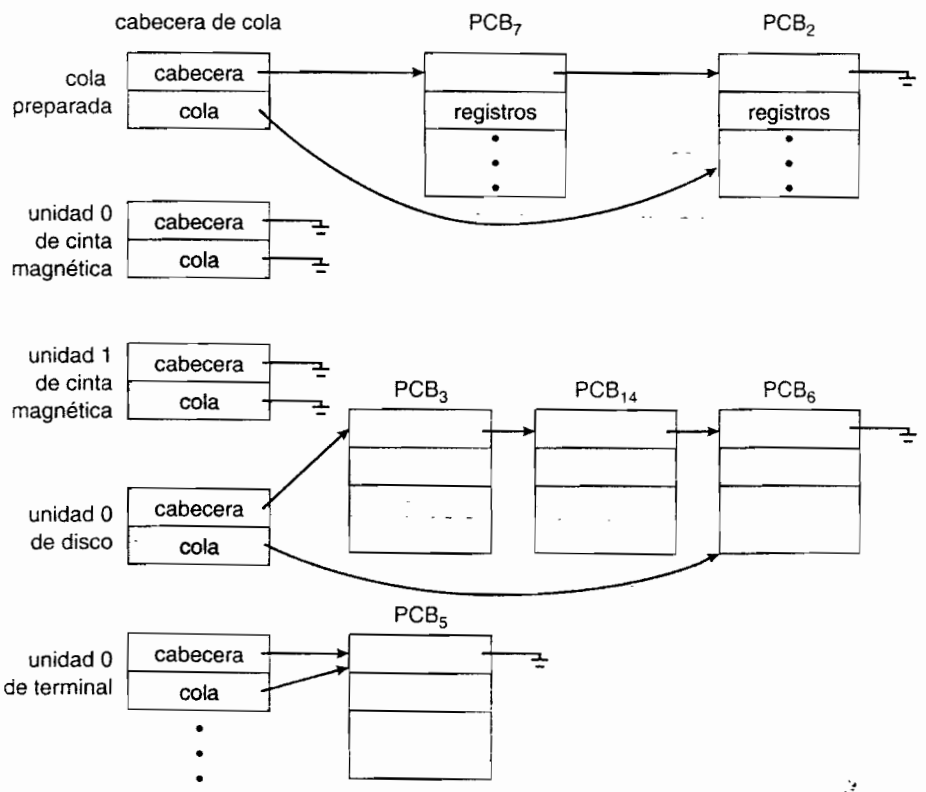

Figura 3.6 Cola de procesos preparados y diversas colas de dispositivos de E/S.

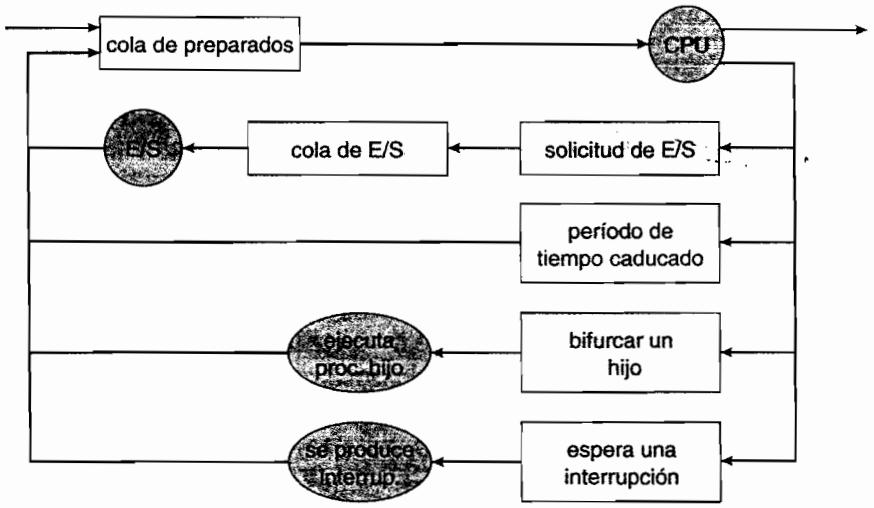

Figura 3.7 Diagrama de colas para la planificación de procesos.

Cada proceso nuevo se coloca inicialmente en la cola de procesos preparados, donde espera hasta que es seleccionado para ejecución, es decir, hasta que es **despachado**. Una vez que se asigna la CPU al proceso y éste comienza a ejecutarse, se puede producir uno de los sucesos siguientes:

- El proceso podría ejecutar una solicitud de E/S y ser colocado, como consecuencia, en una cola de E/S.
- El proceso podría crear un nuevo subproceso y esperar a que éste termine.
- El proceso podría ser desalojado de la CPU como resultado de una interrupción y puesto de nuevo en la cola de procesos preparados.

En los dos primeros casos, el proceso terminará, antes o después, por cambiar del estado de espera al estado preparado y será colocado de nuevo en la cola de procesos preparados. Los procesos siguen este ciclo hasta que termina su ejecución, momento en el que se elimina el proceso de todas las colas y se desasignan su PCB y sus recursos.

3.2.2 Planificadores

Durante su tiempo de vida, los procesos se mueven entre las diversas colas de planificación. El sistema operativo, como parte de la tarea de planificación, debe seleccionar de alguna manera los procesos que se encuentran en estas colas. El proceso de selección se realiza mediante un **planificador** apropiado.

A menudo, en un sistema de procesamiento por lotes, se envían más procesos de los que pueden ser ejecutados de forma inmediata. Estos procesos se guardan en cola en un dispositivo de almacenamiento masivo (normalmente, un disco), donde se mantienen para su posterior ejecución. El **planificador a largo plazo** o **planificador de trabajos** selecciona procesos de esta cola y los carga en memoria para su ejecución. El **planificador a corto plazo** o **planificador de la CPU** selecciona de entre los procesos que ya están preparados para ser ejecutados y asigna la CPU a uno de ellos.

La principal diferencia entre estos dos planificadores se encuentra en la frecuencia de ejecución. El planificador a corto plazo debe seleccionar un nuevo proceso para la CPU frecuentemente. Un proceso puede ejecutarse sólo durante unos pocos milisegundos antes de tener que esperar por una solicitud de E/S. Normalmente, el planificador a corto plazo se ejecuta al menos una vez cada 100 milisegundos. Debido al poco tiempo que hay entre ejecuciones, el planificador a corto plazo debe ser rápido. Si tarda 10 milisegundos en decidir ejecutar un proceso durante 100 milisegundos, entonces el $10/(100+10) = 9$ por ciento del tiempo de CPU se está usando (perdiendo) simplemente para planificar el trabajo.

El planificador a largo plazo se ejecuta mucho menos frecuentemente; pueden pasar minutos entre la creación de un nuevo proceso y el siguiente. El planificador a largo plazo controla el **grado de multiprogramación** (el número de procesos en memoria). Si el grado de multiprogramación es estable, entonces la tasa promedio de creación de procesos debe ser igual a la tasa promedio de salida de procesos del sistema. Por tanto, el planificador a largo plazo puede tener que invocarse sólo cuando un proceso abandona el sistema. Puesto que el intervalo entre ejecuciones es más largo, el planificador a largo plazo puede permitirse emplear más tiempo en decidir qué proceso debe seleccionarse para ser ejecutado.

Es importante que el planificador a largo plazo haga una elección cuidadosa. En general, la mayoría de los procesos pueden describirse como limitados por la E/S o limitados por la CPU. Un **proceso limitado por E/S** es aquel que invierte la mayor parte de su tiempo en operaciones de E/S en lugar de en realizar cálculos. Por el contrario, un **proceso limitado por la CPU** genera solicitudes de E/S con poca frecuencia, usando la mayor parte de su tiempo en realizar cálculos. Es importante que el planificador a largo plazo seleccione una adecuada **mezcla de procesos**, equilibrando los procesos limitados por E/S y los procesos limitados por la CPU. Si todos los procesos son limitados por la E/S, la cola de procesos preparados casi siempre estará vacía y el planificador a corto plazo tendrá poco que hacer. Si todos los procesos son limitados por la CPU, la cola de espera de E/S casi siempre estará vacía, los dispositivos apenas se usarán, y de nuevo el sistema se desequilibrará. Para obtener un mejor rendimiento, el sistema dispondrá entonces de una combinación equilibrada de procesos limitados por la CPU y de procesos limitados por E/S.

En algunos sistemas, el planificador a largo plazo puede no existir o ser mínimo. Por ejemplo, los sistemas de tiempo compartido, tales como UNIX y los sistemas Microsoft Windows, a menudo no disponen de planificador a largo plazo, sino que simplemente ponen todos los procesos nuevos en memoria para que los gestione el planificador a corto plazo. La estabilidad de estos sistemas depende bien de una limitación física (tal como el número de terminales disponibles), bien de la propia naturaleza autoajustable de las personas que utilizan el sistema. Si el rendimiento desciende a niveles inaceptables en un sistema multiusuario, algunos usuarios simplemente lo abandonarán.

Algunos sistemas operativos, como los sistemas de tiempo compartido, pueden introducir un nivel intermedio adicional de planificación; en la Figura 3.8 se muestra este planificador. La idea clave subyacente a un planificador a medio plazo es que, en ocasiones, puede ser ventajoso eliminar procesos de la memoria (con lo que dejan de contender por la CPU) y reducir así el grado de multiprogramación. Después, el proceso puede volver a cargarse en memoria, continuando su ejecución en el punto en que se interrumpió. Este esquema se denomina **intercambio**. El planificador a medio plazo descarga y luego vuelve a cargar el proceso. El intercambio puede ser necesario para mejorar la mezcla de procesos o porque un cambio en los requisitos de memoria haya hecho que se sobrepase la memoria disponible, requiriendo que se libere memoria. En el Capítulo 8 se estudian los mecanismos de intercambio.

3.2.3 Cambio de contexto

Como se ha mencionado en la Sección 1.2.1, las interrupciones hacen que el sistema operativo obligue a la CPU a abandonar su tarea actual, para ejecutar una rutina del *kernel*. Estos sucesos se producen con frecuencia en los sistemas de propósito general. Cuando se produce una interrupción el sistema tiene que guardar el **contexto** actual del proceso que se está ejecutando en la CPU, de modo que pueda restaurar dicho contexto cuando su procesamiento concluya, suspendiendo el proceso y reanudándolo después. El contexto se almacena en el PCB del proceso e incluye el valor de los registros de la CPU, el estado del proceso (véase la Figura 3.2) y la información de gestión de memoria. Es decir, realizamos una **salvaguarda del estado** actual de la CPU, en modo *kernel* o en modo usuario, y una **restauración del estado** para reanudar las operaciones.

La conmutación de la CPU a otro proceso requiere una salvaguarda del estado del proceso actual y una restauración del estado de otro proceso diferente. Esta tarea se conoce como **cambio de contexto**. Cuando se produce un cambio de contexto, el *kernel* guarda el contexto del proceso antiguo en su PCB y carga el contexto almacenado del nuevo proceso que se ha decidido ejecutar

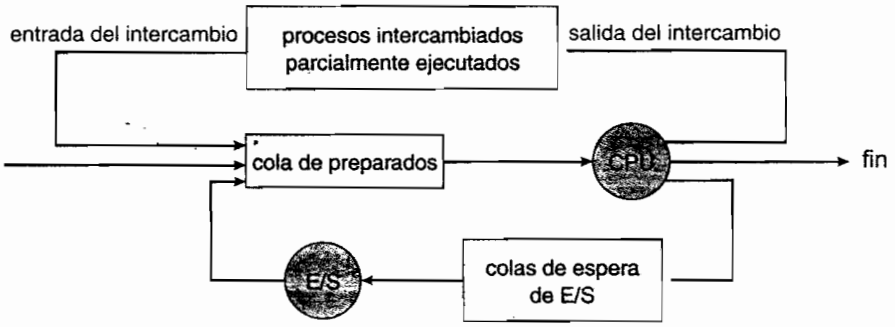

Figura 3.8 Adición de mecanismos de planificación a medio plazo en el diagrama de colas.

El tiempo dedicado al cambio de contexto es tiempo desperdiciado, dado que el sistema no realiza ningún trabajo útil durante la conmutación. La velocidad del cambio de contexto varía de una máquina a otra, dependiendo de la velocidad de memoria, del número de registros que tengan que copiarse y de la existencia de instrucciones especiales (como por ejemplo, una instrucción para cargar o almacenar todos los registros). Las velocidades típicas son del orden de unos pocos milisegundos.

El tiempo empleado en los cambios de contexto depende fundamentalmente del soporte hardware. Por ejemplo, algunos procesadores (como Ultra\SPARC de Sun) proporcionan múltiples conjuntos de registros. En este caso, un cambio de contexto simplemente requiere cambiar el puntero al conjunto actual de registros. Por supuesto, si hay más procesos activos que conjuntos de registros, el sistema recurrirá a copiar los datos de los registros en y desde memoria, al igual que antes. También, cuanto más complejo es el sistema operativo, más trabajo debe realizar durante un cambio de contexto. Como veremos en el Capítulo 8, las técnicas avanzadas de gestión de memoria pueden requerir que con cada contexto se intercambien datos adicionales. Por ejemplo, el espacio de direcciones del proceso actual debe preservarse en el momento de preparar para su uso el espacio de la siguiente tarea. Cómo se conserva el espacio de memoria y qué cantidad de trabajo es necesario para conservarlo depende del método de gestión de memoria utilizado por el sistema operativo.

3.3 Operaciones sobre los procesos

En la mayoría de los sistemas, los procesos pueden ejecutarse de forma concurrente y pueden crearse y eliminarse dinámicamente. Por tanto, estos sistemas deben proporcionar un mecanismo para la creación y terminación de procesos. En esta sección, vamos a ocuparnos de los mecanismos implicados en la creación de procesos y los ilustraremos analizando el caso de los sistemas UNIX y Windows.

3.3.1 Creación de procesos

Un proceso puede crear otros varios procesos nuevos mientras se ejecuta; para ello se utiliza una llamada al sistema específica para la creación de procesos. El proceso creador se denomina proceso **padre** y los nuevos procesos son los **hijos** de dicho proceso. Cada uno de estos procesos nuevos puede a su vez crear otros procesos, dando lugar a un **árbol** de procesos.

La mayoría de los sistemas operativos (incluyendo UNIX y la familia Windows de sistemas operativos) identifican los procesos mediante un **identificador de proceso** unívoco o **pid** (process identifier), que normalmente es un número entero. La Figura 3.9 ilustra un árbol de procesos típico en el sistema operativo Solaris, indicando el nombre de cada proceso y su pid. En Solaris, el proceso situado en la parte superior del árbol es el proceso `sched`, con el pid 0. El proceso `sched` crea varios procesos hijo, incluyendo `pageout` y `fsflush`. Estos procesos son responsables de la gestión de memoria y de los sistemas de archivos. El proceso `sched` también crea el proceso `init`, que sirve como proceso padre raíz para todos los procesos de usuario. En la Figura 3.9

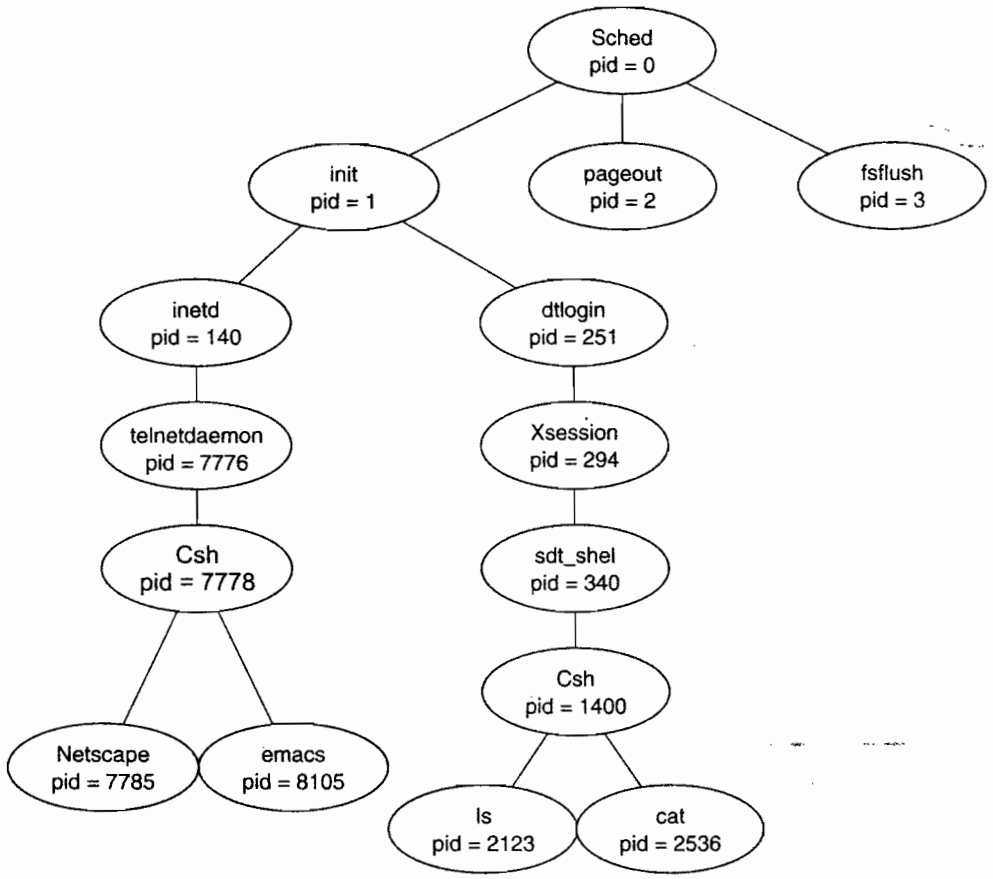

Figura 3.9 Árbol de procesos en un sistema Solaris típico.

vemos dos hijos de `init`: `inetd` y `dtlogin`. El proceso `inetd` es responsable de los servicios de red, como `telnet` y `ftp`; el proceso `dtlogin` es el proceso que representa una pantalla de inicio de sesión de usuario. Cuando un usuario inicia una sesión, `dtlogin` crea una sesión de X-Windows (`Xsession`), que a su vez crea el proceso `sdt_shel`. Por debajo de `sdt_shel`, se crea una *shell* de línea de comandos de usuario, C-*shell* o `csh`. Es en esta interfaz de línea de comandos donde el usuario invoca los distintos procesos hijo, tal como los comandos `ls` y `cat`. También vemos un proceso `csh` con el pid 7778, que representa a un usuario que ha iniciado una sesión en el sistema a través de `telnet`. Este usuario ha iniciado el explorador Netscape (pid 7785) y el editor emacs (pid 8105).

En UNIX, puede obtenerse un listado de los procesos usando el comando `ps`. Por ejemplo, el comando `ps -el` proporciona información completa sobre todos los procesos que están activos actualmente en el sistema. Resulta fácil construir un árbol de procesos similar al que se muestra en la Figura 3.9, trazando recursivamente los procesos padre hasta llegar al proceso `init`.

En general, un proceso necesitará ciertos recursos (tiempo de CPU, memoria, archivos, dispositivos de E/S) para llevar a cabo sus tareas. Cuando un proceso crea un subproceso, dicho subproceso puede obtener sus recursos directamente del sistema operativo o puede estar restringido a un subconjunto de los recursos del proceso padre. El padre puede tener que repartir sus recursos entre sus hijos, o puede compartir algunos recursos (como la memoria o los archivos) con algunos de sus hijos. Restringir un proceso hijo a un subconjunto de los recursos del padre evita que un proceso pueda sobrecargar el sistema creando demasiados subprocesos.

Además de los diversos recursos físicos y lógicos que un proceso obtiene en el momento de su creación, el proceso padre puede pasar datos de inicialización (entrada) al proceso hijo. Por ejemplo, considere un proceso cuya función sea mostrar los contenidos de un archivo, por ejemplo *img.jpg*, en la pantalla de un terminal. Al crearse, obtendrá como entrada de su proceso padre el

nombre del archivo *img.jpg* y empleará dicho nombre de archivo, lo abrirá y mostrará el contenido. También puede recibir el nombre del dispositivo de salida. Algunos sistemas operativos pasan recursos a los procesos hijo. En un sistema así, el proceso nuevo puede obtener como entrada dos archivos abiertos, *img.jpg* y el dispositivo terminal, y simplemente transferir los datos entre ellos.

Cuando un proceso crea otro proceso nuevo, existen dos posibilidades en términos de ejecución:

1. El padre continúa ejecutándose concurrentemente con su hijo.
2. El padre espera hasta que alguno o todos los hijos han terminado de ejecutarse.

También existen dos posibilidades en función del espacio de direcciones del nuevo proceso:

1. El proceso hijo es un duplicado del proceso padre (usa el mismo programa y los mismos datos que el padre).
2. El proceso hijo carga un nuevo programa.

Para ilustrar estas diferencias, consideremos en primer lugar el sistema operativo UNIX. En UNIX, como hemos visto, cada proceso se identifica mediante su identificador de proceso, que es un entero unívoco. Puede crearse un proceso nuevo mediante la llamada al sistema `fork()`. El nuevo proceso consta de una copia del espacio de direcciones del proceso original. Este mecanismo permite al proceso padre comunicarse fácilmente con su proceso hijo. Ambos procesos (padre e hijo) continúan la ejecución en la instrucción que sigue a `fork()`, con una diferencia: el código de retorno para `fork()` es cero en el caso del proceso nuevo (hijo), mientras que al padre se le devuelve el identificador de proceso (distinto de cero) del hijo.

Normalmente, uno de los dos procesos utiliza la llamada al sistema `exec()` después de una llamada al sistema `fork()`, con el fin de sustituir el espacio de memoria del proceso con un nuevo programa. La llamada al sistema `exec()` carga un archivo binario en memoria (destruyendo la imagen en memoria del programa que contiene la llamada al sistema `exec()`) e inicia su ejecución. De esta manera, los dos procesos pueden comunicarse y seguir luego caminos separados. El padre puede crear más hijos, o, si no tiene nada que hacer mientras se ejecuta el hijo, puede ejecutar una llamada al sistema `wait()` para auto-excluirse de la cola de procesos preparados hasta que el proceso hijo se complete.

El programa C mostrado en la Figura 3.10 ilustra las llamadas al sistema descritas, para un sistema UNIX. Ahora tenemos dos procesos diferentes ejecutando una copia del mismo programa. El valor pid del proceso hijo es cero; el del padre es un valor entero mayor que cero. El proceso hijo sustituye su espacio de direcciones mediante el comando `/bin/ls` de UNIX (utilizado para obtener un listado de directorios) usando la llamada al sistema `execlp()` (`execlp()` es una versión de la llamada al sistema `exec()`). El padre espera a que el proceso hijo se complete, usando para ello la llamada al sistema `wait()`. Cuando el proceso hijo termina (invocando implícita o explícitamente `exit()`), el proceso padre reanuda su ejecución después de la llamada a `wait()`, terminando su ejecución mediante la llamada al sistema `exit()`. Esta secuencia se ilustra en la Figura 3.11.

Como ejemplo alternativo, consideremos ahora la creación de procesos en Windows. Los procesos se crean en la API de Win32 mediante la función `CreateProcess()`, que es similar a `fork()` en el sentido de que un padre crea un nuevo proceso hijo. Sin embargo, mientras que con `fork()` el proceso hijo hereda el espacio de direcciones de su padre, `CreateProcess()` requiere cargar un programa específico en el espacio de direcciones del proceso hijo durante su creación. Además, mientras que a `fork()` no se le pasa ningún parámetro, `CreateProcess()` necesita al menos diez parámetros distintos.

El programa C mostrado en la Figura 3.12 ilustra la función `CreateProcess()`, la cual crea un proceso hijo que carga la aplicación mspaint.exe. Hemos optado por muchos de los valores predeterminados de los diez parámetros pasados a `CreateProcess()`. Animamos, a los lectores interesados en profundizar en los detalles sobre la creación y gestión de procesos en la API de Win32, a que consulten las notas bibliográficas incluidas al final del capítulo.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid.t pid;

    /*bifurca un proceso hijo */
    pid =fork();

    if (pid <0) {/* se produce un error */

      fprintf(strderr, "Fork Failed");
      exit(-1);
    }
    else if (pid ==0) {/* proceso hijo /
        execlp("/bin/ls/","ls", NULL);
    }
    else {/* proceso padre*/
      /* el padre espera a que el proceso hijo se complete */
      wait(NULL);
      printf("Hijo completado")
    }
}
```

Figura 3.10 Programa C que bifurca un proceso distinto.

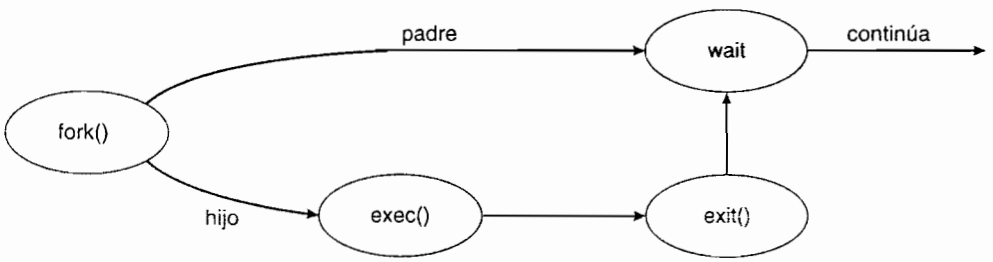

Figura 3.11 Creación de un proceso.

Los dos parámetros pasados a `CreateProcess()` son instancias de las estructuras STARTUPINFO y PROCESS_INFORMATION. STARTUPINFO especifica muchas propiedades del proceso nuevo, como el tamaño y la apariencia de la ventana y gestiona los archivos de entrada y de salida estándar. La estructura PROCESS_INFORMATION contiene un descriptor y los identificadores de los procesos recientemente creados y su hebra. Invocamos la función `ZeroMemory()` para asignar memoria a cada una de estas estructuras antes de continuar con `CreateProcess()`.

Los dos primeros parámetros pasados a `CreateProcess()` son el nombre de la aplicación los parámetros de la línea de comandos. Si el nombre de aplicación es NULL (en cuyo caso estamos), el parámetro de la línea de comandos especifica la aplicación que hay que cargar. En este caso, cargamos la aplicación *mspaint.exe* de Microsoft Windows. Además de estos dos parámetros iniciales, usamos los parámetros predeterminados para heredar los descriptores de procesos hebras, y no especificamos ningún indicador de creación. También usamos el bloque de entorno existente del padre y su directorio de inicio. Por último, proporcionamos dos punteros a las estructuras PROCESS_INFORMATION y STARTUPINFO creadas al principio del programa. En Figura 3.10, el proceso padre espera a que el hijo se complete invocando la llamada al sistema `wait()`. El equivalente en Win 32 es `WaitForSingleObject()`, a la que se pasa un descriptor

```
#include <stdio.h>
#include <windows.h>

int main (VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    // asignar memoria
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // crear proceso hijo
    if (!CreateProcess(NULL,   // utilizar línea de comandos
      "C:\\WINDOWS\\system32\\mspaint.exe",   // línea de comandos
      NULL,   // no hereda descriptor del proceso
      NULL,   // no hereda descriptor de la hebra
      FALSE,   // inhabilitar herencia del descriptor
      0,      // no crear indicadores
      NULL,   // usar bloque de entorno del padre
      NULL,   // usar directorio existente del padre
      &si,
      &pi))
    {
      fprintf(stderr,  "Fallo en la creación del proceso");
      return -1;
    }
    // el padre espera hasta que el hijo termina
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf "Hijo completado");

    // cerrar descritptores
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Figura 3.12 Creación de un proceso separado usando la API de Win32.

del proceso hijo, `pi.hProcess`, cuya ejecución queremos esperar a que se complete. Una vez que el proceso hijo termina, se devuelve el control desde la función `WaitForSingleObject()` del proceso padre.

3.3.2 Terminación de procesos

Un proceso termina cuando ejecuta su última instrucción y pide al sistema operativo que lo elimine usando la llamada al sistema `exit()`. En este momento, el proceso puede devolver un valor de estado (normalmente, un entero) a su proceso padre (a través de la llamada al sistema `wait()`). El sistema operativo libera la asignación de todos los recursos del proceso, incluyendo las memorias física y virtual, los archivos abiertos y los búferes de E/S.

La terminación puede producirse también en otras circunstancias. Un proceso puede causar la terminación de otro proceso a través de la adecuada llamada al sistema (por ejemplo, `TerminateProcess()` en Win32). Normalmente, dicha llamada al sistema sólo puede ser invocada por el padre del proceso que se va a terminar. En caso contrario, los usuarios podrían terminar arbitrariamente los trabajos de otros usuarios. Observe que un padre necesita conocer las

identidades de sus hijos. Por tanto, cuando un proceso crea un proceso nuevo, se pasa al padre la identidad del proceso que se acaba de crear.

Un padre puede terminar la ejecución de uno de sus hijos por diversas razones, como por ejemplo, las siguientes:

- El proceso hijo ha excedido el uso de algunos de los recursos que se le han asignado. Para determinar si tal cosa ha ocurrido, el padre debe disponer de un mecanismo para inspeccionar el estado de sus hijos.
- La tarea asignada al proceso hijo ya no es necesaria.
- El padre abandona el sistema, y el sistema operativo no permite que un proceso hijo continúe si su padre ya ha terminado.

Algunos sistemas, incluyendo VMS, no permiten que un hijo siga existiendo si su proceso padre se ha completado. En tales sistemas, si un proceso termina (sea normal o anormalmente), entonces todos sus hijos también deben terminarse. Este fenómeno, conocido como **terminación en cascada**, normalmente lo inicia el sistema operativo.

Para ilustrar la ejecución y terminación de procesos, considere que, en UNIX, podemos terminar un proceso usando la llamada al sistema `exit()`; su proceso padre puede esperar a la terminación del proceso hijo usando la llamada al sistema `wait()`. La llamada al sistema `wait()` devuelve el identificador de un proceso hijo completado, con el fin de que el padre puede saber cuál de sus muchos hijos ha terminado. Sin embargo, si el proceso padre se ha completado, a todos sus procesos hijo se les asigna el proceso `init` como su nuevo padre. Por tanto, los hijos todavía tienen un padre al que proporcionar su estado y sus estadísticas de ejecución.

3.4 Comunicación interprocesos

Los procesos que se ejecutan concurrentemente pueden ser procesos independientes o procesos cooperativos. Un proceso es **independiente** si no puede afectar o verse afectado por los restantes procesos que se ejecutan en el sistema. Cualquier proceso que no comparte datos con ningún otro proceso es un proceso independiente. Un proceso es **cooperativo** si puede afectar o verse afectado por los demás procesos que se ejecutan en el sistema. Evidentemente, cualquier proceso que comparte datos con otros procesos es un proceso cooperativo.

Hay varias razones para proporcionar un entorno que permita la cooperación entre procesos:

- **Compartir información**. Dado que varios usuarios pueden estar interesados en la misma información (por ejemplo, un archivo compartido), debemos proporcionar un entorno que permita el acceso concurrente a dicha información.
- **Acelerar los cálculos**. Si deseamos que una determinada tarea se ejecute rápidamente, debemos dividirla en subtareas, ejecutándose cada una de ellas en paralelo con las demás. Observe que tal aceleración sólo se puede conseguir si la computadora tiene múltiples elementos de procesamiento, como por ejemplos varias CPU o varios canales de E/S.
- **Modularidad**. Podemos querer construir el sistema de forma modular, dividiendo las funciones del sistema en diferentes procesos o hebras, como se ha explicado en el Capítulo 2.
- **Conveniencia**. Incluso un solo usuario puede querer trabajar en muchas tareas al mismo tiempo. Por ejemplo, un usuario puede estar editando, imprimiendo y compilando en paralelo.

La cooperación entre procesos requiere mecanismos de **comunicación interprocesos** (IPC, interprocess communication) que les permitan intercambiar datos e información. Existen dos modelos fundamentales de comunicación interprocesos: (1) **memoria compartida** y (2) **paso de mensajes**. En el modelo de memoria compartida, se establece una región de la memoria para que sea compartida por los procesos cooperativos. De este modo, los procesos pueden intercambiar información leyendo y escribiendo datos en la zona compartida. En el modelo de paso de mensa-