

***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 aislamien­ to 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 usua­ rios. Aunque su principal cometido es ejecutar programas de usuario, también tiene que ocupar­ se 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 si\_stema 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\_ I que se basa todo el funcio­ namiento de un sistema informático.

. . .. -• Describir los diversos mecanismos relacionados con los procesos,incluyendo Jos de planifica­ ción, 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 monou­ suario, como Microsof t 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 ejecu tar 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 aclividades son similares, por lo que a todas ellas las denominamos *procesos.*

En este texto, los términos *trabajo* y *proceso* se usa n indistintamente. Aunqu\_e personalmente preferimos el término *proceso,* gran parte de la teoría y lern1inología de los sisten1as operativos se

73

74 Capítulo 3 Procesos

desarrolló durante una época en que la principal actividad de los sistemas operativos era el pro- """' cesamiento de trabajos por lotes. Podría resultar confuso, por tanto, evitar la utilización de aque- *J:* llos términos comúnmente aceptados que incluyen la palabra *trabajo* (como por ejemplo *'t planificación de trabajos )* simplemente porque el término *proceso* haya sustituido a *trabajo.* ·

3.1 .1 El proceso

Informalmente, corno 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 deno­ mina 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 loca­ les), y una sección de datos, que contiene las variables globales. El proceso puede incluir, asimis­ mo, 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 deno­ minado 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 eje­ cutable. 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 ..)

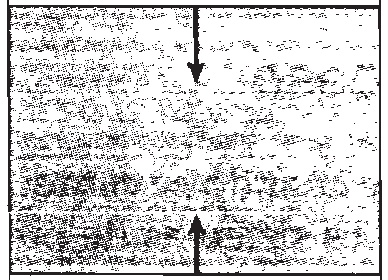
Aunque puede haber dos procesos asociados con el mismo programa, esos procesos se consi­ deran dos secuencias de ejecución separadas. Por ejemplo, varios usuarios pueden estar ejecutan­ do copias diferentes del programa de correo, o el mismo usua rio puede invocar muchas copias del explorador web. Cada una de estas copias es un proceso distinto y, aunque las secciones de texto sean equivalentes, las secciones de datos, del cúmulo *(lzeap )* 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 ejecu ta. En la Sección 3.4 se expiican estas cuestiones.

l

* 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:

max



pila

cúmulo de memoria

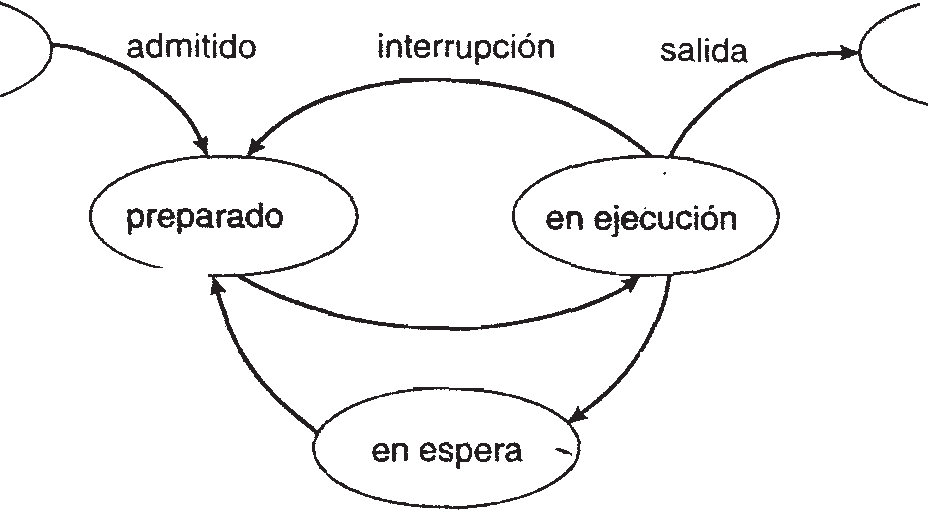
datos

texto

o

Figura 3.1 Proceso en memoria.

3.1 Concepto de proceso 75



nuevo

terminado

terminación de suceso

o de operación de E/S

en espera de suceso

o de operación de E/S

despacho por el planificador

**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 esta­ dos que representan se encuentran en todos los sistemas. Determinados sistemas operativos defi­ nen 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 dia­ grama de estados de un proceso genérico.

* 1. .3 Bloque de control de proceso

Cada proceso se represen ta en el sistema opera tivo mediante un bloque de control de proceso (PCB, process control bloc), también denominado *bloque de control de tarea* (véase la Figura 3.3). Un bloque de control de proceso contiene muchos elementos de inf ormación asociados con un proceso específico, entre los que se incluyen:

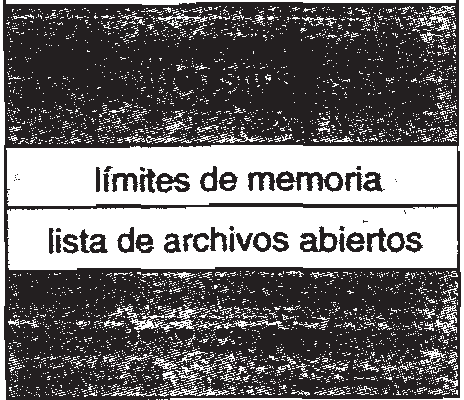
* + - Estado del proceso. El estado puede ser: nuevo, preparado, en ejecución, en espera, dete­ nido, etc.
    - Contador de programa. El contador indica la díf ección de la siguiente instrucción que va a ejecutar dicho proceso.
    - Registros de la CPU. Los registros varían en cuanto a número y tipo, 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 prod uce una interrupción, para que luego el proceso pueda continuar ejecu tán­ dose correctamente (Figura 3.4).
    - Información de planificación de la CPU. Esta información incluye la prioridad del proce­ so, los pu nteros a las colas de planificación y cualesquiera otros parámetros de planificación q ue se req uieran. El Capítu lo 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 página<:, o las tablas de segmentos, dependiendo del mecanismo de estión *d e* memoria u tilizado por el sistema opera tivo (Capítulo *8¡.*

76 Capítulo 3 Procesos



|  |
| --- |
| estado del proceso |
| número del proceso |
| contador de programa |

Figura 3.3 Bloque de control de proceso {PCB).



* Inf ormación 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 F/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 inf ormación que pueda variar de un proceso a otro.

3.i.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 pro­ ceso realizar sólo una tarea cada vez. Por ejemplo, el usuario no puede escribir simultáneamente



ejecución

ejecución

interrupción o llamada al sistema

guardar estado en PCB0



l

recargar estado de PCB1

inactividad interrupción o llamada al sistema

guardar estado en PGB1

recargar estado de PCB0

# )'"'"''""'

ejecución

# )'"''""'"'

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

3.2 Planificación de procesos 77

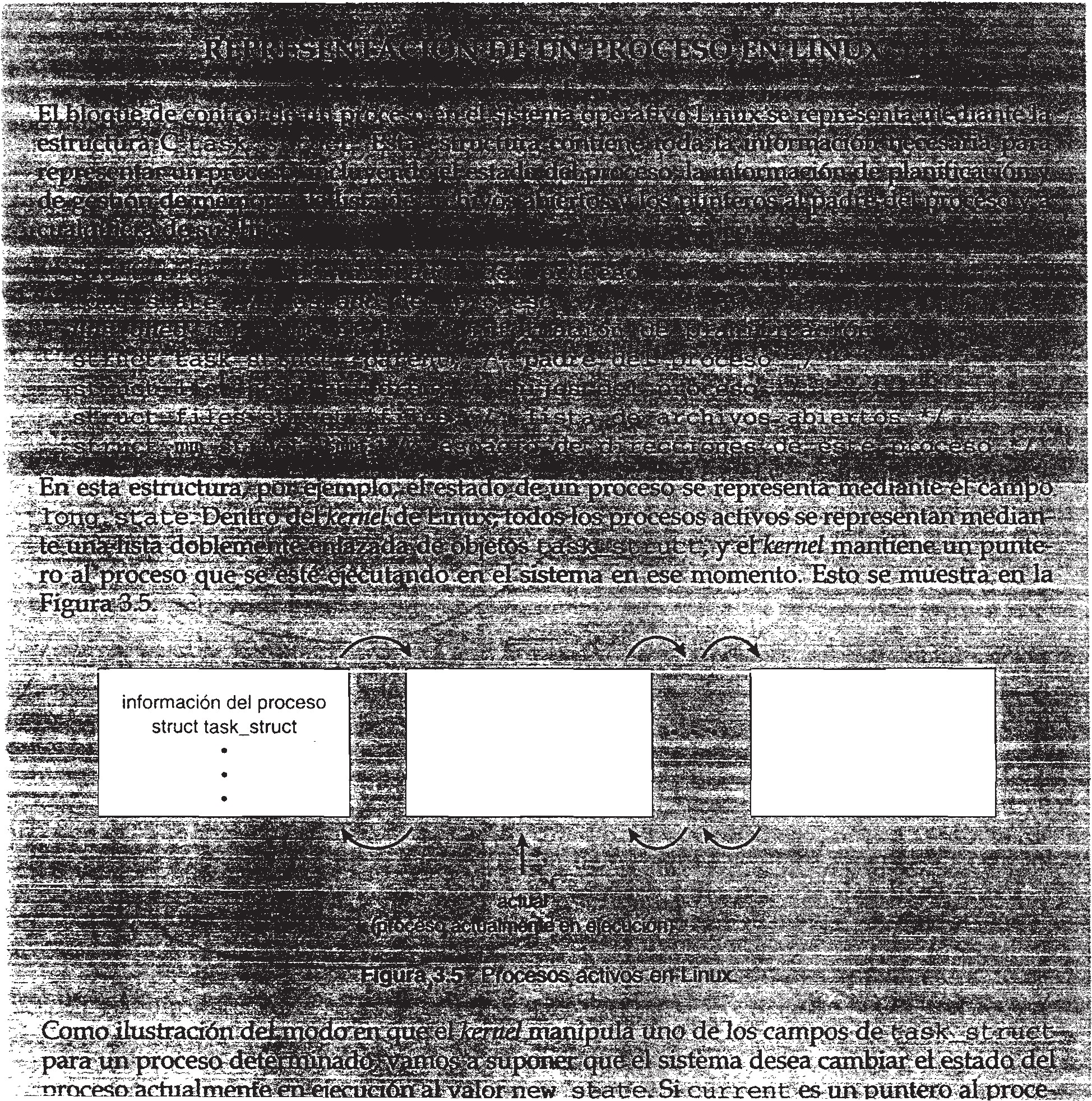


caracteres y pasar el corrector ortográfico dentro del mismo proceso. Muchos sistemas operativos modernos han ampliado el concepto de proceso para permitir que un proceso tenga múltiples hebras de ejecución y, por tanto, pueda llevar a cabo más de una tarea al mismo tiempo. El Capítulo 4 se ocupa en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

**f** -



información del proceso

struct task\_struct

información del proceso

struct task\_struct

1

## f

**t**.

78 Capítulo 3 Procesos

·.

cesos selecciona un proceso disponible (posiblemente de entre un conjunto de varios procesos dis-J ponibles) para ejecutar el programa en la CPU. En los sistemas de un solo procesador, nunca habrá.f:r 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.

* + 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 prepa­ rados y en espera de ejecutarse se mantienen en una lista denominada cola de procesos prepara­ dos. 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 pro­ cesos preparados.

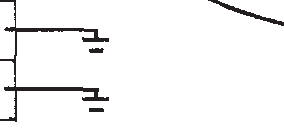
El sistema también incluye otras colas. Cuando se asigna la CPU a un proceso, éste se ejecuta durante un rato y finalmente termina, es interrumpido o espera a que se produzca un determina­ do suceso, como la terminación de una solicitud de E/S. Suponga que el proceso hace una solici­ tud de E/S a un 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 proce­ so. Por tanto, nuestro proceso puede tener que esperar para poder acceder al disco. La lista de pro­ cesos 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 dispositi­ vo. Los círculos representan los recursos que dan servicio a las colas y las flechas Indican el flujo de procesos en el sistema.

cola preparada

cabecera de cola PCB7 PC82

unidad O de cinta magnética



cabecera

cola

registros

registros

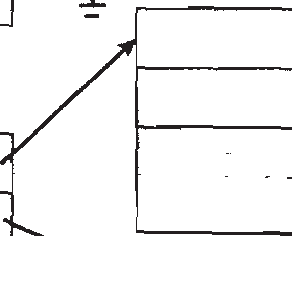
cabecera

cola

unidad 1 de cinta magnética



cabecera



cola

PCB14

PCB6

cabecera

cola

unidad O de disco

PCB5

r

unidad O 1 cabecera

de terminal cola 1

i

1

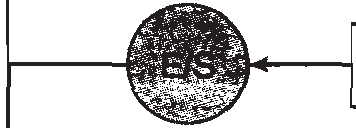
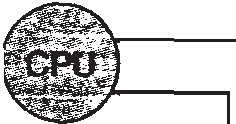
.,

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

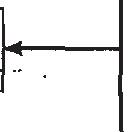
3.2 Planificación de procesos 79



cola de preparados **r-------------+i**



cola de E/S

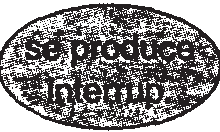


período de tiempo caducado

solicitud de E7S



bifurcar un hijo



espera una interrupción

**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 asig­ na 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 pro­ cesos 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.

* + 1. 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 planifi­ cador apropiado.

A menudo, en un sistema de procesamiento por lotes, se envían más procesos de los que pue­ den ser ejecutados de forma inmed iata. Estos procesos se guardan en cola en un dispositivo de almacenamiento masivo (normalmente, un disco), donde se mantienen para su posterior ejecu­ ción. 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 ejecu­ ción. El planificador a corto plazo debe seleccionar un nuevo proceso para la CPC frecuentemen­ te. Un proceso puede ejecutarse sólo durante unos pocos milisegundos antes de tener que esperar por una solicitud de E/S. Normalmente, el pla nificador a corto plazo se ejecuta al menos una vez cada 100 milisegundos. Debido al poco tiempo que hay entre ejecuciones, el planif icador a corto plazo debe ser rápido. Si tarda 10 milisegundos en decidir ejecutar un proceso dura nte 100 mili­ segundos, entonces el 10/(100 -'-10) = 9 por ciento del tiempo de CPL se está usando ( perdiendo) simplemente para planificar el trabajo.

80 Capítulo 3 Procesos

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 F/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 solicitu­

des 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, equili­ brando 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 planifica­ dor 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 com­ binación 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 menu­ do no disponen de planificado.r..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 sis­ temas 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 elimi­ nar 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 eje­ cución en el punto en que se interrumpió. Este esquema se denomina intercambio. El planifica­ dor a medio plazo descarga y luego vuelve a cargar el proceso. El intercambio puede ser necesaric para mejorar la mezcla deprocesos o porque un cambio en los requisitos de memoria haya hechc que se sobrepase la memoria disponible, requiriendo que se libere memoria. En el Capítulo 8 se estudian los mecanismos de intercambio.

* + 1. Cambio de contexto

Como se ha mencionado en la Sección 1.2.1, las interrupciones hacen que el sistema operativo obli gue a la CPU a abandonar su tarea actual, para ejecu tar una ru tina del *kernel.* Estos sucesos se pro

<lucen con frecuencia en los sistemas de propósito general. Cuando se prod uce una interrupción el sistema tiene que guardar el contexto actual del proceso que se está ejecutando en la ere, d• modo que pueda restaurar dicho contexto cuando su procesamiento concluya, suspendiendo e proceso y rean udánd olo después. El contexto se almacena en el PCB del proceso e incluye el valo de los registros de la CPU, el estado del proceso (véase la Figura 3.2) y la información de gestió1

de memoria. Es decir, realizamos una salvaguarda del estado actual de la CPU, en modo *kemel* '

en modo usua rio, y una restauración del estado para reanudar las operaciones.

La conmu tación de la ere *a* otro proceso requiere una salvaguarda del estado del proces actual y una resta uración del estado de otro proceso diferente. Esta tarea se conoce como cambi

de contexto. Cuando se produce un cambio de contexto, el *kernel* guarda el contexto del proces antiguo en su PCB y carga el contexto almacenado del nuevo proceso que se ha decidido ejecu ta:

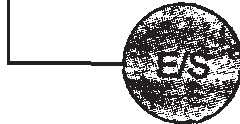
3.3 Operaciones sobrelos procesos 81

entrada del intercambio

procesos intercambiados pa,rcialmente ejecutados

salida del intercambio

-------+1cola de preparados 1-- ----+1



colas de espera de E/S

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 reali­ za 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 hard­ ware. 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 pun­ tero al conjunto actual de registros. Por supuesto, si hay más procesos activos que conjuntos de registros, el sistema recurrirá a c.apiar 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 ejecu tarse de forma concurrente y pueden cre­ arse y eliminarse dinámicamente. Por tanto, estos sistemas deben proporcionar un mecanismo para la creación y terminación de procesos. En esta sección, vamos a ocupamos de los mecanis­ mos implicados en la creación de procesos y los ilustraremos analizando el caso de los sistemas UNIX y Windows.

* + 1. Creación de procesos

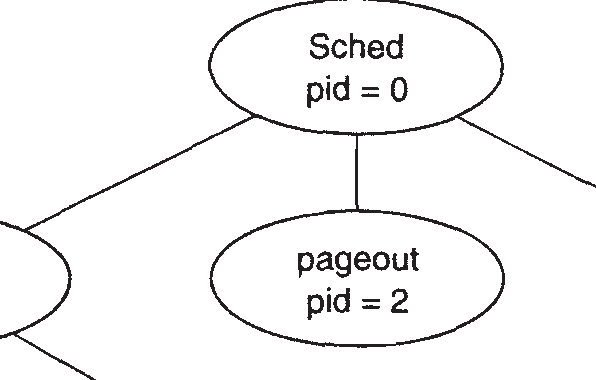
Un proceso puede crear otros varios procesos nuevos mientras se ejecuta; para ello se utiliza una llamada al sistema específica para la creación de procesos. El proceso creador se denomina proce­ so padre y los nuevos procesos son los hijos de dicho proceso. Cada uno de estos procesos·nue­ vos puede 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 ope­ rativos) identifican los procesos med iante un identificador de proceso unívoco o pid (process identifíer), que normalmente es un número entero. La Figura 3.9 ilustra un árbol de procesos típi­ co en el sistema operativo Solaris, ind icando 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 O. El proceso s c::--.2::: crea varios procesos hijo, incluyendo ¡::,a.;;2:=;·J.;:: y f s f lu.sh. Estos procesos son responsables de la gestión de memoria y de los sistemas de archivos. El proceso sched también crea el proceso

:'.. r:.:'.. :-.:. , que sirve como proceso pad re raíz para todos los procesos de usuario. En la Figura 3.9

. ·---

82 Capítulo 3 Procesos



init

pid = 1



fsflush pid = 3

inetd pid = 140

dtlogin pid = 251

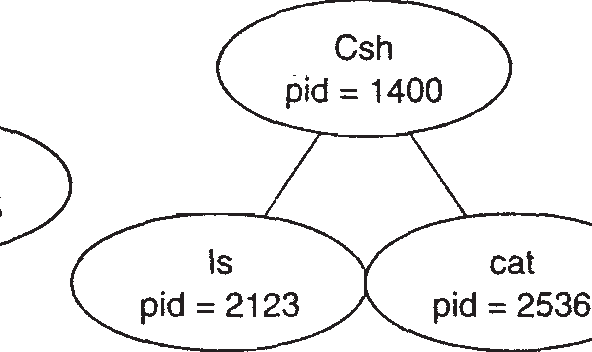
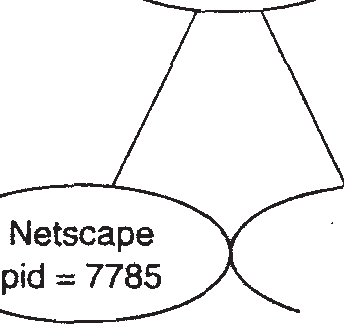
telnetdaemon pid = 7776

Xsession pid = 294

Csh

pid = 7778

sdt\_shel pid = 340



emacs

pid = 8105

**Figura** 3.9 Árbol de procesos en un sistema Solaris típico.

vemos dos hijos de init: inet d y dt login. El proceso inet d es responsable de los servicios de red, como t elnet y f t p; el proceso dt login es el proceso que representa una pantalla de inicio de sesión de usuario. Cuando un usuario inicia una sesión, dt.log in crea una sesión de X­ Windows (Xsess ion), que a su vez crea el proceso sdt\_shel. Por debajo de sdt\_she l, se crea una *shell* de línea de comandos de usuario, *C-shell* o c sh. Es en esta interfaz de línea de comando:: 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 edi­ tor emacs (pid 8105).

En UNIX, puede obtenerse un listado de los procesos usando el comando ps. Por ejemplo, el comando ps -el proporciona inf ormación 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.

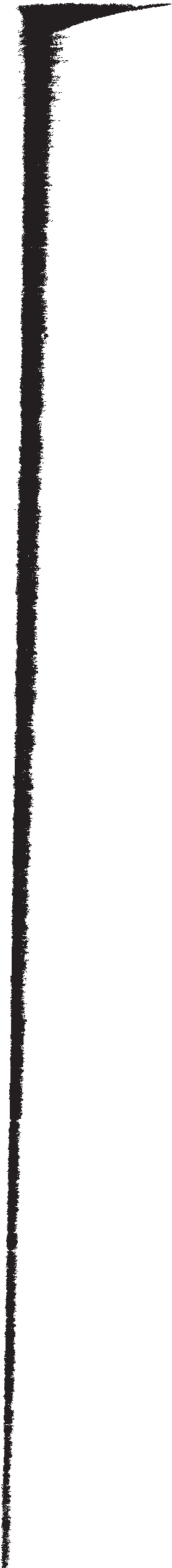
En general, un proceso necesitará ciertos recursos (tiempo de CPU, memoria, archivos, disposi­ tivos de E/S) para llevar a cabo sus tareas. Cuando un proceso crea un subproceso, dicho subpro­ ceso puede obtener sus recursos directamente del sistema operativo o puede estar restringido a un subconjunto de los recursos del proceso padre. El pad re puede tener que repartir sus recurso:: 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 pad re puede pasar datos de inicialización (entrada) al proceso hijo. Por ejem­

., plo, considere un proceso cuya función sea mostrar los contenidos de un archivo, por ejemplo

*inzg.jpg,* en la pantalla de un terminal. Al crearse, obtendrá como entrada de su proceso padre el

3.3 Operaciones sobre los procesos 83



nombre del archivo *img.jpg* y empleará dicho nombre de archivo, lo abrirá y mostrará el conteni­ do. 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 corno 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 ejecu­

ción:

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 f ork ( ) . El nuevo proceso consta de una copia del espacio de direcciones del proceso original. Este mecanis­ mo permite al proceso padre comunicarse fácilmente con su proceso hijo. Ambos procesos (padre e hijo) continúan la ejecución en la instrucción que sigue a f ork ( ) , con una diferencia: el código de retomo para f ork ( ) 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-siterna .exec ( ) después de una lla­ mada al sistema f ork ( ) , 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 ejecu­ ción. 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 eje­ cutar una llamada al sistema wa it ( ) 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 sis­ tema 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 éntero mayor que cero. El proceso hijo sustituye su espacio de direcciones mediante el comando / bin. / .:.. s de UNIX (utilizado para obte­ ner un listado de directorios) usando la llamada al sistema exec lp ( ) (exec lp ( ) es una versión de la llamada al sistema exec ( ) ). El pad re espera a que el proceso hijo se complete, usando para ello la llamada al sistema wa t ( ) . Cuando el proceso hijo termina (invocando implícita o explí­ citamente exit ( ) ), el proceso padre rean uda su ejecución después de la llamada a wa it ( ) , ter­ minando su ejecución mediante la llamada al sistema exit ( J . Esta secuencia se ilustra en la Figura 3.11.

Como ejemplo alternativo, consideremos ahora la creación de procesos en Windows. Los pro­ cesos se crean en la API de Win32 mediante la función Crea ce:?r ocess ( ) , que es similar a f ork ( ) en el sentido de que un padre crea un nuevo proceso hijo. Sin embargo, mientras que con f ork ( ) el proceso hijo hereda el espacio de direcciones de su padre, Crea t eProcess ( ) requie­ re cargar un programa específico en el espacio de direcciones del proceso hijo durante su creación. Además, mientras que a f ork ( ) no se le pasa ningún parámetro, Crea t eProcess ( ) necesita al menos diez parámetros distintos.

1

1

f El programa C mostrado en la Figura 3.12 ilustra la función C:::- eat eProcess ( ) , la cual crea

t

* un proceso hijo que carga f a aplicación mspaint.exe. Hemos optado por muchos de los valores

predeterminados de los diez parámetros pasados a

Crea::: e P:::cc ess ( ) . Animamos, a los lecto­

res interesados en profundizar en los detalles sobre la creación y gestión cj.e procesos en la API de Win32, a que consulten las notas bibliográficas incluidas al final del capítulo.

84 Capítulo 3 Procesos

#include <sys/types.h>

#include <stdio.h>

#include <unistd.h> int main ()

pid. t pid;

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

if (pid <Ü) {/\* se produce un error \* , fprint f (strderr, "Fork Failed") ;

exit (-1) ;

else if (pid == Ü ) {/\* proceso hijo / execlp(•/bin/ls/", "ls" , NULL);

else {/\* proceso padre\*/

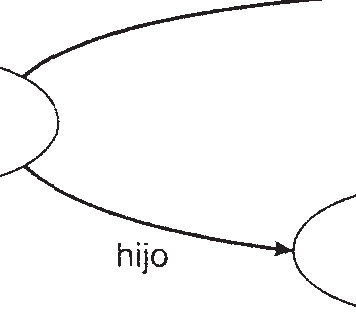
/\* el padre espera a que el proceso ijo se complete \*/ wait: (NULL);

printf ("Hijo completado")

.......\_ **Figura 3.1O** Programa C que bifurca un proceso distinto.

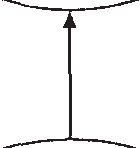
padre

wait



fork()

exec()



**exit()**

continúa

**Figura 3.11** Creación de un proceso.

Los dos parámetros pasados a Crea t eProcess ( ) son instancias de las estructuras STARTl PINFO y PROCESS\_INFORMATION. STARTUPINFO especifica muchas propiedades del proces nuevo, como el tamaño y la apariencia de la ventana y gestiona los archivos de entrada y de sal da estándar. La estructura PROCESS\_INFORMATION contiene un descriptor y los identificadores d los procesos recientemente creados y su hebra. Invocamos la función Z eroMemory ( ) para asi nar memoria a cada una de estas estructuras antes de continuar con Crea t eProcess ( ) .

Los dos primeros parámetros pasados a Crea t eProcess ( ) son el nombre de la aplicación los parámetros de la línea de comandos. Si el nombre de aplicación es NULL (en cuyo caso esL· mos), el parámetro de la línea de comandos especifica la aplicación que hay que cargar. En es' caso, cargamos la aplicación *mspaint.exe* de Microsof t Windows. Además de estos dos parámetn iniciales, usamos los pa rámetros predeterminados para heredar los descriptores de procesos hebras, y no especif icamos n ingú n indicador de creación. También usamos el bloque de entorr existente del padre y su directorio de inicio. Por úl timo, proporcionamos dos punteros a L estructuras PROCESS\_L'<FORMA TION y STARTUPINFO creadas al principio del programa. En

Fi gura 3.1 O, el proceso pad re espera a que el hijo se complete im·ocando la llamada al sisteff

wa i t ( ) . El equivalen te en Win 32 es :·:a i c.Fo rS ingleOb j ec t ( , a la que se pasa un descript'

3.3 Operaciones sobre los procesos 85

#include <s dio.h>

#include <idows.h> int main (·.-J:L D)

STARTUPINFC si;

PROCESS IN?ORM..l\TION pi;

*11* asignar memoria ZeroMecory(&si, sizeof (si)); si.cb = sizeof (si); Zer0Me2ory(&pi, sizeof(pi));

*11* crear proceso hijo

ir (!CreateProcess(NULL, // utilizar línea de comandos "C:\·;·::LNDOWS\\system32\\mspaint.exe" , // línea de comandos NULL, // no hereda descriptor del proceso

NULL, // no hereda descriptor de la hebra

FALS , // inhabilitar herencia del descriptor

O , / / no crear indicadores

NULL, // usar bloque de entorno del padre NULL, // usar directorio existente del padre

&si,

&pi))

fpri::-. *i'* ( st.?err,

rect:::-:--.. -1;

" Fallo en la creación del proceso" );

*/ !* el ;;adre espera hasta que el hijo termina WaitForSingleObject(pi.hProcess, INFINITE); print ·ijo completado" );



*11* cerrar descritptores Close dle(pi. h?rocess); Clos adle(pi.hThread);

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

del proceso hijo, p.:. . hProcess,cuya ejecución queremos esperar a que se complete. Una vez que el proceso hijo termina, se devuelve el control desde la función WaitForSingleübjec:: (l del proceso padre.

* + 1. Terminación de procesos

.Un proceso termina cuando ejecuta su última instrucción y pide al sistema operativo que lo elimi­ ne 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

·.;ait ( ) ). El sistema opera tivo libera la asignación de todos los recursos del proceso, incluyendo las memorias física y vir tual, los archivos abiertos y los búferes de E/S.

La terminación puede prod ucirse también en otras circunstancias. Un proceso puede causar la terminación de otro proceso a través de la adecuada llamada al sistema (por ejemplo,

-:- er'<.i"--=''.: .:? Proc"::;.; *(* l en Win32). Normalmente, dicha llamada al sistema sólo puede ser invo­ cada por el pad re del proceso.«i.ue se va a terminar. En caso contrario, los usuarios podrían termi­ nar arbitrariamente los trabajos de otros usuarios. Observe que un padre necesita conocer las

86 Capítulo 3 Procesos



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 ejem­ pla, 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 inspeccio­ nar 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 conti­ núe 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), enton­ ces todos sus hijos también deben terminarse. Este fenómeno, conocido como terminación en cas­ cada, normalmente lo inicia el sistema operativo.

Para ilustrar la ejecución y terminación de procesos, considere que, en UNIX, podemos termi­ nar un proceso usando la llamada al sistema exi t ( ) ; su proceso padre puede esperar a la termi­ nación del proceso hijo usando la llamada al sistema wa it ( ) . La llamada al sistema wa it ( ) 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 pt'Of>Orcionar su estado y sus estadísticas de ejecución.

## 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 afecta­ do por los demás procesos que se ejecutan en el sistema. Evidentemente, cualquier proceso que comparte datos con otros procesos es un proceso cooperativo. .

1

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

* + - Compartir inf ormación. 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 inf ormación.
    - 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 ele­ mentos de procesamiento, como por ejemplos varias CPU o varios canales de E/S.
    - Modularidad. Podemos querer construir el sistema de forma modular, dividiendo las fun­ ciones del sistema en diferentes procesos o hebras, como se ha explicado en el Capítulo 2.
    - Conveniencia. Incluso un solo usuario puede querer trabajar eñ muchas tareas al mismo tiempo. Por ejemplo, un usuario puede estar editando, imprimiendo y compilando en para- lelo. ·

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-