

# Tema 2. Introducción a los Sistemas Operativos

## Contenidos

### 2.1 Componentes de un Sistema Operativo (SO) multiprogramado.

2.1.1 Sistemas multiprogramados y de tiempo compartido.

2.1.2 Concepto de proceso.

2.1.3 Modelo de cinco estados de los procesos.

### 2.2 Descripción y control de procesos.

2.2.1 Bloque de control de proceso (PCB).

2.2.2 Control de procesos.

### 2.3 Hebras (hilos)

### 2.4 Gestión básica de memoria.

2.4.1 Paginación

2.4.2 Segmentación

## Objetivos

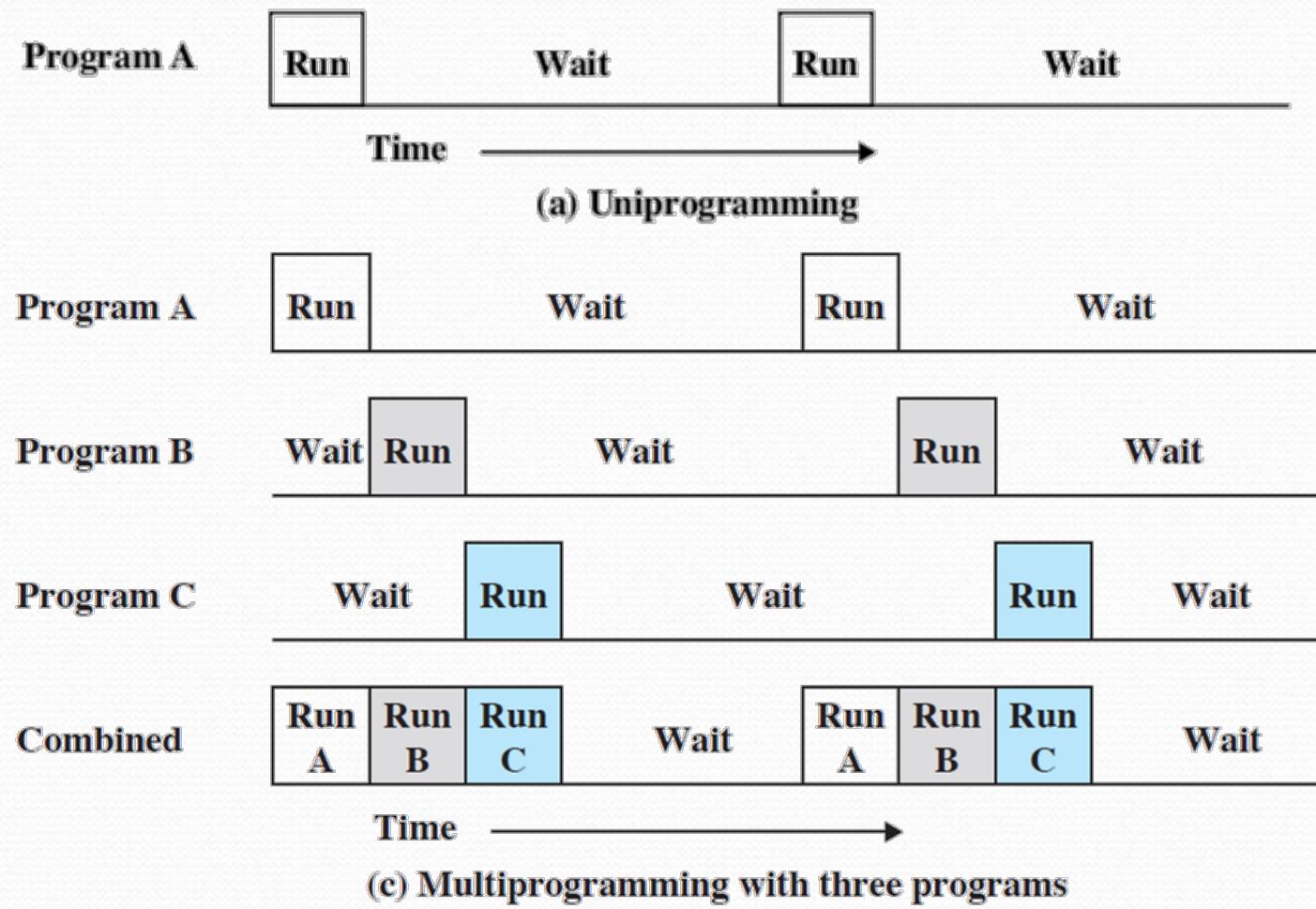
- Conocer los elementos necesarios para implementar la multiprogramación en un sistema operativo (SO).
- Conocer el concepto de proceso y el modelo de cinco estados de los procesos.
- Conocer el uso que realiza el SO del apoyo hardware e integrarlo en el modelo de cinco estados.
- Conocer el concepto de hebra (hilo), su modelo de cinco estados y su utilidad.
- Conocer la gestión básica de memoria que realiza el SO.

## Bibliografía básica

[Stal05] [W. Stallings](#), **Sistemas Operativos, Aspectos Internos y Principios de Diseño (5<sup>a</sup> Edición)**. Pearson Education, 2005

[Carr07] [J. Carretero, F. García, P. de Miguel, F. Pérez](#), **Sistemas Operativos (2<sup>a</sup> Edición)**, McGraw-Hill, 2007

## 2.1.1 Concepto de multiprogramación [Stall05] (pp. 58—67)



La necesidad de cambiar regularmente un sistema operativo introduce ciertos requisitos en su diseño. Un hecho obvio es que el sistema debe tener un diseño modular, con interfaces entre los módulos claramente definidas, y que debe estar bien documentado. Para programas grandes, tal como el típico sistema operativo contemporáneo, llevar a cabo una modularización sencilla no es adecuado [DENN80a]. Se detallará este tema más adelante en el capítulo.

## 2.2. LA EVOLUCIÓN DE LOS SISTEMAS OPERATIVOS

Para comprender los requisitos claves de un sistema operativo y el significado de las principales características de un sistema operativo contemporáneo, es útil considerar la evolución de los sistemas operativos a lo largo de los años.

### PROCESAMIENTO SERIE

Con los primeros computadores, desde finales de los años 40 hasta mediados de los años 50, el programador interaccionaba directamente con el hardware del computador; no existía ningún sistema operativo. Estas máquinas eran utilizadas desde una consola que contenía luces, interruptores, algún dispositivo de entrada y una impresora. Los programas en código máquina se cargaban a través del dispositivo de entrada (por ejemplo, un lector de tarjetas). Si un error provocaba la parada del programa, las luces indicaban la condición de error. El programador podía entonces examinar los registros del procesador y la memoria principal para determinar la causa de error. Si el programa terminaba de forma normal, la salida aparecía en la impresora.

Estos sistemas iniciales presentaban dos problemas principales:

- **Planificación.** La mayoría de las instalaciones utilizaban una plantilla impresa para reservar tiempo de máquina. Típicamente, un usuario podía solicitar un bloque de tiempo en múltiplos de media hora aproximadamente. Un usuario podía obtener una hora y terminar en 45 minutos; esto implicaba malgastar tiempo de procesamiento del computador. Por otro lado, el usuario podía tener problemas, si no finalizaba en el tiempo asignado y era forzado a terminar antes de resolver el problema.
- **Tiempo de configuración.** Un único programa, denominado **trabajo**, podía implicar la carga en memoria del compilador y del programa en lenguaje de alto nivel (programa en código fuente) y a continuación la carga y el enlace del programa objeto y las funciones comunes. Cada uno de estos pasos podían suponer montar y desmontar cintas o configurar tarjetas. Si ocurría un error, el desgraciado usuario normalmente tenía que volver al comienzo de la secuencia de configuración. Por tanto, se utilizaba una cantidad considerable de tiempo en configurar el programa que se iba a ejecutar.

Este modo de operación puede denominarse procesamiento serie, para reflejar el hecho de que los usuarios acceden al computador en serie. A lo largo del tiempo, se han desarrollado varias herramientas de software de sistemas con el fin de realizar el procesamiento serie más eficiente. Estas herramientas incluyen bibliotecas de funciones comunes, enlazadores, cargadores, depuradores, y rutinas de gestión de E/S disponibles como software común para todos los usuarios.

### SISTEMAS EN LOTES SENCILLOS

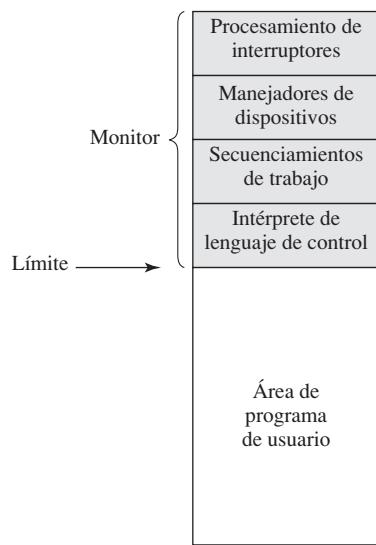
Las primeras máquinas eran muy caras, y por tanto, era importante maximizar su utilización. El tiempo malgastado en la planificación y configuración de los trabajos era inaceptable.

Para mejorar su utilización, se desarrolló el concepto de sistema operativo en lotes. Parece ser que el primer sistema operativo en lotes (y el primer sistema operativo de cualquier clase) fue desarrollado a mediados de los años 50 por General Motors para el uso de un IBM 701 [WEIZ81]. El concepto fue subsecuentemente refinado e implementado en el IBM 704 por un número de clientes de IBM. A principios de los años 60, un número de vendedores había desarrollado sistemas operativos en lote para sus sistemas de computación. IBSYS, el sistema operativo de IBM para los computadores 7090/7094, es particularmente notable por su gran influencia en otros sistemas.

La idea central bajo el esquema de procesamiento en lotes sencillo es el uso de una pieza de software denominada **monitor**. Con este tipo de sistema operativo, el usuario no tiene que acceder directamente a la máquina. En su lugar, el usuario envía un trabajo a través de una tarjeta o cinta al operador del computador, que crea un sistema por lotes con todos los trabajos enviados y coloca la secuencia de trabajos en el dispositivo de entrada, para que lo utilice el monitor. Cuando un programa finaliza su procesamiento, devuelve el control al monitor, punto en el cual dicho monitor comienza la carga del siguiente programa.

Para comprender cómo funciona este esquema, se puede analizar desde dos puntos de vista: el del monitor y el del procesador.

- **Punto de vista del monitor.** El monitor controla la secuencia de eventos. Para ello, una gran parte del monitor debe estar siempre en memoria principal y disponible para la ejecución (Figura 2.3). Esta porción del monitor se denomina **monitor residente**. El resto del monitor está formado por un conjunto de utilidades y funciones comunes que se cargan como subrutinas en el programa de usuario, al comienzo de cualquier trabajo que las requiera. El monitor lee de uno en uno los trabajos desde el dispositivo de entrada (normalmente un lector de tarjetas o dispositivo de cinta magnética). Una vez leído el dispositivo, el trabajo actual se coloca en el área de programa de usuario, y se le pasa el control. Cuando el trabajo se ha completado, devuelve el control al monitor, que inmediatamente lee el siguiente trabajo. Los resultados de cada trabajo se envían a un dispositivo de salida, (por ejemplo, una impresora), para entregárselo al usuario.



**Figura 2.3.** Disposición de memoria de un monitor residente.

- **Punto de vista del procesador.** En un cierto punto, el procesador ejecuta instrucciones de la zona de memoria principal que contiene el monitor. Estas instrucciones provocan que se lea el siguiente trabajo y se almacene en otra zona de memoria principal. Una vez que el trabajo se ha leído, el procesador encontrará una instrucción de salto en el monitor que le indica al procesador que continúe la ejecución al inicio del programa de usuario. El procesador entonces ejecutará las instrucciones del programa usuario hasta que encuentre una condición de finalización o de error. Cualquiera de estas condiciones hace que el procesador ejecute la siguiente instrucción del programa monitor. Por tanto, la frase «se pasa el control al trabajo» simplemente significa que el procesador leerá y ejecutará instrucciones del programa de usuario, y la frase «se devuelve el control al monitor» indica que el procesador leerá y ejecutará instrucciones del programa monitor.

El monitor realiza una función de planificación: en una cola se sitúa un lote de trabajos, y los trabajos se ejecutan lo más rápidamente posible, sin ninguna clase de tiempo ocioso entre medias. Además, el monitor mejora el tiempo de configuración de los trabajos. Con cada uno de los trabajos, se incluye un conjunto de instrucciones en algún formato primitivo de **lenguaje de control de trabajos** (*Job Control Language*, JCL). Se trata de un tipo especial de lenguaje de programación utilizado para dotar de instrucciones al monitor. Un ejemplo sencillo consiste en un usuario enviando un programa escrito en el lenguaje de programación FORTRAN más algunos datos que serán utilizados por el programa. Además del código en FORTRAN y las líneas de datos, el trabajo incluye instrucciones de control del trabajo, que se representan mediante líneas que comienzan mediante el símbolo `\$. El formato general del trabajo tiene el siguiente aspecto:

```

$JOB
$FTN
•|
•| Instrucciones FORTRAN
•|
$LOAD
$RU
N|
•| Datos
•|
$END

```

Para ejecutar este trabajo, el monitor lee la línea \$FTN y carga el compilador apropiado de su sistema de almacenamiento (normalmente una cinta). El compilador traduce el programa de usuario en código objeto, el cual se almacena en memoria en el sistema de almacenamiento. Si se almacena en memoria, la operación se denomina «compilar, cargar, y ejecutar». En el caso de que se almacene en una cinta, se necesita utilizar la instrucción \$LOAD. El monitor lee esta instrucción y recupera el control después de la operación de compilación. El monitor invoca al cargador, que carga el programa objeto en memoria (en el lugar del compilador) y le transfiere el control. De esta forma, se puede compartir un gran segmento de memoria principal entre diferentes subsistemas, aunque sólo uno de ellos se puede ejecutar en un momento determinado.

Durante la ejecución del programa de usuario, cualquier instrucción de entrada implica la lectura de una línea de datos. La instrucción de entrada del programa de usuario supone la invocación de una rutina de entrada, que forma parte del sistema operativo. La rutina de entrada comprueba que el pro-

grama no lea accidentalmente una línea JCL. Si esto sucede, se genera un error y se transfiere el control al monitor. Al finalizar el trabajo de usuario, el monitor analizará todas las líneas de entrada hasta que encuentra la siguiente instrucción JCL. De esta forma, el sistema queda protegido frente a un programa con excesivas o escasas líneas de datos.

El monitor, o sistema operativo en lotes, es simplemente un programa. Éste confía en la habilidad del procesador para cargar instrucciones de diferentes porciones de la memoria principal que de forma alternativa le permiten tomar y abandonar el control. Otras características hardware son también deseables:

- **Protección de memoria.** Durante la ejecución del programa de usuario, éste no debe alterar el área de memoria que contiene el monitor. Si esto ocurriera, el hardware del procesador debe detectar un error y transferir el control al monitor. El monitor entonces abortará el trabajo, imprimirá un mensaje de error y cargará el siguiente trabajo.
- **Temporizador.** Se utiliza un temporizador para evitar que un único trabajo monopolice el sistema. Se activa el temporizador al comienzo de cada trabajo. Si el temporizador expira, se para el programa de usuario, y se devuelve el control al monitor.
- **Instrucciones privilegiadas.** Ciertas instrucciones a nivel de máquina se denominan privilegiadas y sólo las puede ejecutar el monitor. Si el procesador encuentra estas instrucciones mientras ejecuta un programa de usuario, se produce un error provocando que el control se transfiera al monitor. Entre las instrucciones privilegiadas se encuentran las instrucciones de E/S, que permiten que el monitor tome control de los dispositivos de E/S. Esto evita, por ejemplo, que un programa de usuario de forma accidental lea instrucciones de control de trabajos del siguiente trabajo. Si un programa de usuario desea realizar operaciones de E/S, debe solicitar al monitor que realice las operaciones por él.
- **Interrupciones.** Los modelos de computadores iniciales no tenían esta capacidad. Esta característica proporciona al sistema operativo más flexibilidad para dejar y retomar el control desde los programas de usuario.

Ciertas consideraciones sobre la protección de memoria y las instrucciones privilegiadas llevan al concepto de modos de operación. Un programa de usuario ejecuta en **modo usuario**, en el cual los usuarios no pueden acceder a ciertas áreas de memoria y no puede ejecutar ciertas instrucciones. El monitor ejecuta en modo sistema, o lo que se denomina **modo núcleo**, en el cual se pueden ejecutar instrucciones privilegiadas y se puede acceder a áreas de memoria protegidas.

Por supuesto, se puede construir un sistema operativo sin estas características. Pero los fabricantes de computadores rápidamente se dieron cuenta de que los resultados no eran buenos, y de este modo, se construyeron sistemas operativos en lotes primitivos con estas características hardware.

Con un sistema operativo en lotes, el tiempo de máquina alterna la ejecución de programas de usuario y la ejecución del monitor. Esto implica dos sacrificios: el monitor utiliza parte de la memoria principal y consume parte del tiempo de máquina. Ambas situaciones implican una sobrecarga. A pesar de esta sobrecarga, el sistema en lotes simple mejora la utilización del computador.

## SISTEMAS EN LOTES MULTIPROGRAMADOS

El procesador se encuentra frecuentemente ocioso, incluso con el secuenciamiento de trabajos automático que proporciona un sistema operativo en lotes simple. El problema consiste en que los dispositivos de E/S son lentos comparados con el procesador. La Figura 2.4 detalla un cálculo representativo de este hecho, que corresponde a un programa que procesa un fichero con registros y

realiza de media 100 instrucciones máquina por registro. En este ejemplo, el computador malgasta aproximadamente el 96% de su tiempo esperando a que los dispositivos de E/S terminen de transferir datos a y desde el fichero. La Figura 2.5a muestra esta situación, donde existe un único programa, lo que se denomina monoprogramación. El procesador ejecuta durante cierto tiempo hasta que alcanza una instrucción de E/S. Entonces debe esperar que la instrucción de E/S concluya antes de continuar.

Leer un registro del fichero	$15 \mu\text{s}$
Ejecutar 100 instrucciones	$1 \mu\text{s}$
Escribir un registro al fichero	$15 \mu\text{s}$
TOTAL	$31 \mu\text{s}$
Porcentaje de utilización de la CPU	$\frac{1}{31} = 0,032 = 3,2\%$

Figura 2.4. Ejemplo de utilización del sistema.

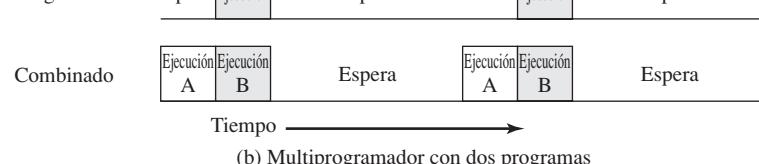
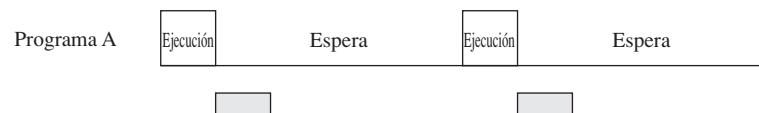
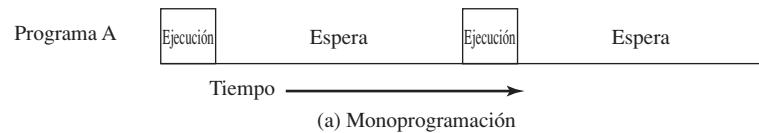


Figura 2.5. Ejemplo de multiprogramación.

Esta ineficiencia puede evitarse. Se sabe que existe suficiente memoria para contener al sistema operativo (monitor residente) y un programa de usuario. Supóngase que hay espacio para el sistema operativo y dos programas de usuario. Cuando un trabajo necesita esperar por la E/S, se puede asignar el procesador al otro trabajo, que probablemente no esté esperando por una operación de E/S (Figura 2.5b). Más aún, se puede expandir la memoria para que albergue tres, cuatro o más programas y pueda haber multiplexación entre todos ellos (Figura 2.5c). Este enfoque se conoce como **multiprogramación** o **multitarea**. Es el tema central de los sistemas operativos modernos.

Para mostrar los beneficios de la multiprogramación, se describe un ejemplo sencillo. Sea un computador con 250 Mbytes de memoria disponible (sin utilizar por el sistema operativo), un disco, un terminal y una impresora. Se envían simultáneamente a ejecución tres programas TRABAJO1, TRABAJO2 y TRABAJO3, con las características listadas en la Tabla 2.1. Se asumen requisitos mínimos de procesador para los trabajos TRABAJO1 y TRABAJO3, así como uso continuo de impresora y disco por parte del trabajo TRABAJO3. En un entorno por lotes sencillo, estos trabajos se ejecutarán en secuencia. Por tanto, el trabajo TRABAJO1 se completará en 5 minutos. El trabajo TRABAJO2 esperará estos 5 minutos y a continuación se ejecutará, terminando 15 minutos después. El trabajo TRABAJO3 esperará estos 20 minutos y se completará 30 minutos después de haberse enviado. La media de utilización de recursos, productividad y tiempos de respuestas se muestran en la columna de monoprogramación de la Tabla 2.2. La utilización de cada dispositivo se ilustra en la Figura 2.6a. Es evidente que existe una infroutilización de todos los recursos cuando se compara respecto al periodo de 30 minutos requerido.

**Tabla 2.1.** Atributos de ejecución de ejemplos de programas.

	<b>TRABAJO 1</b>	<b>TRABAJO 2</b>	<b>TRABAJO 3</b>
<b>Tipo de trabajo</b>	Computación pesada	Gran cantidad de E/S	Gran cantidad de E/S
<b>Duración</b>	5 minutos	15 minutos	10 minutos
<b>Memoria requerida</b>	50 M	100 M	75 M
<b>¿Necesita disco?</b>	No	No	Sí
<b>¿Necesita terminal?</b>	No	Sí	No
<b>¿Necesita impresora?</b>	No	No	Sí

**Tabla 2.2.** Efectos de la utilización de recursos sobre la multiprogramación.

	<b>Monoprogramación</b>	<b>Multiprogramación</b>
<b>Uso de procesador</b>	20%	40%
<b>Uso de memoria</b>	33%	67%
<b>Uso de disco</b>	33%	67%
<b>Uso de impresora</b>	33%	67%
<b>Tiempo transcurrido</b>	30 minutos	15 minutos
<b>Productividad</b>	6 trabajos/hora	12 trabajos/hora
<b>Tiempo de respuesta medio</b>	18 minutos	10 minutos

Ahora supóngase que los trabajos se ejecutan concurrentemente bajo un sistema operativo multiprogramado. Debido a que hay poco conflicto entre los trabajos, todos pueden ejecutar casi en el mínimo tiempo mientras coexisten con los otros en el computador (asumiendo que se asigne a los trabajos TRABAJO2 y TRABAJO3 suficiente tiempo de procesador para mantener sus operaciones de entrada y salida activas). El trabajo TRABAJO1 todavía requerirá 5 minutos para completarse, pero al final de este tiempo, TRABAJO2 habrá completado un tercio de su trabajo y TRABAJO3 la mitad. Los tres trabajos habrán finalizado en 15 minutos. La mejora es evidente al examinar la columna de multiprogramación de la Tabla 2.2, obtenido del histograma mostrado en la Figura 2.6b.

Del mismo modo que un sistema en lotes simple, un sistema en lotes multiprogramado también debe basarse en ciertas características hardware del computador. La característica adicional más notable que es útil para la multiprogramación es el hardware que soporta las interrupciones de E/S y DMA (*Direct Memory Access*: acceso directo a memoria). Con la E/S gestionada a través de interrupciones o DMA, el procesador puede solicitar un mandato de E/S para un trabajo y continuar con la ejecución de otro trabajo mientras el controlador del dispositivo gestiona dicha operación de E/S. Cuando esta última operación finaliza, el procesador es interrumpido y se pasa el control a un programa de tratamiento de interrupciones del sistema operativo. Entonces, el sistema operativo pasará el control a otro trabajo.

Los sistemas operativos multiprogramados son bastante sofisticados, comparados con los sistemas **monoprogramados**. Para tener varios trabajos listos para ejecutar, éstos deben guardarse en memoria principal, requiriendo alguna forma de **gestión de memoria**. Adicionalmente, si varios trabajos están listos para su ejecución, el procesador debe decidir cuál de ellos ejecutar; esta decisión requiere un algoritmo para planificación. Estos conceptos se discuten más adelante en este capítulo.

## SISTEMAS DE TIEMPO COMPARTIDO

Con el uso de la multiprogramación, el procesamiento en lotes puede ser bastante eficiente. Sin embargo, para muchos trabajos, es deseable proporcionar un modo en el cual el usuario interaccione directamente con el computador. De hecho, para algunos trabajos, tal como el procesamiento de transacciones, un modo interactivo es esencial.

Hoy en día, los computadores personales dedicados o estaciones de trabajo pueden cumplir, y frecuentemente lo hacen, los requisitos que necesita una utilidad de computación interactiva. Esta opción no estuvo disponible hasta los años 60, cuando la mayoría de los computadores eran grandes y costosos. En su lugar, se desarrolló el concepto de tiempo compartido.

Del mismo modo que la multiprogramación permite al procesador gestionar múltiples trabajos en lotes en un determinado tiempo, la multiprogramación también se puede utilizar para gestionar múltiples trabajos interactivos. En este último caso, la técnica se denomina **tiempo compartido**, porque se comparte el tiempo de procesador entre múltiples usuarios. En un sistema de tiempo compartido, múltiples usuarios acceden simultáneamente al sistema a través de terminales, siendo el sistema operativo el encargado de entrelazar la ejecución de cada programa de usuario en pequeños intervalos de tiempo o cuantos de computación. Por tanto, si hay  $n$  usuarios activos solicitando un servicio a la vez, cada usuario sólo verá en media  $1/n$  de la capacidad de computación efectiva, sin contar la sobrecarga introducida por el sistema operativo. Sin embargo, dado el tiempo de reacción relativamente lento de los humanos, el tiempo de respuesta de un sistema diseñado adecuadamente debería ser similar al de un computador dedicado.

Ambos tipos de procesamiento, en lotes y tiempo compartido, utilizan multiprogramación. Las diferencias más importantes se listan en la Tabla 2.3.

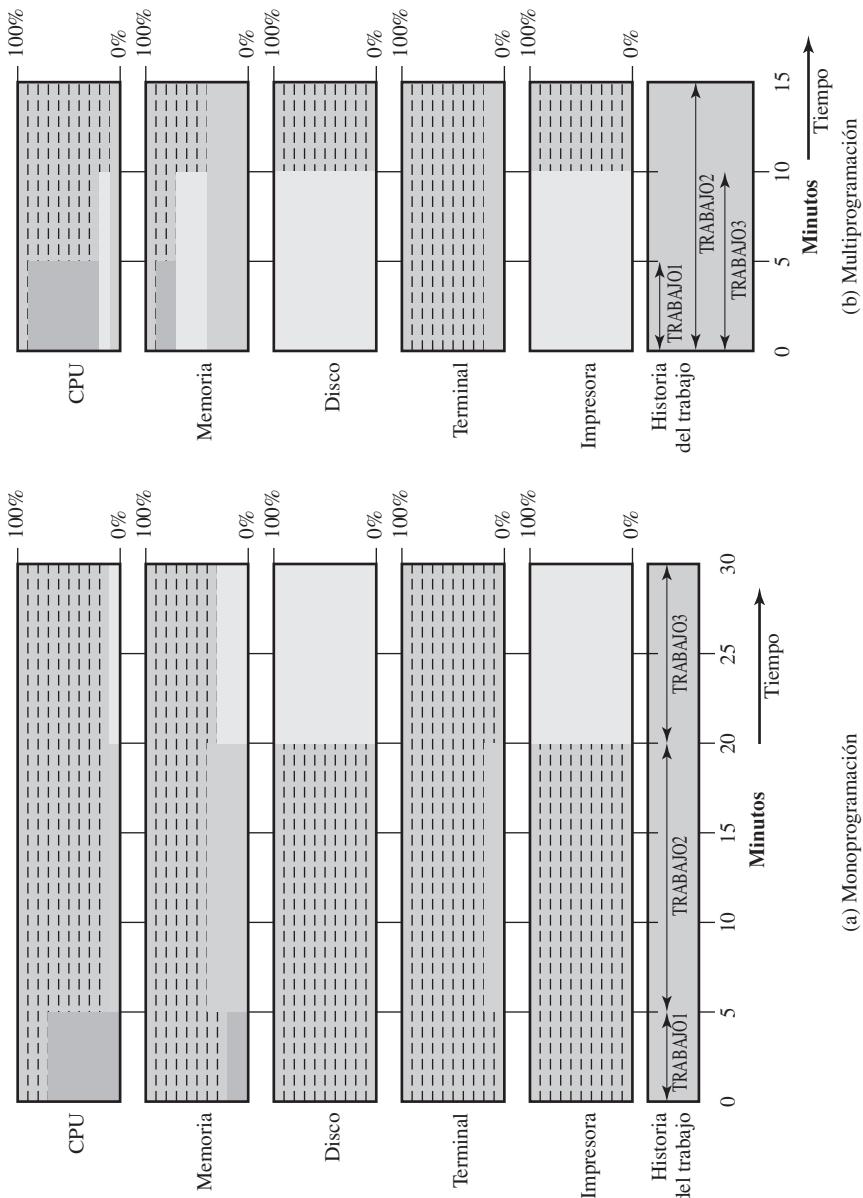


Figura 2.6. Histogramas de utilización.

**Tabla 2.3.** Multiprogramación en lotes frente a tiempo compartido.

	<b>Multiprogramación en lotes</b>	<b>Tiempo compartido</b>
<b>Objetivo principal</b>	Maximizar el uso del procesador	Minimizar el tiempo de respuesta
<b>Fuente de directivas al sistema operativo</b>	Mandatos del lenguaje de control de trabajos proporcionados por el trabajo	Mandatos introducidos al terminal

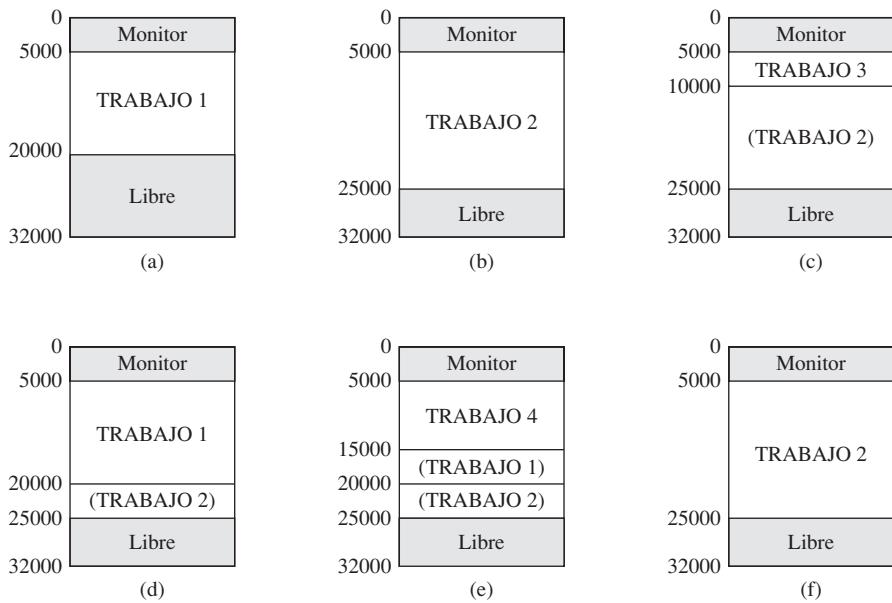
Uno de los primeros sistemas operativos de tiempo compartido desarrollados fue el sistema CTSS (*Compatible Time-Sharing System*) [CORB62], desarrollado en el MIT por un grupo conocido como Proyecto MAC (*Machine-Aided Cognition*, o *Multiple-Access Computers*). El sistema fue inicialmente desarrollado para el IBM 709 en 1961 y más tarde transferido al IBM 7094.

Comparado con sistemas posteriores, CTSS es primitivo. El sistema ejecutó en una máquina con memoria principal con 32.000 palabras de 36 bits, con el monitor residente ocupando 5000 palabras. Cuando el control se asignaba a un usuario interactivo, el programa de usuario y los datos se cargaban en las restantes 27.000 palabras de memoria principal. Para arrancar, un programa siempre se cargaba al comienzo de la palabra 5000; esto simplificaba tanto el monitor como la gestión de memoria. Un reloj del sistema generaba una interrupción cada 0,2 segundos aproximadamente. En cada interrupción de reloj, el sistema operativo retomaba el control y podía asignar el procesador a otro usuario. Por tanto, a intervalos regulares de tiempo, el usuario actual podría ser desalojado y otro usuario puesto a ejecutar. Para preservar el estado del programa de usuario antiguo, los programas de usuario y los datos se escriben en el disco antes de que se lean los nuevos programas de usuario y nuevos datos. Posteriormente, el código y los datos del programa de usuario antiguo se restauran en memoria principal cuando dicho programa vuelve a ser planificado.

Para minimizar el tráfico de disco, la memoria de usuario sólo es escrita a disco cuando el programa entrante la sobreescribe. Este principio queda ilustrado en la Figura 2.7. Sean cuatro usuarios interactivos con los siguientes requisitos de memoria:

- TRABAJO1: 15.000
- TRABAJO2: 20.000
- TRABAJO3: 5000
- TRABAJO4: 10.000

Inicialmente, el monitor carga el trabajo TRABAJO1 y le transfiere control (a). Despues, el monitor decide transferir el control al trabajo TRABAJO2. Debido a que el TRABAJO2 requiere más memoria que el TRABAJO1, se debe escribir primero el TRABAJO1 en disco, y a continuación debe cargarse el TRABAJO2 (b). A continuación, se debe cargar el TRABAJO3 para ejecutarse. Sin embargo, debido a que el TRABAJO3 es más pequeño que el TRABAJO2, una porción de este último queda en memoria, reduciendo el tiempo de escritura de disco (c). Posteriormente, el monitor decide transferir el control de nuevo al TRABAJO1. Una porción adicional de TRABAJO2 debe escribirse en disco cuando se carga de nuevo el TRABAJO1 en memoria (d). Cuando se carga el TRABAJO4, parte del trabajo TRABAJO1 y la porción de TRABAJO2 permanecen en memoria (e). En este punto, si cualquiera de estos trabajos (TRABAJO1 o TRABAJO2) son activados, sólo se requiere una carga parcial. En este ejemplo, es el TRABAJO2 el que ejecuta de nuevo. Esto requiere que el TRABAJO4 y la porción residente de el TRABAJO1 se escriban en el disco y la parte que falta de el TRABAJO2 se lea (f).



**Figura 2.7.** Operación del CTSS.

La técnica utilizada por CTSS es primitiva comparada con las técnicas de tiempo compartido actuales, pero funcionaba. Era extremadamente sencilla, lo que minimizaba el tamaño del monitor. Debido a que un trabajo siempre se cargaba en la misma dirección de memoria, no había necesidad de utilizar técnicas de reubicación en tiempo de carga (que se discutirán más adelante). El hecho de sólo escribir en disco cuando es necesario, minimiza la actividad del disco. Ejecutando sobre el 7094, CTSS permitía un número máximo de 32 usuarios.

La compartición de tiempo y la multiprogramación implican nuevos problemas para el sistema operativo. Si existen múltiples trabajos en memoria, éstos deben protegerse para evitar que interfieran entre sí, por ejemplo, a través de la modificación de los datos de los mismos. Con múltiples usuarios interactivos, el sistema de ficheros debe ser protegido, de forma que sólo usuarios autorizados tengan acceso a un fichero particular. También debe gestionarse los conflictos entre los recursos, tal como impresoras y dispositivos de almacenamiento masivo. Éstos y otros problemas, con sus posibles soluciones, se describirán a lo largo de este libro.

### 2.3. PRINCIPALES LOGROS

Los sistemas operativos se encuentran entre las piezas de software más complejas jamás desarrolladas. Esto refleja el reto de intentar resolver la dificultad de alcanzar determinados objetivos, algunas veces conflictivos, de conveniencia, eficiencia y capacidad de evolución. [DENN80a] propone cinco principales avances teóricos en el desarrollo de los sistemas operativos:

- Procesos.
- Gestión de memoria.
- Protección y seguridad de la información.
- Planificación y gestión de los recursos.
- Estructura del sistema.

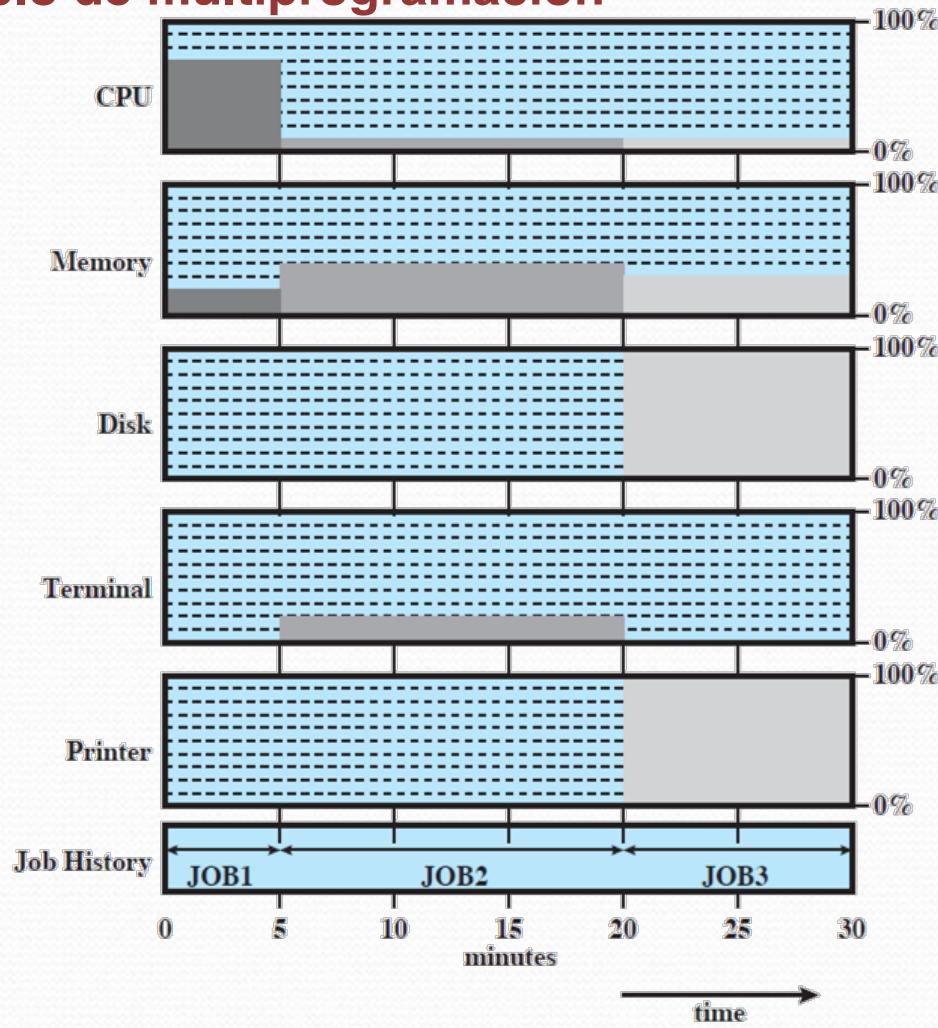
### 2.1.1 Concepto de multiprogramación

	Trabajo 1	Trabajo 2	Trabajo 3
<b>Tipo de trabajo</b>	Computación pesada	Gran cantidad de E/S	Gran cantidad de E/S
<b>Duración</b>	5 minutos	15 minutos	10 minutos
<b>Memoria requerida</b>	50 MB	100 MB	75 MB
<b>¿Necesita disco?</b>	NO	NO	SI
<b>¿Necesita terminal?</b>	NO	SI	NO
<b>¿Necesita impresora?</b>	NO	NO	SI

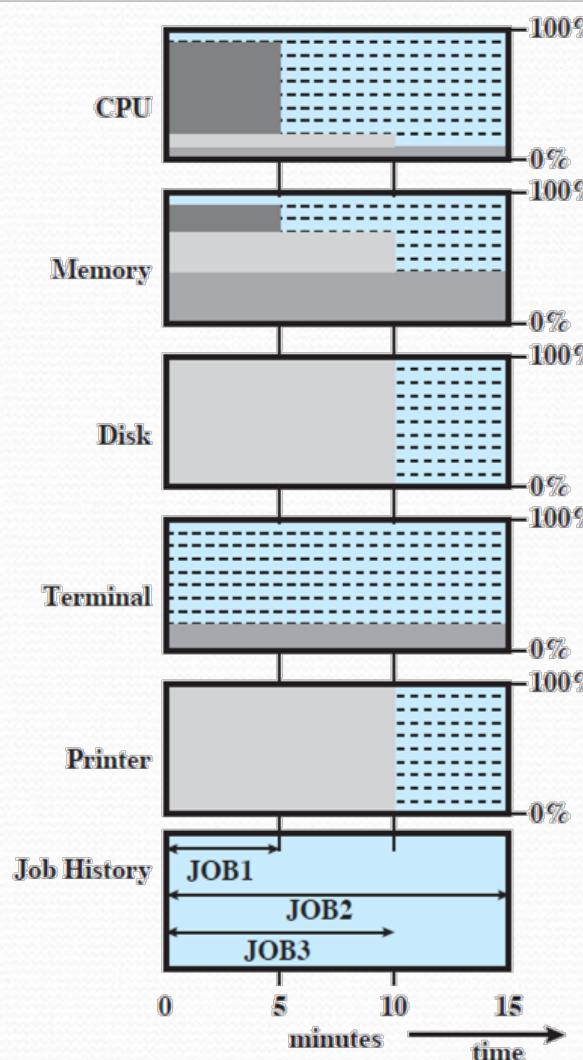
- Trabajo1 utiliza mucho la CPU, Trabajo2 y Trabajo3 utilizan mucho los periféricos de E/S.

## 2.1 Componentes de un SO multiprogramado

### Ejemplo de multiprogramación



(a) Uniprogramming



(b) Multiprogramming

### Ejemplo de multiprogramación

	Monoprogramación	Multiprogramación
<b>Uso del procesador</b>	20%	40%
<b>Uso de memoria</b>	33%	67%
<b>Uso de disco</b>	33%	67%
<b>Uso de impresora</b>	33%	67%
<b>Tiempo transcurrido</b>	30 minutos	15 minutos
<b>Productividad</b>	6 trabajos/hora	12 trabajos/hora
<b>Tiempo de respuesta medio</b>	18 minutos	10 minutos

### 2.1.2 Concepto de proceso [Stal05] (pp. 68-71)

- Un programa en ejecución.
- Una **instancia de un programa** ejecutándose en un ordenador
- La entidad que se puede asignar o ejecutar en un procesador.
- Una **unidad de actividad** caracterizada por **un solo flujo de ejecución**, un **estado actual** y un **conjunto de recursos** del sistema asociados.
- Un **proceso** está formado por:
  - ❑ **Un programa ejecutable.**
  - ❑ **Datos** que necesita el **SO** para **ejecutar el programa.**

Cada avance se caracteriza por principios, o abstracciones, que se han desarrollado para resolver problemas prácticos. Tomadas de forma conjunta, estas cinco áreas incluyen la mayoría de los aspectos clave de diseño e implementación de los sistemas operativos modernos. La breve revisión de estas cinco áreas en esta sección sirve como una visión global de gran parte del resto del libro.

## PROCESOS

El concepto de proceso es fundamental en la estructura de los sistemas operativos. Este término fue utilizado por primera vez por los diseñadores del sistema Multics en los años 60 [DALE68]. Es un término un poco más general que el de trabajo. Se han dado muchas definiciones del término *proceso*, incluyendo:

- Un programa en ejecución.
- Una instancia de un programa ejecutándose en un computador.
- La entidad que se puede asignar o ejecutar en un procesador.
- Una unidad de actividad caracterizada por un solo hilo secuencial de ejecución, un estado actual, y un conjunto de recursos del sistema asociados.

Este concepto se aclarará a lo largo del texto.

Tres líneas principales de desarrollo del sistema de computación crearon problemas de temporización y sincronización que contribuyeron al desarrollo del concepto de proceso: operación en lotes multiprogramados, tiempo compartido, y sistemas de transacciones de tiempo real. Como ya se ha visto, la multiprogramación se diseñó para permitir el uso simultáneo del procesador y los dispositivos de E/S, incluyendo los dispositivos de almacenamiento, para alcanzar la máxima eficiencia. El mecanismo clave es éste: en respuesta a las señales que indican la finalización de las transacciones de E/S, el procesador es planificado para los diferentes programas que residen en memoria principal.

Una segunda línea de desarrollo fue el tiempo compartido de propósito general. En este caso, el objetivo clave de diseño es responder a las necesidades del usuario y, debido a razones económicas, ser capaz de soportar muchos usuarios simultáneamente. Estos objetivos son compatibles debido al tiempo de reacción relativamente lento del usuario. Por ejemplo, si un usuario típico necesita una media de 2 segundos de tiempo de procesamiento por minuto, entonces una cantidad de 30 usuarios aproximadamente podría compartir el mismo sistema sin interferencias notables. Por supuesto, la sobrecarga del sistema debe tenerse en cuenta para realizar estos cálculos.

Otra línea importante de desarrollo han sido los sistemas de procesamiento de transacciones de tiempo real. En este caso, un cierto número de usuarios realizan consultas o actualizaciones sobre una base de datos. Un ejemplo es un sistema de reserva para una compañía aérea. La principal diferencia entre el sistema de procesamiento de transacciones y el sistema de tiempo real es que el primero está limitado a una o unas pocas aplicaciones, mientras que los usuarios de un sistema de tiempo real pueden estar comprometidos en el desarrollo de programas, la ejecución de trabajos, y el uso de varias aplicaciones. En ambos casos, el tiempo de respuesta del sistema es impresionante.

La principal herramienta disponible para programadores de sistema para el desarrollo de la inicial multiprogramación y los sistemas interactivos multiusuario fue la interrupción. Cualquier trabajo podía suspender su actividad por la ocurrencia de un evento definido, tal como la finalización de una operación de E/S. El procesador guardaría alguna forma de contexto (por ejemplo, el contador de programa y otros registros) y saltaría a una rutina de tratamiento de interrupciones, que determinaría la naturaleza de la interrupción, procesaría la interrupción, y después continuaría el procesamiento de usuario con el trabajo interrumpido o algún otro trabajo.

El diseño del software del sistema para coordinar estas diversas actividades resultó ser notablemente difícil. Con la progresión simultánea de muchos trabajos, cada uno de los cuales suponía la realización de numerosos pasos para su ejecución secuencial, era imposible analizar todas las posibles combinaciones de secuencias de eventos. Con la ausencia de algún método sistemático de coordinación y cooperación entre las actividades, los programadores acudían a métodos «*ad hoc*» basados en la comprensión del entorno que el sistema operativo tenía que controlar. Estos esfuerzos eran vulnerables frente a errores de programación sutiles, cuyos efectos sólo podían observarse cuando ciertas extrañas secuencias de acciones ocurrían. Estos errores eran difíciles de diagnosticar, porque necesitaban distinguirse de los errores software y hardware de las aplicaciones. Incluso cuando se detectaba el error, era difícil determinar la causa, porque las condiciones precisas bajo las cuales el error aparecía, eran difíciles de reproducir. En términos generales, existen cuatro causas principales de dichos errores [DEBB80a]:

- **Inapropiada sincronización.** Es frecuente el hecho de que una rutina se suspenda esperando por algún evento en el sistema. Por ejemplo, un programa que inicia una lectura de E/S debe esperar hasta que los datos estén disponibles en un *buffer* antes de proceder. En este caso, se necesita una señal procedente de otra rutina. El diseño inapropiado del mecanismo de señalización puede provocar que las señales se pierdan o se reciban señales duplicadas.
- **Violación de la exclusión mutua.** Frecuentemente, más de un programa o usuario intentan hacer uso de recursos compartidos simultáneamente. Por ejemplo, dos usuarios podrían intentar editar el mismo fichero a la vez. Si estos accesos no se controlan, podría ocurrir un error. Debe existir algún tipo de mecanismo de exclusión mutua que permita que sólo una rutina en un momento determinado actualice un fichero. Es difícil verificar que la implementación de la exclusión mutua es correcta en todas las posibles secuencias de eventos.
- **Operación no determinista de un programa.** Los resultados de un programa particular normalmente dependen sólo de la entrada a dicho programa y no de las actividades de otro programa en un sistema compartido. Pero cuando los programas comparten memoria, y sus ejecuciones son entrelazadas por el procesador, podrían interferir entre ellos, sobre escribiendo zonas de memoria comunes de una forma impredecible. Por tanto, el orden en el que diversos programas se planifican puede afectar a la salida de cualquier programa particular.
- **Interbloqueos.** Es posible que dos o más programas se queden bloqueados esperándose entre sí. Por ejemplo, dos programas podrían requerir dos dispositivos de E/S para llevar a cabo una determinada operación (por ejemplo, una copia de un disco o una cinta). Uno de los programas ha tomado control de uno de los dispositivos y el otro programa tiene control del otro dispositivo. Cada uno de ellos está esperando a que el otro programa libere el recurso que no poseen. Dicho interbloqueo puede depender de la temporización de la asignación y liberación de recursos.

Lo que se necesita para enfrentarse a estos problemas es una forma sistemática de monitorizar y controlar la ejecución de varios programas en el procesador. El concepto de proceso proporciona los fundamentos. Se puede considerar que un proceso está formado por los siguientes tres componentes:

- Un programa ejecutable.
- Los datos asociados que necesita el programa (variables, espacio de trabajo, *buffers*, etc.).
- El contexto de ejecución del programa.

Este último elemento es esencial. El **contexto de ejecución**, o **estado del proceso**, es el conjunto de datos interno por el cual el sistema operativo es capaz de supervisar y controlar el proceso. Esta información interna está separada del proceso, porque el sistema operativo tiene información a la que

el proceso no puede acceder. El contexto incluye toda la información que el sistema operativo necesita para gestionar el proceso y que el procesador necesita para ejecutar el proceso apropiadamente. El contexto incluye el contenido de diversos registros del procesador, tales como el contador de programa y los registros de datos. También incluye información de uso del sistema operativo, como la prioridad del proceso y si un proceso está esperando por la finalización de un evento de E/S particular.

La Figura 2.8 indica una forma en la cual los procesos pueden gestionarse. Dos procesos, A y B, se encuentran en una porción de memoria principal. Es decir, se ha asignado un bloque de memoria a cada proceso, que contiene el programa, datos e información de contexto. Se incluye a cada proceso en una lista de procesos que construye y mantiene el sistema operativo. La lista de procesos contiene una entrada por cada proceso, e incluye un puntero a la ubicación del bloque de memoria que contiene el proceso. La entrada podría también incluir parte o todo el contexto de ejecución del proceso. El resto del contexto de ejecución es almacenado en otro lugar, tal vez junto al propio proceso (como queda reflejado en la Figura 2.8) o frecuentemente en una región de memoria separada. El registro índice del proceso contiene el índice del proceso que el procesador está actualmente controlando en la lista de procesos. El contador de programa apunta a la siguiente instrucción del proceso que se va a ejecutar. Los registros base y límite definen la región de memoria ocupada por el proceso: el registro base contiene la dirección inicial de la región de memoria y el registro límite el tamaño de la región (en bytes o palabras). El contador de programa y todas las referencias de datos se interpretan de for-

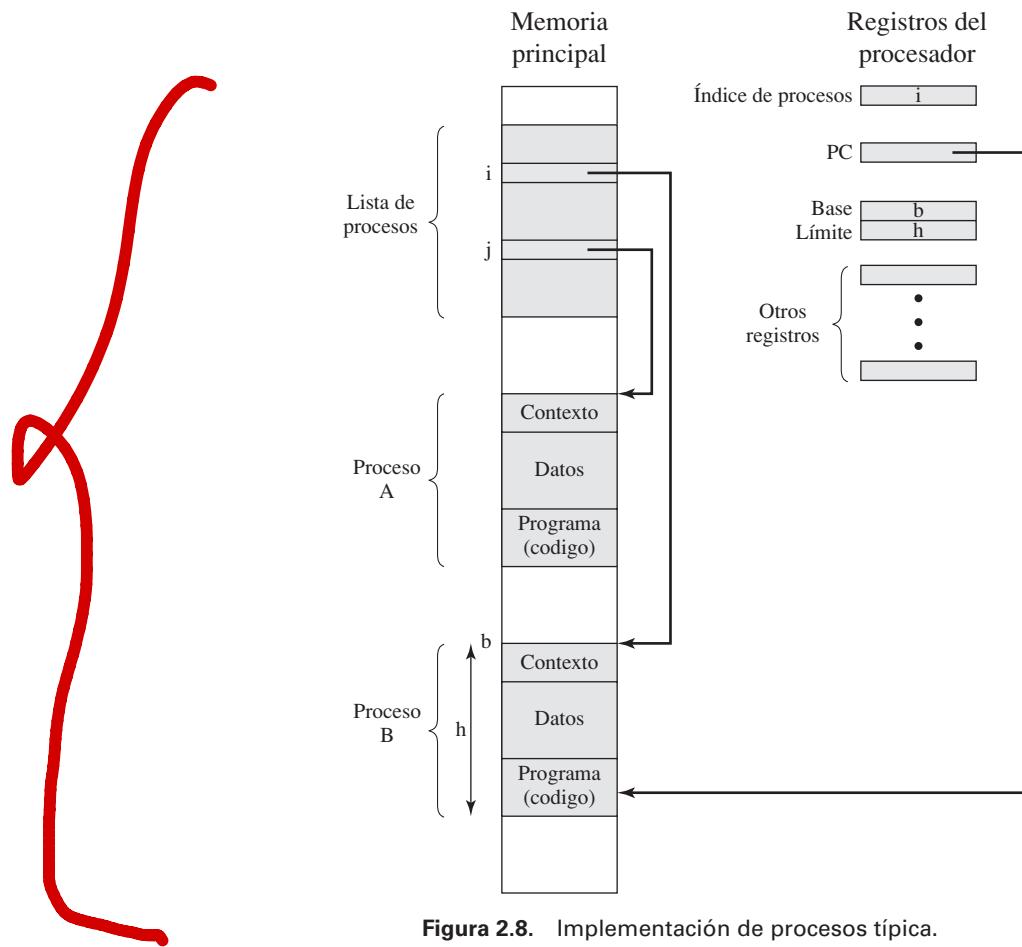


Figura 2.8. Implementación de procesos típica.

ma relativa al registro base y no deben exceder el valor almacenado en el registro límite. Esto previene la interferencia entre los procesos.

En la Figura 2.8, el registro índice del proceso indica que el proceso B está ejecutando. El proceso A estaba ejecutando previamente, pero fue interrumpido temporalmente. Los contenidos de todos los registros en el momento de la interrupción de A fueron guardados en su contexto de ejecución. Posteriormente, el sistema operativo puede cambiar el proceso en ejecución y continuar la ejecución del contexto de A. Cuando se carga el contador de programa con un valor que apunta al área de programa de A, el proceso A continuará la ejecución automáticamente.

Por tanto, el proceso puede verse como una estructura de datos. Un proceso puede estar en ejecución o esperando ejecutarse. El **estado** completo del proceso en un instante dado se contiene en su contexto. Esta estructura permite el desarrollo de técnicas potentes que aseguran la coordinación y la cooperación entre los procesos. Se pueden diseñar e incorporar nuevas características en el sistema operativo (por ejemplo, la prioridad), expandiendo el contexto para incluir cualquier información nueva que se utilice para dar soporte a dicha característica. A lo largo del libro, veremos un gran número de ejemplos donde se utiliza esta estructura de proceso para resolver los problemas provocados por la multiprogramación o la compartición de recursos.

## GESTIÓN DE MEMORIA

Un entorno de computación que permita programación modular y el uso flexible de los datos puede ayudar a resolver mejor las necesidades de los usuarios. Los gestores de sistema necesitan un control eficiente y ordenado de la asignación de los recursos. Para satisfacer estos requisitos, el sistema operativo tiene cinco responsabilidades principales de gestión de almacenamiento:

- **Aislamiento de procesos.** El sistema operativo debe evitar que los procesos independientes interfieran en la memoria de otro proceso, tanto datos como instrucciones.
- **Asignación y gestión automática.** Los programas deben tener una asignación dinámica de memoria por demanda, en cualquier nivel de la jerarquía de memoria. La asignación debe ser transparente al programador. Por tanto, el programador no debe preocuparse de aspectos relacionados con limitaciones de memoria, y el sistema operativo puede lograr incrementar la eficiencia, asignando memoria a los trabajos sólo cuando se necesiten.
- **Soporte a la programación modular.** Los programadores deben ser capaces de definir módulos de programación y crear, destruir, y alterar el tamaño de los módulos dinámicamente.
- **Protección y control de acceso.** La compartición de memoria, en cualquier nivel de la jerarquía de memoria, permite que un programa dirija un espacio de memoria de otro proceso. Esto es deseable cuando se necesita la compartición por parte de determinadas aplicaciones. Otras veces, esta característica amenaza la integridad de los programas e incluso del propio sistema operativo. El sistema operativo debe permitir que varios usuarios puedan acceder de distintas formas a porciones de memoria.
- **Almacenamiento a largo plazo.** Muchas aplicaciones requieren formas de almacenar la información durante largos períodos de tiempo, después de que el computador se haya apagado.

Normalmente, los sistemas operativos alcanzan estos requisitos a través del uso de la memoria virtual y las utilidades de los sistemas operativos. El sistema operativo implementa un almacenamiento a largo plazo, con la información almacenada en objetos denominados ficheros. El fichero es un concepto lógico, conveniente para el programador y es una unidad útil de control de acceso y protección para los sistemas operativos.

### Bloque de Control de Proceso (PCB, Process Control Block)

[Stall05] (pp. 108-120)

- **Identificador de proceso**, (PID, del inglés Process Identifier)
- **Contexto de ejecución**: Registros del procesador
- **Memoria** donde reside el programa
- **Información** relacionada con recursos del sistema
- **Estado**. En que situación se encuentra el proceso en cada momento (modelo de estados)
- **Más información**

Identificador
Estado
Prioridad
Contador de programa
Punteros de memoria
Datos de contexto
Información de estado de E/S
Información de auditoría
.
.
.

El diseño de un sistema operativo debe reflejar ciertos requisitos generales. Todos los sistemas operativos multiprogramados, desde los sistemas operativos monousuario como Windows 98 hasta sistemas *mainframes* como IBM z/OS, que son capaces de dar soporte a miles de usuarios, se construyen en torno al concepto de proceso. La mayoría de los requisitos que un sistema operativo debe cumplir se pueden expresar con referencia a los procesos:

- El sistema operativo debe intercalar la ejecución de múltiples procesos, para maximizar la utilización del procesador mientras se proporciona un tiempo de respuesta razonable.
- El sistema operativo debe reservar recursos para los procesos conforme a una política específica (por ejemplo, ciertas funciones o aplicaciones son de mayor prioridad) mientras que al mismo tiempo evita interbloqueos<sup>1</sup>.
- Un sistema operativo puede requerir dar soporte a la comunicación entre procesos y la creación de procesos, mediante las cuales ayuda a la estructuración de las aplicaciones.

Se comienza el estudio detallado de los sistemas operativos examinando la forma en la que éstos representan y controlan los procesos. Después de una introducción al concepto de proceso, el capítulo presentará los estados de los procesos, que caracterizan el comportamiento de los mismos. Seguidamente, se presentarán las estructuras de datos que el sistema operativo usa para gestionar los procesos. Éstas incluyen las estructuras para representar el estado de cada proceso así como para registrar características de los mismos que el sistema operativo necesita para alcanzar sus objetivos. Posteriormente, se verá cómo el sistema operativo utiliza estas estructuras para controlar la ejecución de los procesos. Por último, se discute la gestión de procesos en UNIX SVR4. El Capítulo 4 proporciona ejemplos más modernos de gestión de procesos, tales como Solaris, Windows, y Linux.

Nota: en este capítulo hay referencias puntuales a la memoria virtual. La mayoría de las veces podemos ignorar este concepto en relación con los procesos, pero en ciertos puntos de la discusión, las consideraciones sobre memoria virtual se hacen pertinentes. La memoria virtual no se discutirá en detalle hasta el Capítulo 8; ya se ha proporcionado una somera visión general en el Capítulo 2.

### 3.1. ¿QUÉ ES UN PROCESO?

#### CONCEPTOS PREVIOS

Antes de definir el término proceso, es útil recapitular algunos de los conceptos ya presentados en los Capítulos 1 y 2:

1. Una plataforma de computación consiste en una colección de recursos hardware, como procesador, memoria, módulos de E/S, relojes, unidades de disco y similares.
2. Las aplicaciones para computadores se desarrollan para realizar determinadas tareas. Suelen aceptar entradas del mundo exterior, realizar algún procesamiento y generar salidas.
3. No es eficiente que las aplicaciones estén escritas directamente para una plataforma hardware específica. Las principales razones son las siguientes:

---

<sup>1</sup> Los interbloqueos se examinarán en el Capítulo 6. Como ejemplo sencillo, un interbloqueo ocurre si dos procesos necesitan dos recursos iguales para continuar y cada uno de los procesos tiene la posesión de uno de los recursos. A menos que se realice alguna acción, cada proceso esperará indefinidamente por conseguir el otro recurso.

- a) Numerosas aplicaciones pueden desarrollarse para la misma plataforma, de forma que tiene sentido desarrollar rutinas comunes para acceder a los recursos del computador.
  - b) El procesador por sí mismo proporciona únicamente soporte muy limitado para la multi-programación. Es necesario disponer de software para gestionar la compartición del procesador así como otros recursos por parte de múltiples aplicaciones al mismo tiempo.
  - c) Cuando múltiples aplicaciones están activas al mismo tiempo es necesario proteger los datos, el uso de la E/S y los recursos propios de cada aplicación con respecto a las demás.
4. El sistema operativo se desarrolló para proporcionar una interfaz apropiada para las aplicaciones, rica en funcionalidades, segura y consistente. El sistema operativo es una capa de software entre las aplicaciones y el hardware del computador (Figura 2.1) que da soporte a aplicaciones y utilidades.
  5. Se puede considerar que el sistema operativo proporciona una representación uniforme y abstracta de los recursos, que las aplicaciones pueden solicitar y acceder. Los recursos incluyen la memoria principal, las interfaces de red, los sistemas de ficheros, etc. Una vez que el sistema operativo ha creado estas abstracciones de los recursos para que las aplicaciones las usen, debe también controlar su uso. Por ejemplo, un sistema operativo podría permitir compartición y protección de recursos.

Ahora que se conocen los conceptos de aplicaciones, software de sistema y de recursos se está en disposición de hablar sobre cómo un sistema operativo puede, de forma ordenada, gestionar la ejecución de aplicaciones de forma que:

- Los recursos estén disponibles para múltiples aplicaciones.
- El procesador físico se commute entre múltiples aplicaciones, de forma que todas lleguen a procesarse.
- El procesador y los dispositivos de E/S se puedan usar de forma eficiente.

El enfoque adoptado por todos los sistemas operativos modernos recae en un modelo bajo el cual la ejecución de una aplicación se corresponde con la existencia de uno o más procesos.

## PROCESOS Y BLOQUES DE CONTROL DE PROCESOS

Se debe recordar que en el Capítulo 2 se sugirieron diversas definiciones del término *proceso*, incluyendo:

- Un programa en ejecución.
- Una instancia de un programa ejecutado en un computador.
- La entidad que se puede asignar y ejecutar en un procesador.
- Una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados.

También se puede pensar en un proceso como en una entidad que consiste en un número de elementos. Los dos elementos esenciales serían el **código de programa** (que puede compartirse con otros procesos que estén ejecutando el mismo programa) y un **conjunto de datos** asociados a dicho

código. Supongamos que el procesador comienza a ejecutar este código de programa, y que nos referiremos a esta entidad en ejecución como un proceso. En cualquier instante puntual del tiempo, *mientras el proceso está en ejecución*, este proceso se puede caracterizar por una serie de elementos, incluyendo los siguientes:

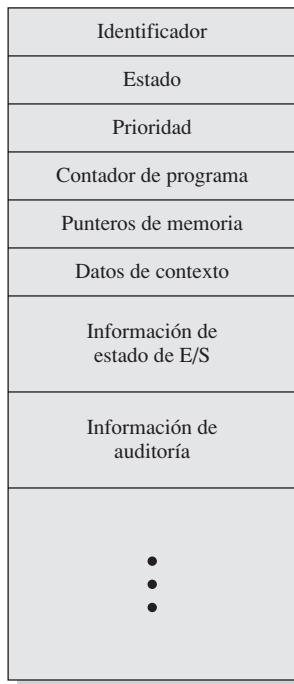
- **Identificador.** Un identificador único asociado a este proceso, para distinguirlo del resto de procesos.
- **Estado.** Si el proceso está actualmente corriendo, está en el estado *en ejecución*.
- **Prioridad:** Nivel de prioridad relativo al resto de procesos.
- **Contador de programa.** La dirección de la siguiente instrucción del programa que se ejecutará.
- **Punteros a memoria.** Incluye los punteros al código de programa y los datos asociados a dicho proceso, además de cualquier bloque de memoria compartido con otros procesos.
- **Datos de contexto.** Estos son datos que están presentes en los registros del procesador cuando el proceso está corriendo.
- **Información de estado de E/S.** Incluye las peticiones de E/S pendientes, dispositivos de E/S (por ejemplo, una unidad de cinta) asignados a dicho proceso, una lista de los ficheros en uso por el mismo, etc.
- **Información de auditoría.** Puede incluir la cantidad de tiempo de procesador y de tiempo de reloj utilizados, así como los límites de tiempo, registros contables, etc.

La información de la lista anterior se almacena en una estructura de datos, que se suele llamar bloque de control de proceso (*process control block*) (Figura 3.1), que el sistema operativo crea y gestiona. El punto más significativo en relación al bloque de control de proceso, o BCP, es que contiene suficiente información de forma que es posible interrumpir el proceso cuando está corriendo y posteriormente restaurar su estado de ejecución como si no hubiera habido interrupción alguna. El BCP es la herramienta clave que permite al sistema operativo dar soporte a múltiples procesos y proporcionar multiprogramación. Cuando un proceso se interrumpe, los valores actuales del contador de programa y los registros del procesador (datos de contexto) se guardan en los campos correspondientes del BCP y el estado del proceso se cambia a cualquier otro valor, como *bloqueado* o *listo* (descriptos a continuación). El sistema operativo es libre ahora para poner otro proceso en estado de ejecución. El contador de programa y los datos de contexto se recuperan y cargan en los registros del procesador y este proceso comienza a correr.

De esta forma, se puede decir que un proceso está compuesto del código de programa y los datos asociados, además del bloque de control de proceso o BCP. Para un computador monoprocesador, en un instante determinado, como máximo un único proceso puede estar corriendo y dicho proceso estará en el estado *en ejecución*.

### 3.2. ESTADOS DE LOS PROCESOS

Como se acaba de comentar, para que un programa se ejecute, se debe crear un proceso o tarea para dicho programa. Desde el punto de vista del procesador, él ejecuta instrucciones de su repertorio de instrucciones en una secuencia dictada por el cambio de los valores del registro contador de programa. A lo largo del tiempo, el contador de programa puede apuntar al código de diferentes programas que son parte de diferentes procesos. Desde el punto de vista de un programa individual, su ejecución implica una secuencia de instrucciones dentro de dicho programa.



**Figura 3.1.** Bloque de control de programa (BCP) simplificado.

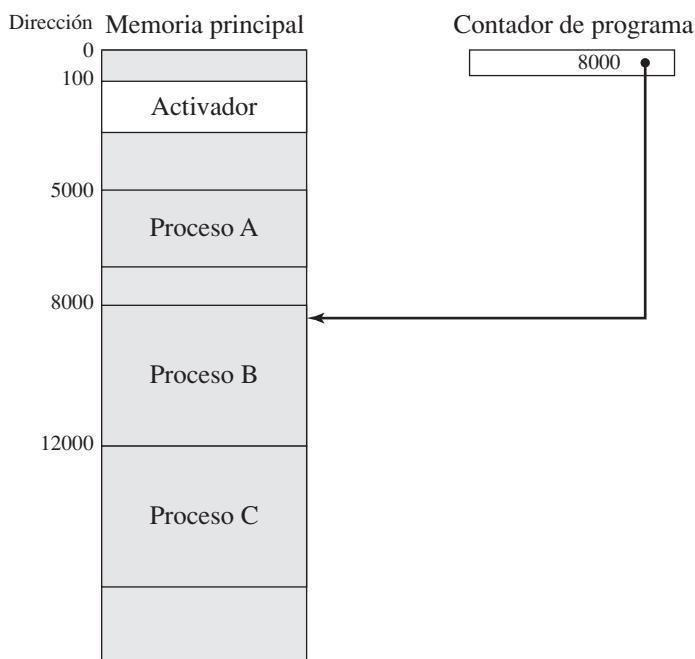
Se puede caracterizar el comportamiento de un determinado proceso, listando la secuencia de instrucciones que se ejecutan para dicho proceso. A esta lista se la denomina **traza** del proceso. Se puede caracterizar el comportamiento de un procesador mostrando cómo las trazas de varios procesos se entrelazan.

Considere un ejemplo. La Figura 3.2 muestra el despliegue en memoria de tres procesos. Para simplificar la exposición, se asume que dichos procesos no usan memoria virtual; por tanto, los tres procesos están representados por programas que residen en memoria principal. De manera adicional, existe un pequeño programa **activador** (*dispatcher*) que intercambia el procesador de un proceso a otro. La Figura 3.3 muestra las trazas de cada uno de los procesos en los primeros instantes de ejecución. Se muestran las 12 primeras instrucciones ejecutadas por los procesos A y C. El proceso B ejecuta 4 instrucciones y se asume que la cuarta instrucción invoca una operación de E/S, a la cual el proceso debe esperar.

Ahora vea estas trazas desde el punto de vista del procesador. La Figura 3.4 muestra las trazas entrelazadas resultante de los 52 primeros ciclos de ejecución (por conveniencia los ciclos de instrucciones han sido numerados). En este ejemplo, se asume que el sistema operativo sólo deja que un proceso continúe durante seis ciclos de instrucción, después de los cuales se interrumpe; lo cual previene que un solo proceso monopolice el uso del tiempo del procesador. Como muestra la Figura 3.4, las primeras seis instrucciones del proceso A se ejecutan seguidas de una alarma de temporización (*time-out*) y de la ejecución de cierto código del activador, que ejecuta seis instrucciones antes de devolver el control al proceso B<sup>2</sup>. Después de que se ejecuten cuatro instrucciones, el proceso B solicita una acción de

---

<sup>2</sup> El reducido número de instrucciones ejecutadas por los procesos y por el planificador es irreal; se ha utilizado para simplificar el ejemplo y clarificar las explicaciones.



**Figura 3.2.** Instantánea de un ejemplo de ejecución (Figura 3.4) en el ciclo de instrucción 13.

5000 5001 5002 5003 5004 5005 5006 5007 5008 5009 5010 5011	8000 8001 8002 8003	12000 12001 12002 12003 12004 12005 12006 12007 12008 12009 12010 12011
(a) Traza del Proceso A	(b) Traza del Proceso B	(c) Traza del Proceso C

5000 = Dirección de comienzo del programa del Proceso A.

8000 = Dirección de comienzo del programa del Proceso B.

12000 = Dirección de comienzo del programa del Proceso C.

**Figura 3.3.** Traza de los procesos de la Figura 3.2.

E/S, para la cual debe esperar. Por tanto, el procesador deja de ejecutar el proceso B y pasa a ejecutar el proceso C, por medio del activador. Después de otra alarma de temporización, el procesador vuelve al proceso A. Cuando este proceso llega a su temporización, el proceso B aún estará esperando que se complete su operación de E/S, por lo que el activador pasa de nuevo al proceso C.

## UN MODELO DE PROCESO DE DOS ESTADOS

La responsabilidad principal del sistema operativo es controlar la ejecución de los procesos; esto incluye determinar el patrón de entrelazado para la ejecución y asignar recursos a los procesos. El primer paso en el diseño de un sistema operativo para el control de procesos es describir el comportamiento que se desea que tengan los procesos.

Se puede construir el modelo más simple posible observando que, en un instante dado, un proceso está siendo ejecutando por el procesador o no. En este modelo, un proceso puede estar en dos estados: Ejecutando o No Ejecutando, como se muestra en la Figura 3.5a. Cuando el sistema operativo crea un nuevo proceso, crea el bloque de control de proceso (BCP) para el nuevo proceso e inserta dicho proceso en el sistema en estado No Ejecutando. El proceso existe, es conocido por el sistema operativo, y está esperando su oportunidad de ejecutar. De cuando en cuando, el proceso actualmente en ejecución se interrumpirá y una parte del sistema operativo, el activador, seleccionará otro proceso

1	5000		27	12004	
2	5001		28	12005	
3	5002				Temporización
4	5003		29	100	
5	5004		30	101	
6	5005		31	102	
		Temporización	32	103	
7	100		33	104	
8	101		34	105	
9	102				35
10	103				5006
11	104				36
12	105				5007
13	8000				37
14	8001				5008
15	8002				38
16	8003				5009
		Petición de E/S	39	5010	
			40	5011	
					41
					100
					42
					101
					43
					102
					44
					103
					45
					104
					46
					105
					47
					12006
					48
					12007
					49
					12008
					50
					12009
					51
					12010
					52
					12011
					Temporización

100 = Dirección de comienzo del programa activador.

Las zonas sombreadas indican la ejecución del proceso de activación;

la primera y la tercera columna cuentan ciclos de instrucciones;

la segunda y la cuarta columna las direcciones de las instrucciones que se ejecutan

**Figura 3.4.** Traza combinada de los procesos de la Figura 3.2.

a ejecutar. El proceso saliente pasará del estado Ejecutando a No Ejecutando y pasará a Ejecutando un nuevo proceso.

De este modelo simple, ya se puede apreciar algo del diseño de los elementos del sistema operativo. Cada proceso debe representarse de tal manera que el sistema operativo pueda seguirle la pista. Es decir, debe haber información correspondiente a cada proceso, incluyendo el estado actual y su localización en memoria; esto es el bloque de control de programa. Los procesos que no están ejecutando deben estar en una especie de cola, esperando su turno de ejecución. La Figura 3.5b sugiere esta estructura. Existe una sola cola cuyas entradas son punteros al BCP de un proceso en particular. Alternativamente, la cola debe consistir en una lista enlazada de bloques de datos, en la cual cada bloque representa un proceso; exploraremos posteriormente esta última implementación.

Podemos describir el comportamiento del activador en términos de este diagrama de colas. Un proceso que se interrumpe se transfiere a la cola de procesos en espera. Alternativamente, si el proceso ha finalizado o ha sido abortado, se descarta (sale del sistema). En cualquier caso, el activador selecciona un proceso de la cola para ejecutar.

## CREACIÓN Y TERMINACIÓN DE PROCESOS

Antes de intentar refinar nuestro sencillo modelo de dos estados, resultará útil hablar de la creación y terminación de los procesos; por último, y de forma independiente del modelo de comportamiento de procesos que se use, la vida de un proceso está acotada entre su creación y su terminación.

**Creación de un proceso.** Cuando se va a añadir un nuevo proceso a aquellos que se están gestionando en un determinado momento, el sistema operativo construye las estructuras de datos que se usan para manejar el proceso (como se describió en la Sección 3.3) y reserva el espacio de direcciones en memoria principal para el proceso. Estas acciones constituyen la creación de un nuevo proceso.

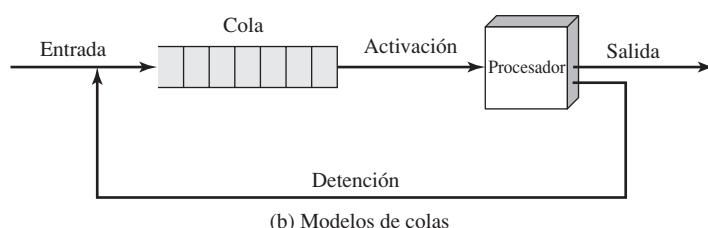
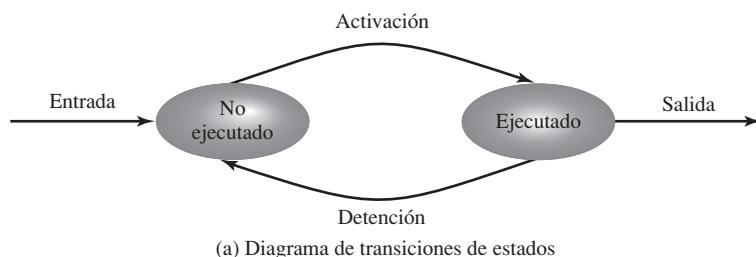


Figura 3.5. Modelo de proceso de dos estados.

Existen cuatro eventos comunes que llevan a la creación de un proceso, como se indica en la Tabla 3.1. En un entorno por lotes, un proceso se crea como respuesta a una solicitud de trabajo. En un entorno interactivo, un proceso se crea cuando un nuevo usuario entra en el sistema. En ambos casos el sistema operativo es responsable de la creación de nuevos procesos. Un sistema operativo puede, a petición de una aplicación, crear procesos. Por ejemplo, si un usuario solicita que se imprima un fichero, el sistema operativo puede crear un proceso que gestione la impresión. El proceso solicitado puede, de esta manera, operar independientemente del tiempo requerido para completar la tarea de impresión.

**TABLA 3.1.** Razones para la creación de un proceso.

Nuevo proceso de lotes	El sistema operativo dispone de un flujo de control de lotes de trabajos, habitualmente una cinta un disco. Cuando el sistema operativo está listo para procesar un nuevo trabajo, leerá la siguiente secuencia de mandatos de control de trabajos.
Sesión interactiva	Un usuario desde un terminal entra en el sistema.
Creado por el sistema operativo para proporcionar un servicio	El sistema operativo puede crear un proceso para realizar una función en representación de un programa de usuario, sin que el usuario tenga que esperar (por ejemplo, un proceso para controlar la impresión).
Creado por un proceso existente	Por motivos de modularidad o para explotar el paralelismo, un programa de usuario puede ordenar la creación de un número de procesos.

Tradicionalmente, un sistema operativo creaba todos los procesos de una forma que era transparente para usuarios y programas. Esto aún es bastante común en muchos sistemas operativos contemporáneos. Sin embargo, puede ser muy útil permitir a un proceso la creación de otro. Por ejemplo, un proceso de aplicación puede generar otro proceso para recibir los datos que la aplicación está generando y para organizar esos datos de una forma apropiada para su posterior análisis. El nuevo proceso ejecuta en paralelo con el proceso original y se activa cuando los nuevos datos están disponibles. Esta organización puede ser muy útil en la estructuración de la aplicación. Otro ejemplo, un proceso servidor (por ejemplo, un servidor de impresoras, servidor de ficheros) puede generar un proceso por cada solicitud que esté manejando. Cuando un sistema operativo crea un proceso a petición explícita de otro proceso, dicha acción se denomina *creación del proceso*.

Cuando un proceso lanza otro, al primero se le denomina **proceso padre**, y al proceso creado se le denomina **proceso hijo**. Habitualmente, la relación entre procesos necesita comunicación y cooperación entre ellos. Alcanzar esta cooperación es una tarea complicada para un programador; este aspecto se verá en el Capítulo 5.

**Terminación de procesos.** La Tabla 3.2 resume las razones típicas para la terminación de un proceso. Todo sistema debe proporcionar los mecanismos mediante los cuales un proceso indica su finalización, o que ha completado su tarea. Un trabajo por lotes debe incluir una instrucción HALT o una llamada a un servicio de sistema operativo específica para su terminación. En el caso anterior, la instrucción HALT generará una interrupción para indicar al sistema operativo que dicho proceso ha finalizado. Para una aplicación interactiva, las acciones del usuario indicarán cuando el proceso ha terminado. Por ejemplo, en un sistema de tiempo compartido, el proceso de un usuario en particular puede terminar cuando el usuario sale del sistema o apaga su terminal. En un ordenador personal o una estación de trabajo, el usuario puede salir de una aplicación (por ejemplo, un procesador de texto o una

hoja de cálculo). Todas estas acciones tienen como resultado final la solicitud de un servicio al sistema operativo para terminar con el proceso solicitante.

Adicionalmente, un número de error o una condición de fallo puede llevar a la finalización de un proceso. La Tabla 3.2 muestra una lista de las condiciones más habituales de finalización por esos motivos<sup>3</sup>.

**Tabla 3.2.** Razones para la terminación de un proceso.

Finalización normal	El proceso ejecuta una llamada al sistema operativo para indicar que ha completado su ejecución
Límite de tiempo excedido	El proceso ha ejecutado más tiempo del especificado en un límite máximo. Existen varias posibilidades para medir dicho tiempo. Estas incluyen el tiempo total utilizado, el tiempo utilizado únicamente en ejecución, y, en el caso de procesos interactivos, la cantidad de tiempo desde que el usuario realizó la última entrada.
Memoria no disponible	El proceso requiere más memoria de la que el sistema puede proporcionar.
Violaciones de frontera	El proceso trata de acceder a una posición de memoria a la cual no tiene acceso permitido.
Error de protección	El proceso trata de usar un recurso, por ejemplo un fichero, al que no tiene permitido acceder, o trata de utilizarlo de una forma no apropiada, por ejemplo, escribiendo en un fichero de sólo lectura.
Error aritmético	El proceso trata de realizar una operación de cálculo no permitida, tal como una división por 0, o trata de almacenar números mayores de los que la representación hardware puede codificar.
Límite de tiempo	El proceso ha esperado más tiempo que el especificado en un valor máximo para que se cumpla un determinado evento.
Fallo de E/S	Se ha producido un error durante una operación de entrada o salida, por ejemplo la imposibilidad de encontrar un fichero, fallo en la lectura o escritura después de un límite máximo de intentos (cuando, por ejemplo, se encuentra un área defectuosa en una cinta), o una operación inválida (la lectura de una impresora en línea).
Instrucción no válida	El proceso intenta ejecutar una instrucción inexistente (habitualmente el resultado de un salto a un área de datos y el intento de ejecutar dichos datos).
Instrucción privilegiada	El proceso intenta utilizar una instrucción reservada al sistema operativo.
Uso inapropiado de datos	Una porción de datos es de tipo erróneo o no se encuentra inicializada.
Intervención del operador por el sistema operativo	Por alguna razón, el operador o el sistema operativo ha finalizado el proceso (por ejemplo, se ha dado una condición de interbloqueo).
Terminación del proceso padre	Cuando un proceso padre termina, el sistema operativo puede automáticamente finalizar todos los procesos hijos descendientes de dicho padre.
Solicitud del proceso padre	Un proceso padre habitualmente tiene autoridad para finalizar sus propios procesos descendientes.

<sup>3</sup> Un sistema operativo compasivo podría en algunos casos permitir al usuario recuperarse de un fallo sin terminar el proceso. Por ejemplo, si un usuario solicita acceder a un fichero y se deniega ese acceso, el sistema operativo podría simplemente informar al usuario de ese hecho y permitir que el proceso continúe.

Por último, en ciertos sistemas operativos, un proceso puede terminarse por parte del proceso que lo creó o cuando dicho proceso padre a su vez ha terminado.

## MODELO DE PROCESO DE CINCO ESTADOS

Si todos los procesos estuviesen siempre preparados para ejecutar, la gestión de colas proporcionada en la Figura 3.5b sería efectiva. La cola es una lista de tipo FIFO y el procesador opera siguiendo una estrategia cíclica (*round-robin* o turno rotatorio) sobre todos los procesos disponibles (cada proceso de la cola tiene cierta cantidad de tiempo, por turnos, para ejecutar y regresar de nuevo a la cola, a menos que se bloquee). Sin embargo, hasta con el sencillo ejemplo que vimos antes, esta implementación es inadecuada: algunos procesos que están en el estado de No Ejecutando están listos para ejecutar, mientras que otros están bloqueados, esperando a que se complete una operación de E/S. Por tanto, utilizando una única cola, el activador no puede seleccionar únicamente los procesos que lleven más tiempo en la cola. En su lugar, debería recorrer la lista buscando los procesos que no estén bloqueados y que lleven en la cola más tiempo.

Una forma más natural para manejar esta situación es dividir el estado de No Ejecutando en dos estados, Listo y Bloqueado. Esto se muestra la Figura 3.6. Para gestionarlo correctamente, se han añadido dos estados adicionales que resultarán muy útiles. Estos cinco estados en el nuevo diagrama son los siguientes:

- **Ejecutando.** El proceso está actualmente en ejecución. Para este capítulo asumimos que el computador tiene un único procesador, de forma que sólo un proceso puede estar en este estado en un instante determinado.
- **Listo.** Un proceso que se prepara para ejecutar cuando tenga oportunidad.
- **Bloqueado.** Un proceso que no puede ejecutar hasta que se cumpla un evento determinado o se complete una operación E/S.
- **Nuevo.** Un proceso que se acaba de crear y que aún no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo. Típicamente, se trata de un nuevo proceso que no ha sido cargado en memoria principal, aunque su bloque de control de proceso (BCP) si ha sido creado.
- **Saliente.** Un proceso que ha sido liberado del grupo de procesos ejecutables por el sistema operativo, debido a que ha sido detenido o que ha sido abortado por alguna razón.

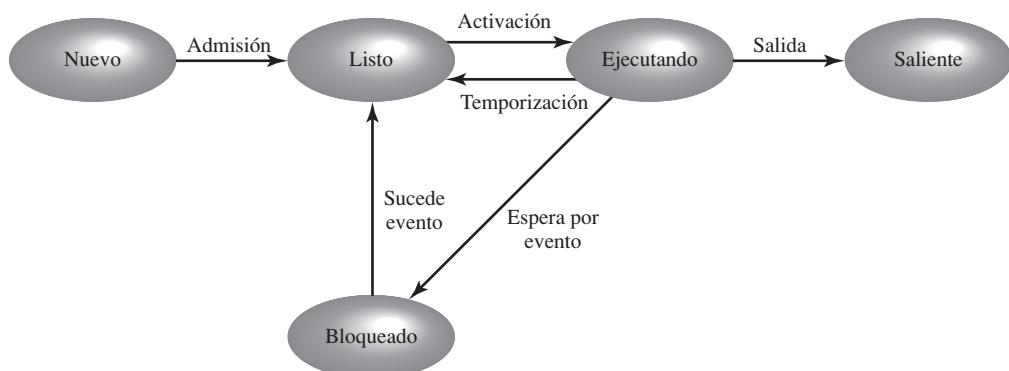


Figura 3.6. Modelo de proceso de cinco estados.

Los estados Nuevo y Saliente son útiles para construir la gestión de procesos. El estado Nuevo se corresponde con un proceso que acaba de ser definido. Por ejemplo, si un nuevo usuario intenta entrar dentro de un sistema de tiempo compartido o cuando se solicita un nuevo trabajo a un sistema de proceso por lotes, el sistema operativo puede definir un nuevo proceso en dos etapas. Primero, el sistema operativo realiza todas las tareas internas que correspondan. Se asocia un identificador a dicho proceso. Se reservan y construyen todas aquellas tablas que se necesiten para gestionar al proceso. En este punto, el proceso se encuentra el estado Nuevo. Esto significa que el sistema operativo ha realizado todas las tareas necesarias para crear el proceso pero el proceso en sí, aún no se ha puesto en ejecución. Por ejemplo, un sistema operativo puede limitar el número de procesos que puede haber en el sistema por razones de rendimiento o limitaciones de memoria principal. Mientras un proceso está en el estado Nuevo, la información relativa al proceso que se necesite por parte del sistema operativo se mantiene en tablas de control de memoria principal. Sin embargo, el proceso en sí mismo no se encuentra en memoria principal. Esto es, el código de programa a ejecutar no se encuentra en memoria principal, y no se ha reservado ningún espacio para los datos asociados al programa. Cuando un proceso se encuentra en el estado Nuevo, el programa permanece en almacenamiento secundario, normalmente en disco<sup>4</sup>.

De forma similar, un proceso sale del sistema en dos fases. Primero, el proceso termina cuando alcanza su punto de finalización natural, cuando es abortado debido a un error no recuperable, o cuando otro proceso con autoridad apropiada causa que el proceso se aborte. La terminación mueve el proceso al estado Saliente. En este punto, el proceso no es elegible de nuevo para su ejecución. Las tablas y otra información asociada con el trabajo se encuentran temporalmente preservadas por el sistema operativo, el cual proporciona tiempo para que programas auxiliares o de soporte extraigan la información necesaria. Por ejemplo, un programa de auditoría puede requerir registrar el tiempo de proceso y otros recursos utilizados por este proceso saliente con objeto de realizar una contabilidad de los recursos del sistema. Un programa de utilidad puede requerir extraer información sobre el histórico de los procesos por temas relativos con el rendimiento o análisis de la utilización. Una vez que estos programas han extraído la información necesaria, el sistema operativo no necesita mantener ningún dato relativo al proceso y el proceso se borra del sistema.

La Figura 3.6 indica que tipos de eventos llevan a cada transición de estado para cada proceso; las posibles transiciones son las siguientes:

- **Null → Nuevo.** Se crea un nuevo proceso para ejecutar un programa. Este evento ocurre por cualquiera de las relaciones indicadas en la tabla 3.1.
- **Nuevo → Listo.** El sistema operativo mueve a un proceso del estado nuevo al estado listo cuando éste se encuentre preparado para ejecutar un nuevo proceso. La mayoría de sistemas fijan un límite basado en el número de procesos existentes o la cantidad de memoria virtual que se podrá utilizar por parte de los procesos existentes. Este límite asegura que no haya demasiados procesos activos y que se degrade el rendimiento sistema.
- **Listo → Ejecutando.** Cuando llega el momento de seleccionar un nuevo proceso para ejecutar, el sistema operativo selecciona uno los procesos que se encuentre en el estado Listo. Esta es una tarea la lleva acabo el planificador. El planificador se estudiará más adelante en la Parte Cuatro.
- **Ejecutando → Saliente.** el proceso actual en ejecución se finaliza por parte del sistema operativo tanto si el proceso indica que ha completado su ejecución como si éste se aborta. Véase tabla 3.2.

---

<sup>4</sup> En las explicaciones de este párrafo, hemos ignorado el concepto de memoria virtual. En sistemas que soporten la memoria virtual, cuando un proceso se mueve de Nuevo a Listo, su código de programa y sus datos se cargan en memoria virtual. La memoria virtual se ha explicado brevemente en el Capítulo 2 y se examinará con más detalle en el Capítulo 8.

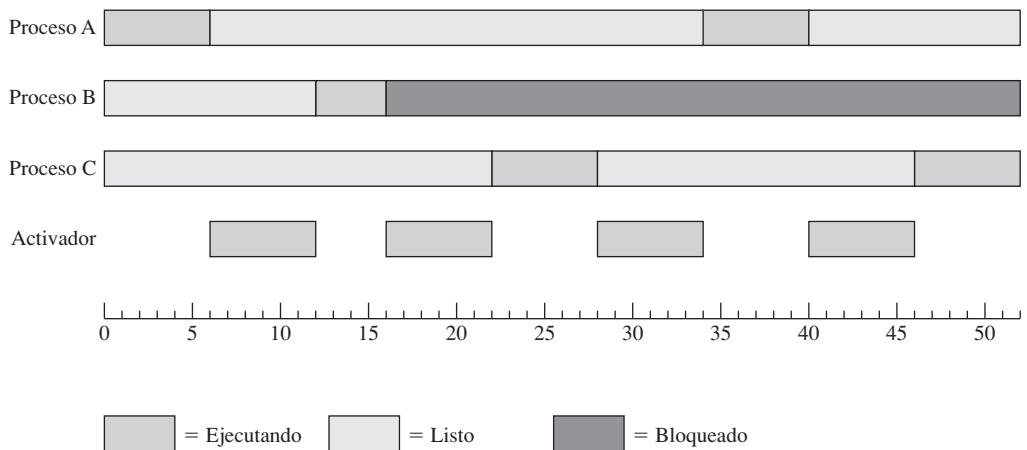
- **Ejecutando → Listo.** La razón más habitual para esta transición es que el proceso en ejecución haya alcanzado el máximo tiempo posible de ejecución de forma ininterrumpida; prácticamente todos los sistemas operativos multiprogramados imponen este tipo de restricción de tiempo. Existen otras posibles causas alternativas para esta transición, que no están incluidas en todos los sistemas operativos. Es de particular importancia el caso en el cual el sistema operativo asigna diferentes niveles de prioridad a diferentes procesos. Supóngase, por ejemplo, que el proceso A está ejecutando a un determinado nivel de prioridad, y el proceso B, a un nivel de prioridad mayor, y que se encuentra bloqueado. Si el sistema operativo se da cuenta de que se produce un evento al cual el proceso B está esperando, moverá el proceso B al estado de Listo. Esto puede interrumpir al proceso A y poner en ejecución al proceso B. Decimos, en este caso, que el sistema operativo ha expulsado al proceso A<sup>5</sup>. Adicionalmente, un proceso puede voluntariamente dejar de utilizar el procesador. Un ejemplo son los procesos que realiza alguna función de auditoría o de mantenimiento de forma periódica.
- **Ejecutando → Bloqueado.** Un proceso se pone en el estado Bloqueado si solicita algo por lo cual debe esperar. Una solicitud al sistema operativo se realiza habitualmente por medio de una llamada al sistema; esto es, una llamada del proceso en ejecución a un procedimiento que es parte del código del sistema operativo. Por ejemplo, un proceso ha solicitado un servicio que el sistema operativo no puede realizar en ese momento. Puede solicitar un recurso, como por ejemplo un fichero o una sección compartida de memoria virtual, que no está inmediatamente disponible. Cuando un proceso quiere iniciar una acción, tal como una operación de E/S, que debe completarse antes de que el proceso continúe. Cuando un proceso se comunica con otro, un proceso puede bloquearse mientras está esperando a que otro proceso le proporcione datos o esperando un mensaje de ese otro proceso.
- **Bloqueado → Listo.** Un proceso en estado Bloqueado se mueve al estado Listo cuando sucede el evento por el cual estaba esperando.
- **Listo → Saliente.** Por claridad, esta transición no se muestra en el diagrama de estados. En algunos sistemas, un padre puede terminar la ejecución de un proceso hijo en cualquier momento. También, si el padre termina, todos los procesos hijos asociados con dicho padre pueden finalizarse.
- **Bloqueado → Saliente.** Se aplican los comentarios indicados en el caso anterior.

Si regresamos a nuestro sencillo ejemplo, Figura 3.7, ahí se muestra la transición entre cada uno de los estados de proceso. La figura 3.8a sugiere la forma de aplicar un esquema de dos colas: la colas de Listos y la cola de Bloqueados. Cada proceso admitido por el sistema, se coloca en la cola de Listos. Cuando llega el momento de que el sistema operativo seleccione otro proceso a ejecutar, selecciona uno de la cola de Listos. En ausencia de un esquema de prioridad, esta cola puede ser una lista de tipo FIFO (*first-in-first-out*). Cuando el proceso en ejecución termina de utilizar el procesador, o bien finaliza o bien se coloca en la cola de Listos o de Bloqueados, dependiendo de las circunstancias. Por último, cuando sucede un evento, cualquier proceso en la cola de Bloqueados que únicamente esté esperando a dicho evento, se mueve a la cola de Listos.

Esta última transición significa que, cuando sucede un evento, el sistema operativo debe recorrer la cola entera de Bloqueados, buscando aquellos procesos que estén esperando por dicho evento. En

---

<sup>5</sup> En general, el término **expulsión** (*preemption*) se define como la reclamación de un recurso por parte de un proceso antes de que el proceso que lo poseía finalice su uso. En este caso, el recurso es el procesador. El proceso está ejecutando y puede continuar su ejecución pero es expulsado por otro proceso que va a entrar a ejecutar.



**Figura 3.7** Estado de los procesos de la traza de la Figura 3.4.

los sistemas operativos con muchos procesos, esto puede significar cientos o incluso miles de procesos en esta lista, por lo que sería mucho más eficiente tener una cola por cada evento. De esta forma, cuando sucede un evento, la lista entera de procesos de la cola correspondiente se movería al estado de Listo (Figura 3.8b).

Un refinamiento final sería: si la activación de procesos está dictada por un esquema de prioridades, sería conveniente tener varias colas de procesos listos, una por cada nivel de prioridad. El sistema operativo podría determinar cual es el proceso listo de mayor prioridad simplemente seleccionando éstas en orden.

## PROCESOS SUSPENDIDOS

**La necesidad de intercambio o swapping.** Los tres principales estados descritos (Listo, Ejecutando, Bloqueado) proporcionan una forma sistemática de modelizar el comportamiento de los procesos y diseñar la implementación del sistema operativo. Se han construido algunos sistemas operativos utilizando únicamente estos tres estados.

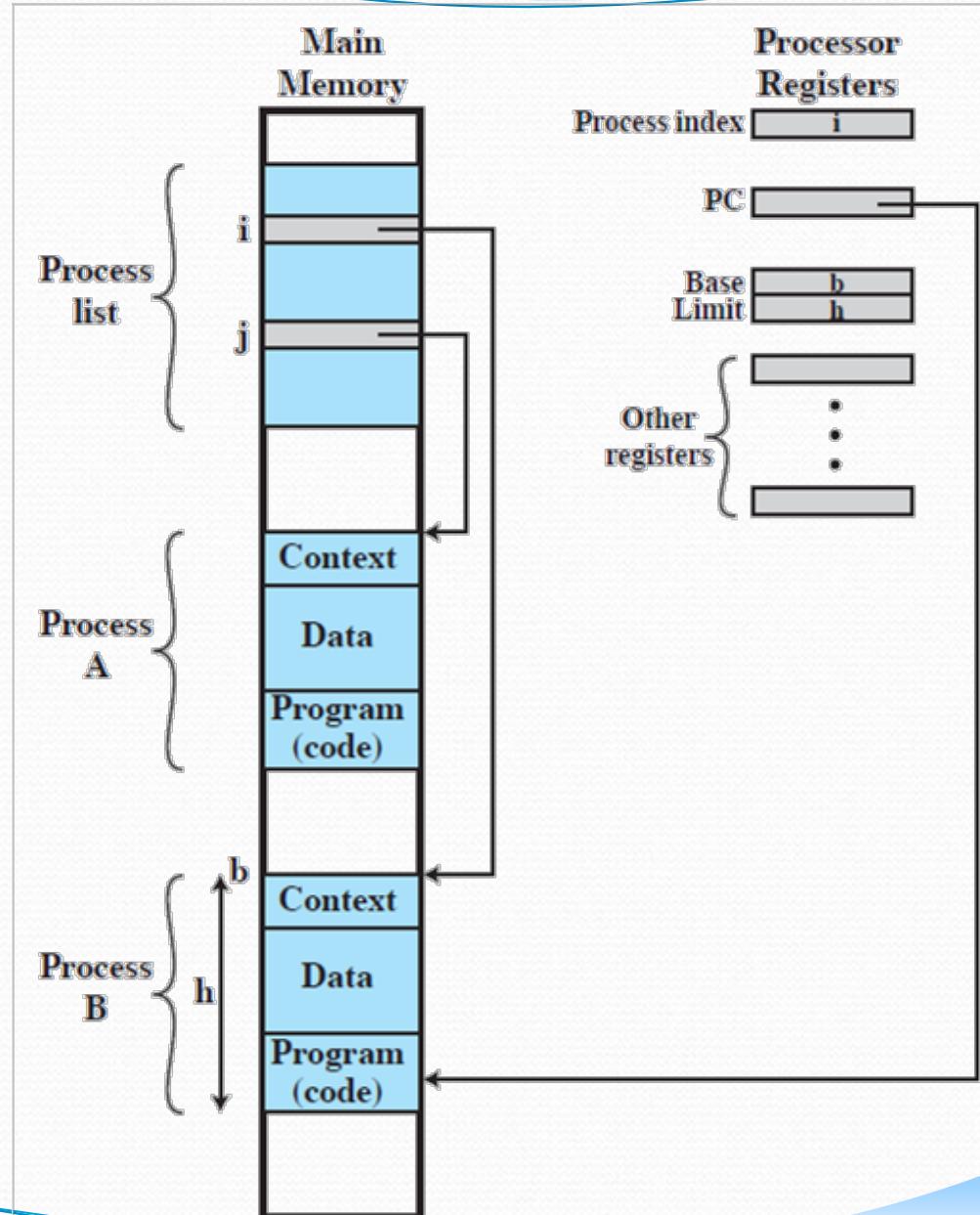
Sin embargo, existe una buena justificación para añadir otros estados al modelo. Para ver este beneficio de nuevos estados, vamos a suponer un sistema que no utiliza memoria virtual. Cada proceso que se ejecuta debe cargarse completamente en memoria principal. Por ejemplo, en la Figura 3.8b, todos los procesos en todas las colas deben residir en memoria principal.

Recuérdese que la razón de toda esta compleja maquinaria es que las operaciones de E/S son mucho más lentas que los procesos de cómputo y, por tanto, el procesador en un sistema monoprogramado estaría ocioso la mayor parte del tiempo. Pero los ajustes de la Figura 3.8b no resuelven completamente el problema. Es verdad que, en este caso, la memoria almacena múltiples procesos y el procesador puede asignarse a otro proceso si el que lo usa se queda bloqueado. La diferencia de velocidad entre el procesador y la E/S es tal que sería muy habitual que todos los procesos en memoria se encontrasen a esperas de dichas operaciones. Por tanto, incluso en un sistema multiprogramado, el procesador puede estar ocioso la mayor parte del tiempo.

¿Qué se puede hacer? La memoria principal puede expandirse para acomodar más procesos. Hay dos fallos en esta solución. Primero, existe un coste asociado a la memoria principal, que, desde un

### Implementación de Procesos Típica [Stal05] (Fig. 2.8)

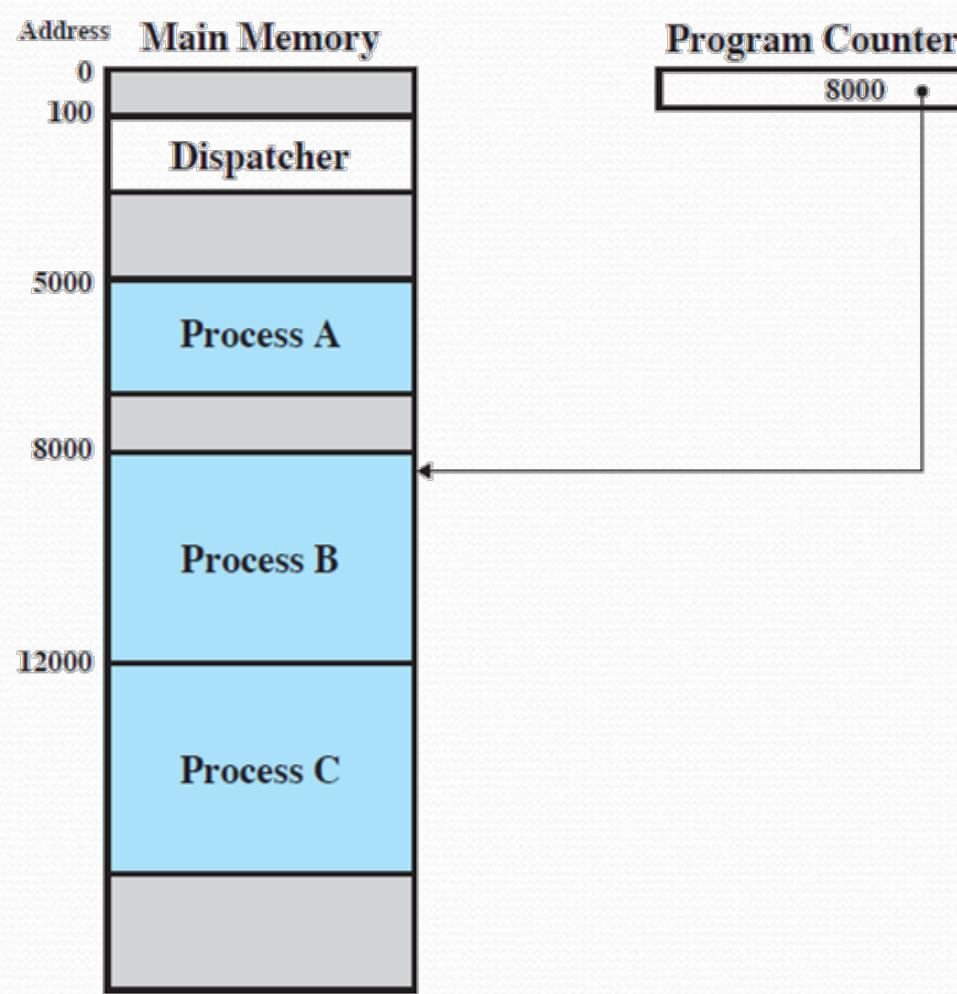
- Esta implementación permite ver al **proceso** como una **estructura de datos**.
- El **estado** completo del proceso en un instante dado se almacena en su **contexto**.
- Esta estructura permite el desarrollo de técnicas potentes que aseguren la **coordinación** y la **cooperación** entre los **procesos**.



### Concepto de traza de ejecución

- Una **traza de ejecución** es un listado de la secuencia de instrucciones de programa que realiza el procesador para un proceso
- Desde el punto de vista del procesador se entremezclan las trazas de ejecución de los procesos y las trazas del código del sistema operativo
- ¿Como consecuencia de qué situaciones se entremezclan las trazas de los procesos y las trazas del SO? Ejemplo

### Concepto de traza de ejecución



## 2.1 Componentes de un SO multiprogramado

### Concepto de traza de ejecución

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A

(b) Trace of Process B

(c) Trace of Process C

5000 = Starting address of program of Process A

8000 = Starting address of program of Process B

12000 = Starting address of program of Process C

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;  
first and third columns count instruction cycles;  
second and fourth columns show address of instruction being executed

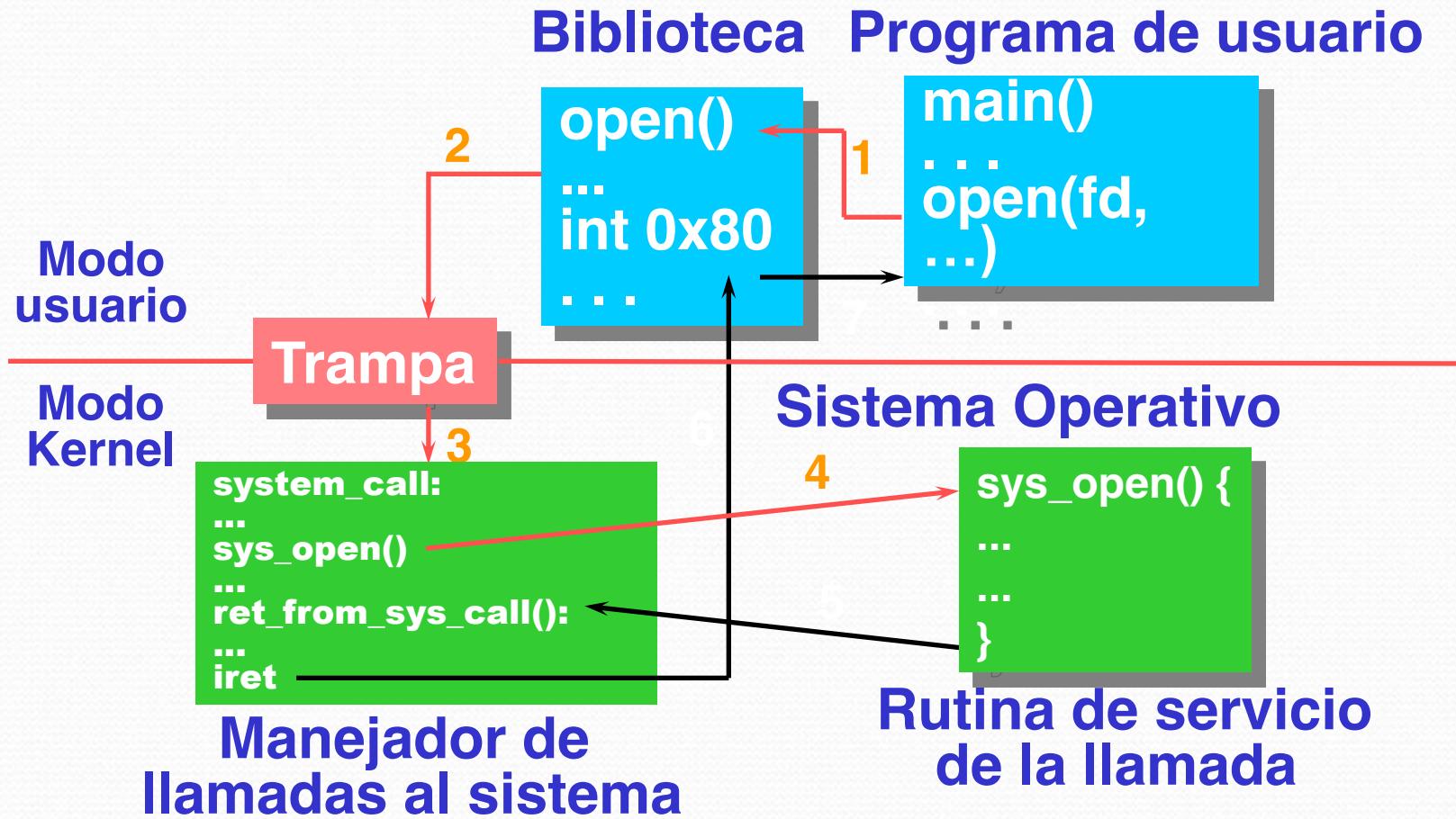
1	5000		
2	5001		
3	5002		
4	5003		
5	5004		
6	5005		
7	100		
8	101		
9	102		
10	103		
11	104		
12	105		
13	8000		
14	8001		
15	8002		
16	8003		
17	100		
18	101		
19	102		
20	103		
21	104		
22	105		
23	12000		
24	12001		
25	12002		
26	12003		
27	12004		
28	12005		
29	100	Timeout	
30	101		
31	102		
32	103		
33	104		
34	105		
35	5006		
36	5007		
37	5008		
38	5009		
39	5010		
40	5011		
41	100	Timeout	
42	101		
43	102		
44	103		
45	104		
46	105		
47	12006		
48	12007		
49	12008		
50	12009		
51	12010		
52	12011		
53	100	Timeout	

### Llamadas al Sistema

- Es la forma en la que se comunican los programas de usuario con el SO en tiempo de ejecución.
- Son solicitudes al SO de petición de servicio.
- Ejemplos de llamadas al sistema:
  - Solicitudes de E/S.
  - Gestión de procesos.
  - Gestión de memoria.
- Se implementan a través de una trampa o “interrupción software”.

## 2.1 Componentes de un SO multiprogramado

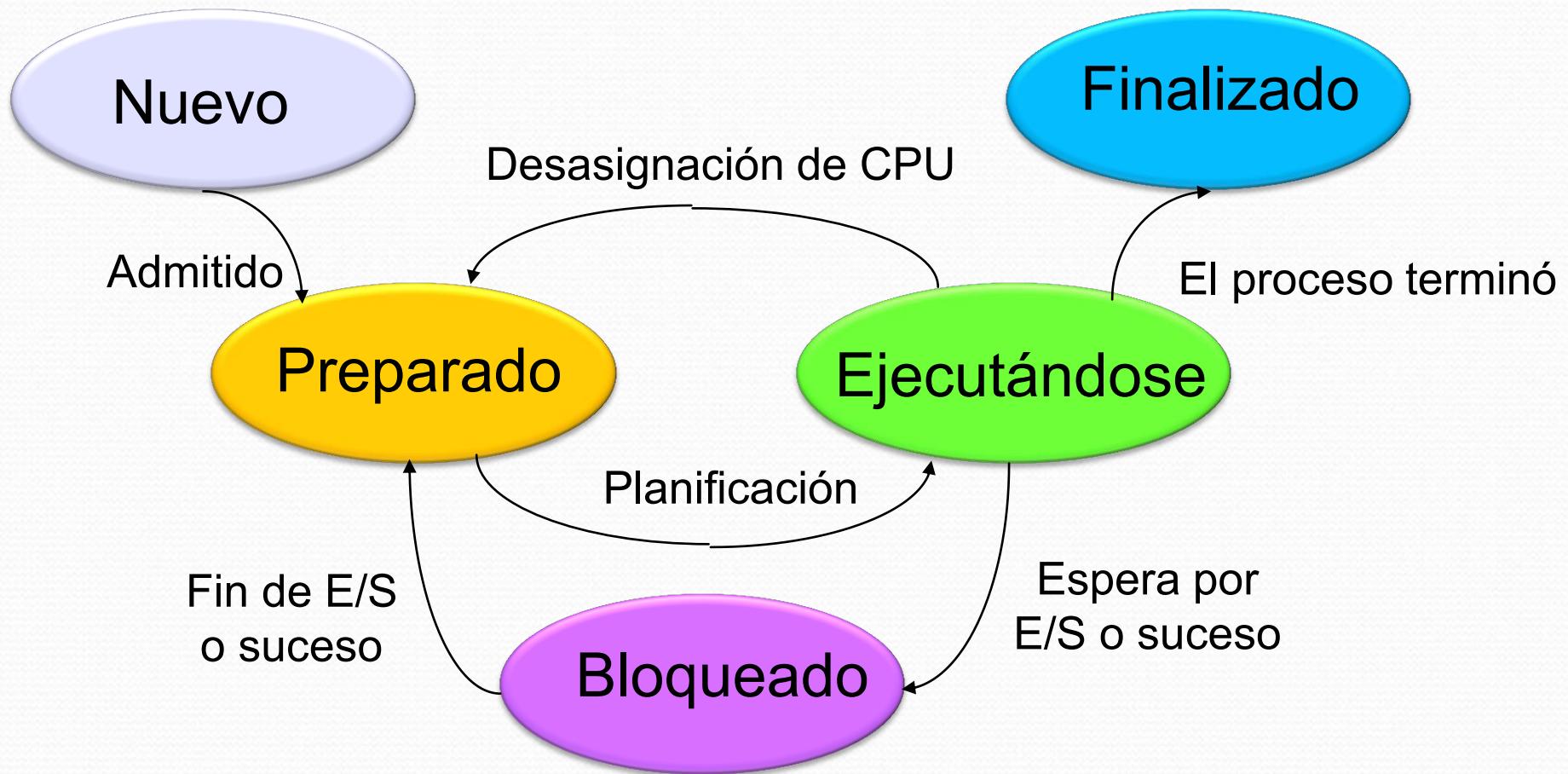
### Llamadas al Sistema



### 2.1.3 Modelo de cinco estados de los procesos

- El modelo de cinco estados trata de representar las actividades que el SO lleva a cabo sobre los procesos:
  - Creación
  - Terminación
  - Multiprogramación
- Para ello hace uso de cinco estados:
  - Ejecutándose**
  - Preparado** (listo para ejecutarse)
  - Bloqueado**
  - Nuevo**
  - Finalizado**

### 2.1.3 Modelo de cinco estados de los procesos



### Transiciones entre estados

- **Nuevo → Preparado.** El PCB está creado y el programa está disponible en memoria.
- **Ejecutándose → Finalizado.** El proceso finaliza normalmente o es abortado por el SO a causa de un error no recuperable.
- **Preparado → Ejecutándose.** El SO (planificador CPU) selecciona un proceso para que se ejecute en el procesador.
- **Ejecutándose → Bloqueado.** El proceso solicita algo al SO por lo que debe esperar.
- **Ejecutándose → Preparado.** Un proceso ha alcanzado el máximo tiempo de ejecución ininterrumpida.
- **Bloqueado → Preparado.** Se produce el evento por el cual el SO bloqueó al proceso.
- **Preparado (o Bloqueado) → Finalizado.** Terminación de un proceso por parte de otro.

### 2.2.1 Descripción de procesos: PCB

- **Identificadores:** Del proceso, del padre del proceso, del usuario, ...
- **Contexto** de registros del procesador: **PC, PSW, SP, ...**
- **Información para control del proceso:**
  - Estado del proceso. (modelo de estados)
  - Parámetros de **planificación**
  - **Evento** que mantiene al proceso bloqueado
  - Cómo acceder a la **memoria** que aloja el programa asociado al proceso
  - **Recursos** utilizados por el proceso

### Creación de un proceso: Inicialización de PCB

- Asignar **identificador** único al **proceso**
- Asignar un nuevo **PCB**
- Asignar **memoria** para el programa asociado
- **Iniciar PCB:**
  - **PC**: Dirección inicial de comienzo del programa
  - **SP**: Dirección de la pila de sistema
  - **Memoria** donde reside el programa
  - El resto de **campos** se inicializan a valores por omisión

### 2.2.2 Control de procesos: modos de ejecución del procesador

- **Modo usuario.** El programa (de usuario) que se ejecuta en este modo sólo se tiene acceso a:
  - Un subconjunto de los registros del procesador
  - Un subconjunto del repertorio de instrucciones máquina
  - Un área de la memoria
- **Modo núcleo (kernel).** El programa (SO) que se ejecuta en este modo tiene acceso a todos los recursos de la máquina

### ¿Cómo utiliza el SO el modo de ejecución?

- El modo de ejecución (incluido en PSW) cambia a modo kernel, automáticamente por hardware, cuando se produce:
  - Una interrupción
  - Una excepción
  - Una llamada al sistema
- Seguidamente se ejecuta la rutina del SO correspondiente al evento producido
- Finalmente, cuando termina la rutina, el hardware restaura automáticamente el modo de ejecución a modo usuario

### Control de procesos: cambio de contexto (cambio de proceso)

- Un proceso en estado “**Ejecutándose**” cambia a otro estado y un proceso en estado “**Preparado**” pasa a estado “**Ejecutándose**”
- ¿Cuándo puede realizarse? Cuando el **SO** pueda **ejecutarse** y decida llevarlo a cabo. Luego solamente como resultado de:
  - Una interrupción
  - Una excepción
  - Una llamada al sistema

### Pasos en un cambio de contexto

- 1) **Salvar los registros** del procesador en el **PCB** del **proceso** que actualmente está en estado “**Ejecutándose**”
- 2) **Actualizar** el campo **estado** del **proceso** al nuevo estado al que pasa e insertar el **PCB** en la **cola** correspondiente
- 3) Seleccionar un **nuevo proceso** del **conjunto** de los que se encuentran en estado “**Preparado**” (**Scheduler** o **Planificador** de CPU)
- 4) **Actualizar** el **estado del proceso** seleccionado a “**Ejecutándose**” y **sacarlo** de la **cola** de listos
- 5) **Cargar** los **registros** del procesador con la información de los registros almacenada en el **PCB** del **proceso** seleccionado

### Control de procesos: cambio de modo

- Se ejecuta una **rutina del SO** en el contexto del proceso que se encuentra en estado “**Ejecutándose**”
- ¿Cuándo puede realizarse? Siempre que el SO pueda ejecutarse, luego solamente como resultado de:
  - Una interrupción
  - Una excepción
  - Una llamada al sistema

### Pasos en un cambio de modo

- 1) El **hardware** automáticamente **salva** como mínimo el **PC** y **PSW** y cambia a **modo kernel**.
- 2) **Determinar** automáticamente la **rutina del SO** que debe **ejecutarse** y cargar el **PC** con su dirección de comienzo.
- 3) **Ejecutar la rutina**. Posiblemente la rutina comience **salvando el resto de registros** del procesador y termine **restaurando** en el procesador la información de **registros** previamente **salvada**.
- 4) Volver de la **rutina del SO**. El **hardware** automáticamente **restaura** en el procesador la información del **PC** y **PSW** previamente **salvada**.

### 2.3 Concepto de Hebra

- El concepto de **proceso (tarea)** tiene dos características diferenciadas que permiten al SO:
  - **Controlar la asignación de los recursos necesarios para la ejecución** de programas.
  - La **ejecución del programa** asociado al **proceso** de forma **intercalada** con otros programas.
- El concepto de **proceso (tarea)** y **hebras asociadas** se basa en separar estas dos características:
  - La **tarea** se encarga de **soportar** todos los **recursos necesarios** (incluida la **memoria**).
  - Cada una de las **hebras** permite la **ejecución del programa** de forma “**independiente**” del resto de hebras.

### Concepto de Hebra

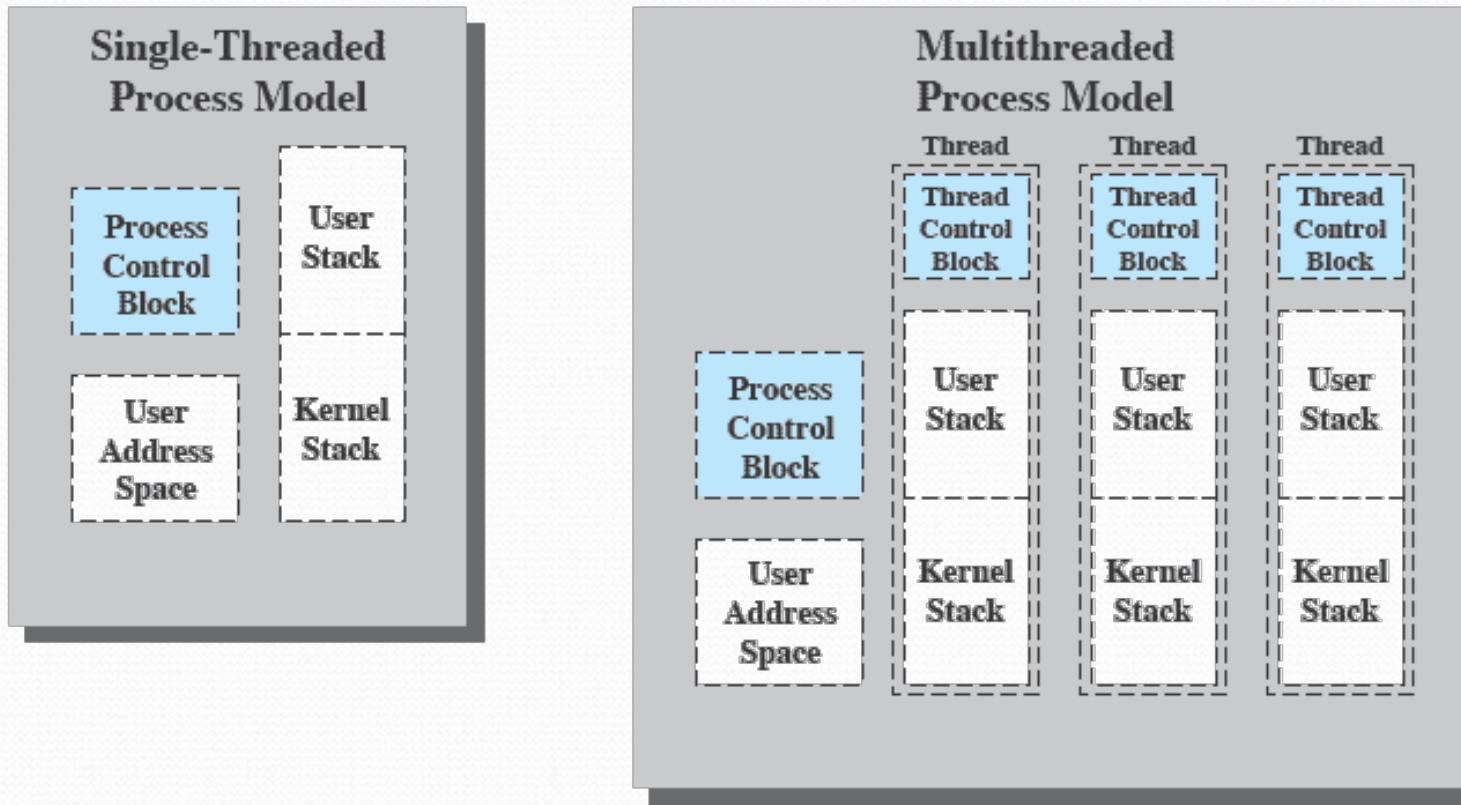
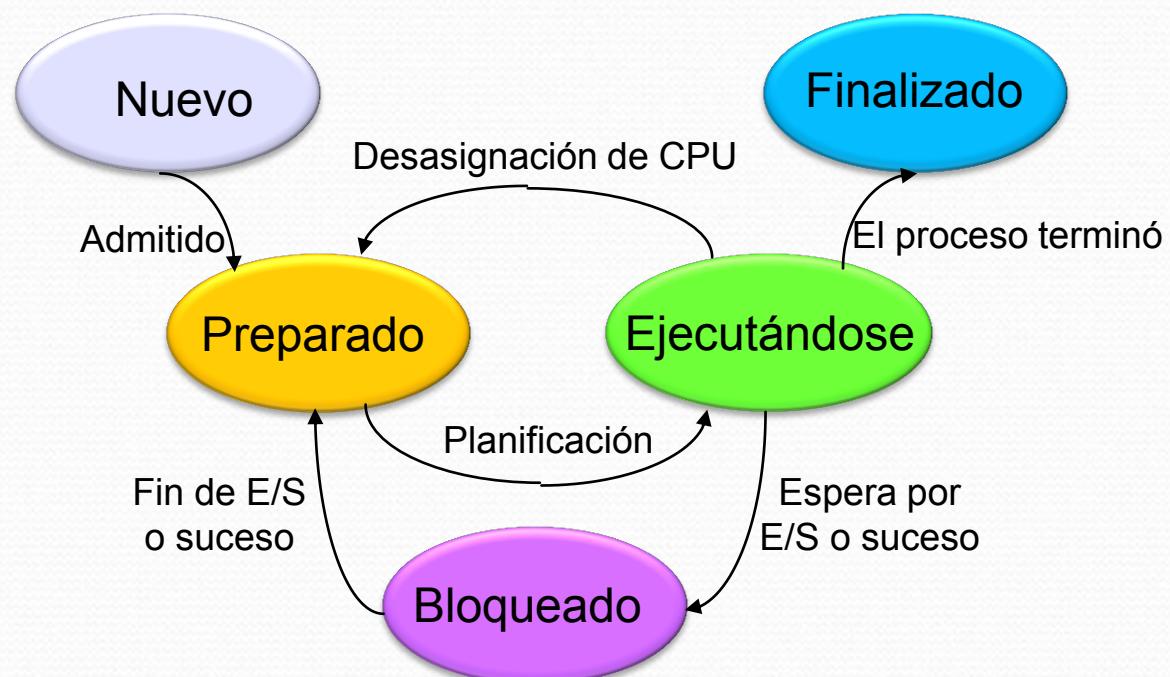


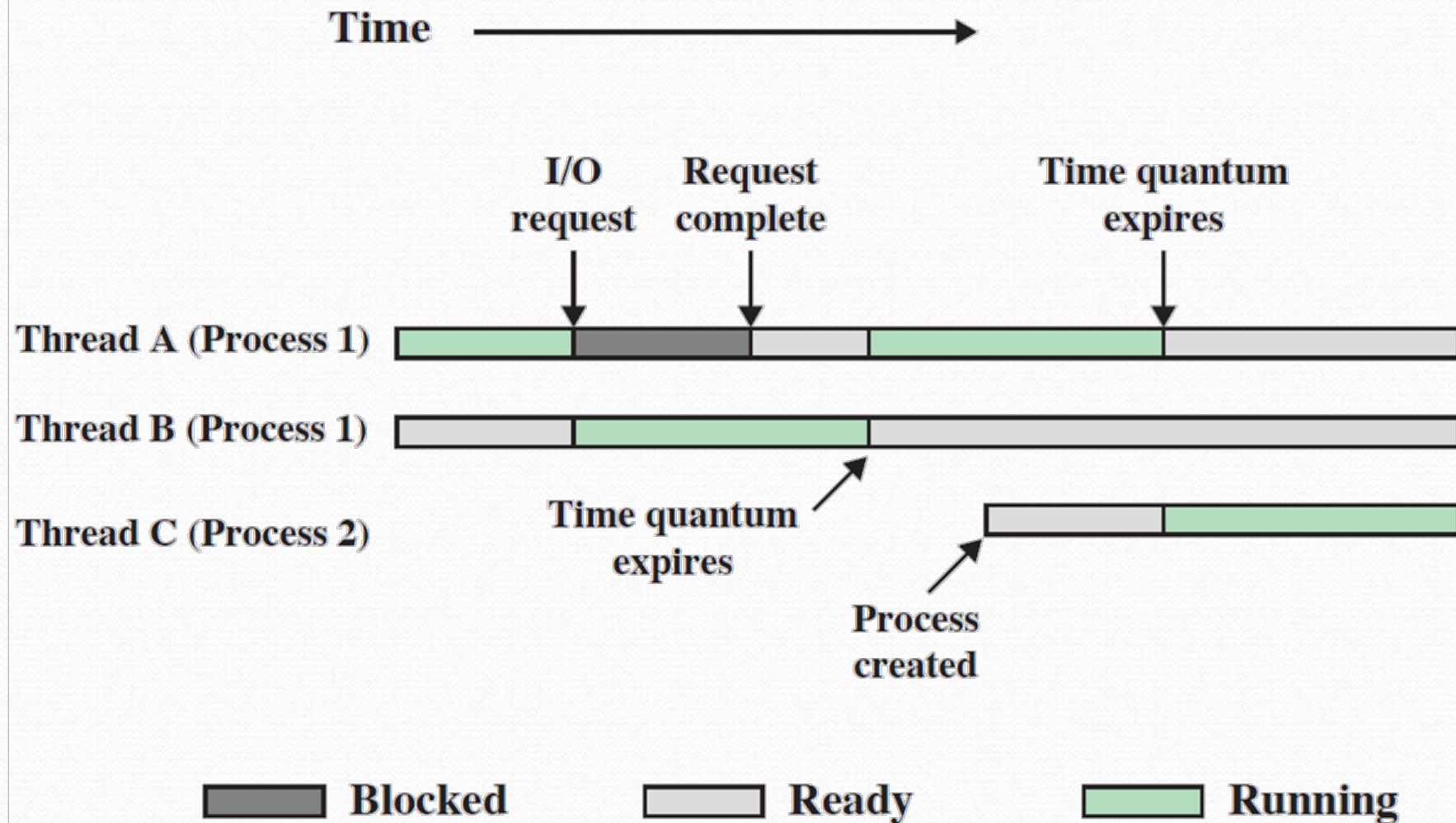
Figure 4.2 Single Threaded and Multithreaded Process Models

### Modelo de cinco estados para hebras

- Las hebras debido a su característica de ejecución de programas presentan cinco estados análogos al modelo de estados para procesos:
  - Ejecutándose**
  - Preparado** (listo para ejecutarse)
  - Bloqueado**
  - Nuevo**
  - Finalizado**

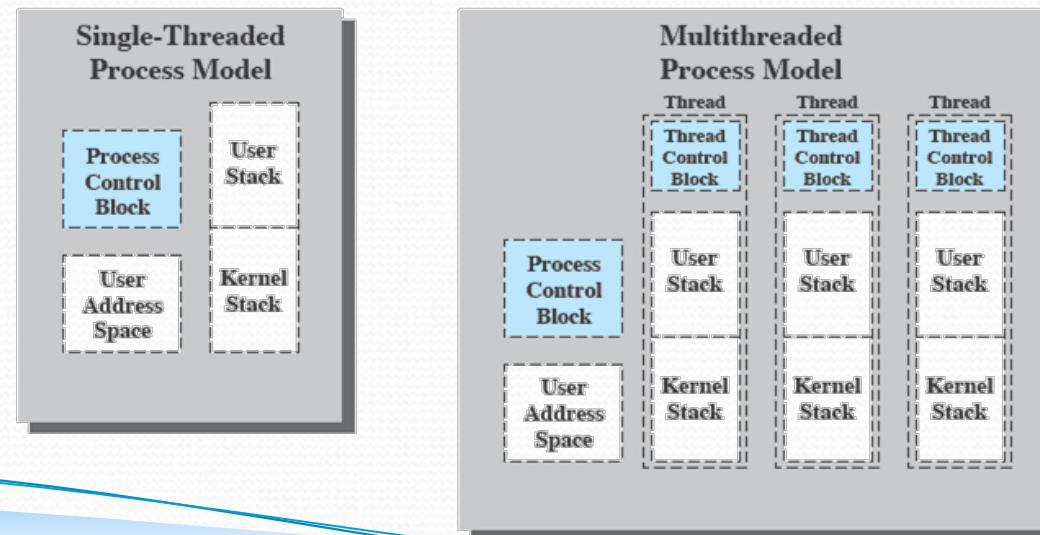


### Ventajas de las hebras



### Ventajas de las hebras

- Menor tiempo de **creación** de una **hebra** en un proceso ya creado que la creación de un nuevo proceso.
- Menor tiempo de **finalización** de una **hebra** que de un proceso.
- Menor tiempo de **cambio de contexto (hebra)** entre hebras pertenecientes al mismo proceso.
- Facilitan la **comunicación** entre hebras pertenecientes al mismo proceso.
- Permiten aprovechar las **técnicas de programación concurrente** y el **multiprocesamiento simétrico**.



### 2.4 Carga absoluta y reubicación [Stall05] (pp. 308-327)

- **Carga absoluta.** Asignar direcciones físicas (direcciones de memoria principal) al programa en tiempo de compilación. El programa no es reubicable.
- **Reubicación.** Capacidad de cargar y ejecutar un programa en un lugar arbitrario de la memoria.



*En un sistema monoprogramado, la memoria se divide en dos partes: una parte para el sistema operativo (monitor residente, núcleo) y una parte para el programa actualmente en ejecución. En un sistema multiprogramado, la parte de «usuario» de la memoria se debe subdividir posteriormente para acomodar múltiples procesos. El sistema operativo es el encargado de la tarea de subdivisión y a esta tarea se le denomina **gestión de la memoria**.*

*Una gestión de la memoria efectiva es vital en un sistema multiprogramado. Si sólo unos pocos procesos se encuentran en memoria, entonces durante una gran parte del tiempo todos los procesos esperarían por operaciones de E/S y el procesador estaría ocioso. Por tanto, es necesario asignar la memoria para asegurar una cantidad de procesos listos que consuman el tiempo de procesador disponible.*

*Comenzaremos este capítulo con una descripción de los requisitos que la gestión de la memoria pretende satisfacer. A continuación, se mostrará la tecnología de la gestión de la memoria, analizando una variedad de esquemas simples que se han utilizado. El capítulo se enfoca en el requisito de que un programa se debe cargar en memoria principal para ejecutarse. Esta discusión introduce algunos de los principios fundamentales de la gestión de la memoria.*



## 7.1. REQUISITOS DE LA GESTIÓN DE LA MEMORIA

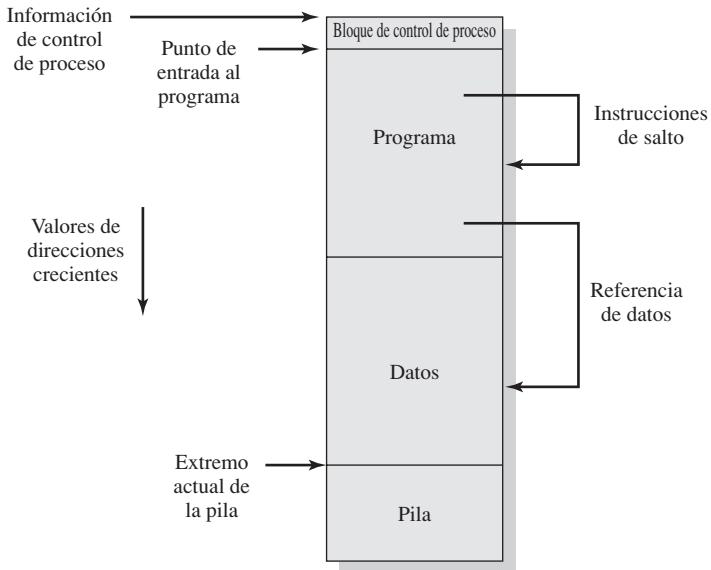
Mientras se analizan varios mecanismos y políticas asociados con la gestión de la memoria, es útil mantener en mente los requisitos que la gestión de la memoria debe satisfacer. [LIST93] sugiere cinco requisitos:

- Reubicación.
- Protección.
- Compartición.
- Organización lógica.
- Organización física.

### REUBICACIÓN

En un sistema multiprogramado, la memoria principal disponible se comparte generalmente entre varios procesos. Normalmente, no es posible que el programador sepa anticipadamente qué programas residirán en memoria principal en tiempo de ejecución de su programa. Adicionalmente, sería bueno poder intercambiar procesos en la memoria principal para maximizar la utilización del procesador, proporcionando un gran número de procesos para la ejecución. Una vez que un programa se ha llevado al disco, sería bastante limitante tener que colocarlo en la misma región de memoria principal donde se hallaba anteriormente, cuando éste se trae de nuevo a la memoria. Por el contrario, podría ser necesario **reubicar** el proceso a un área de memoria diferente.

Por tanto, no se puede conocer de forma anticipada dónde se va a colocar un programa y se debe permitir que los programas se puedan mover en la memoria principal, debido al intercambio o *swap*. Estos hechos ponen de manifiesto algunos aspectos técnicos relacionados con el direccionamiento, como se muestra en la Figura 7.1. La figura representa la imagen de un proceso. Por razones de simplicidad, se asumirá que la imagen de un proceso ocupa una región contigua de la memoria principal. Claramente, el sistema operativo necesitará conocer la ubicación de la información de control



**Figura 7.1.** Requisitos de direccionamiento para un proceso.

del proceso y de la pila de ejecución, así como el punto de entrada que utilizará el proceso para iniciar la ejecución. Debido a que el sistema operativo se encarga de gestionar la memoria y es responsable de traer el proceso a la memoria principal, estas direcciones son fáciles de adquirir. Adicionalmente, sin embargo, el procesador debe tratar con referencias de memoria dentro del propio programa. Las instrucciones de salto contienen una dirección para referenciar la instrucción que se va a ejecutar a continuación. Las instrucciones de referencia de los datos contienen la dirección del byte o palabra de datos referenciados. De alguna forma, el hardware del procesador y el software del sistema operativo deben poder traducir las referencias de memoria encontradas en el código del programa en direcciones de memoria físicas, que reflejan la ubicación actual del programa en la memoria principal.

## PROTECCIÓN

Cada proceso debe protegerse contra interferencias no deseadas por parte de otros procesos, sean accidentales o intencionadas. Por tanto, los programas de otros procesos no deben ser capaces de referenciar sin permiso posiciones de memoria de un proceso, tanto en modo lectura como escritura. Por un lado, lograr los requisitos de la reubicación incrementa la dificultad de satisfacer los requisitos de protección. Más aún, la mayoría de los lenguajes de programación permite el cálculo dinámico de direcciones en tiempo de ejecución (por ejemplo, calculando un índice de posición en un vector o un puntero a una estructura de datos). Por tanto, todas las referencias de memoria generadas por un proceso deben comprobarse en tiempo de ejecución para poder asegurar que se refieren sólo al espacio de memoria asignado a dicho proceso. Afortunadamente, se verá que los mecanismos que dan soporte a la reasignación también dan soporte al requisito de protección.

Normalmente, un proceso de usuario no puede acceder a cualquier porción del sistema operativo, ni al código ni a los datos. De nuevo, un programa de un proceso no puede saltar a una instrucción de otro proceso. Sin un trato especial, un programa de un proceso no puede acceder al área de datos de otro proceso. El procesador debe ser capaz de abortar tales instrucciones en el punto de ejecución.

Obsérvese que los requisitos de protección de memoria deben ser satisfechos por el procesador (hardware) en lugar del sistema operativo (software). Esto es debido a que el sistema operativo no puede anticipar todas las referencias de memoria que un programa hará. Incluso si tal anticipación fuera posible, llevaría demasiado tiempo calcularlo para cada programa a fin de comprobar la violación de referencias de la memoria. Por tanto, sólo es posible evaluar la permisibilidad de una referencia (acceso a datos o salto) en tiempo de ejecución de la instrucción que realiza dicha referencia. Para llevar a cabo esto, el hardware del procesador debe tener esta capacidad.

## COMPARTICIÓN

Cualquier mecanismo de protección debe tener la flexibilidad de permitir a varios procesos acceder a la misma porción de memoria principal. Por ejemplo, si varios programas están ejecutando el mismo programa, es ventajoso permitir que cada proceso pueda acceder a la misma copia del programa en lugar de tener su propia copia separada. Procesos que estén cooperando en la misma tarea podrían necesitar compartir el acceso a la misma estructura de datos. Por tanto, el sistema de gestión de la memoria debe permitir el acceso controlado a áreas de memoria compartidas sin comprometer la protección esencial. De nuevo, se verá que los mecanismos utilizados para dar soporte a la reubicación soportan también capacidades para la compartición.

## ORGANIZACIÓN LÓGICA

Casi invariablemente, la memoria principal de un computador se organiza como un espacio de almacenamiento lineal o unidimensional, compuesto por una secuencia de bytes o palabras. A nivel físico, la memoria secundaria está organizada de forma similar. Mientras que esta organización es similar al hardware real de la máquina, no se corresponde a la forma en la cual los programas se construyen normalmente. La mayoría de los programas se organizan en módulos, algunos de los cuales no se pueden modificar (sólo lectura, sólo ejecución) y algunos de los cuales contienen datos que se pueden modificar. Si el sistema operativo y el hardware del computador pueden tratar de forma efectiva los programas de usuarios y los datos en la forma de módulos de algún tipo, entonces se pueden lograr varias ventajas:

1. Los módulos se pueden escribir y compilar independientemente, con todas las referencias de un módulo desde otro resueltas por el sistema en tiempo de ejecución.
2. Con una sobrecarga adicional modesta, se puede proporcionar diferentes grados de protección a los módulos (sólo lectura, sólo ejecución).
3. Es posible introducir mecanismos por los cuales los módulos se pueden compartir entre los procesos. La ventaja de proporcionar compartición a nivel de módulo es que se corresponde con la forma en la que el usuario ve el problema, y por tanto es fácil para éste especificar la compartición deseada.

La herramienta que más adecuadamente satisface estos requisitos es la segmentación, que es una de las técnicas de gestión de la memoria exploradas en este capítulo.

## ORGANIZACIÓN FÍSICA

Como se discute en la Sección 1.5, la memoria del computador se organiza en al menos dos niveles, conocidos como memoria principal y memoria secundaria. La memoria principal proporciona acceso

rápido a un coste relativamente alto. Adicionalmente, la memoria principal es volátil; es decir, no proporciona almacenamiento permanente. La memoria secundaria es más lenta y más barata que la memoria principal y normalmente no es volátil. Por tanto, la memoria secundaria de larga capacidad puede proporcionar almacenamiento para programas y datos a largo plazo, mientras que una memoria principal más pequeña contiene programas y datos actualmente en uso.

En este esquema de dos niveles, la organización del flujo de información entre la memoria principal y secundaria supone una de las preocupaciones principales del sistema. La responsabilidad para este flujo podría asignarse a cada programador en particular, pero no es practicable o deseable por dos motivos:

1. La memoria principal disponible para un programa más sus datos podría ser insuficiente. En este caso, el programador debería utilizar una técnica conocida como **superposición** (*overlaying*), en la cual los programas y los datos se organizan de tal forma que se puede asignar la misma región de memoria a varios módulos, con un programa principal responsable para intercambiar los módulos entre disco y memoria según las necesidades. Incluso con la ayuda de herramientas de compilación, la programación con *overlays* malgasta tiempo del programador.
2. En un entorno multiprogramado, el programador no conoce en tiempo de codificación cuánto espacio estará disponible o dónde se localizará dicho espacio.

Por tanto, está claro que la tarea de mover la información entre los dos niveles de la memoria debería ser una responsabilidad del sistema. Esta tarea es la esencia de la gestión de la memoria.

## 7.2. PARTICIONAMIENTO DE LA MEMORIA

La operación principal de la gestión de la memoria es traer los procesos a la memoria principal para que el procesador los pueda ejecutar. En casi todos los sistemas multiprogramados modernos, esto implica el uso de un esquema sofisticado denominado memoria virtual. Por su parte, la memoria virtual se basa en una o ambas de las siguientes técnicas básicas: segmentación y paginación. Antes de fijarse en estas técnicas de memoria virtual, se debe preparar el camino, analizando técnicas más sencillas que no utilizan memoria virtual (Tabla 7.1). Una de estas técnicas, el particionamiento, se ha utilizado en algunas variantes de ciertos sistemas operativos ahora obsoletos. Las otras dos técnicas, paginación sencilla y segmentación sencilla, no son utilizadas de forma aislada. Sin embargo, quedará más clara la discusión de la memoria virtual si se analizan primero estas dos técnicas sin tener en cuenta consideraciones de memoria virtual.

### PARTICIONAMIENTO FIJO

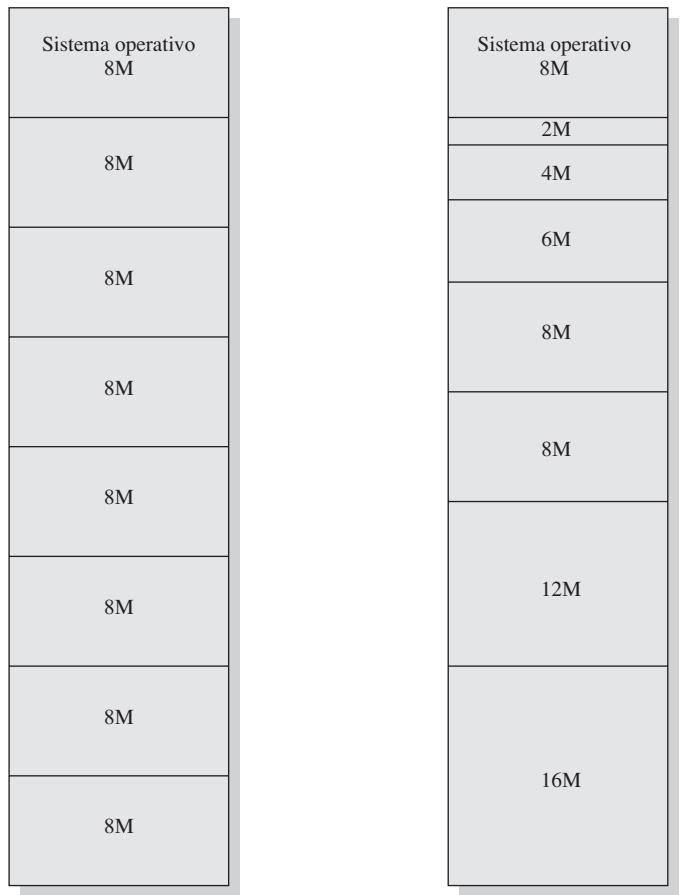
En la mayoría de los esquemas para gestión de la memoria, se puede asumir que el sistema operativo ocupa alguna porción fija de la memoria principal y que el resto de la memoria principal está disponible para múltiples procesos. El esquema más simple para gestionar la memoria disponible es repartirla en regiones con límites fijos.

#### Tamaños de partición

La Figura 7.2 muestra ejemplos de dos alternativas para el particionamiento fijo. Una posibilidad consiste en hacer uso de particiones del mismo tamaño. En este caso, cualquier proceso cuyo tamaño

**Tabla 7.1.** Técnicas de gestión de memoria.

Técnica	Descripción	Virtudes	Defectos
<b>Particionamiento fijo</b>	La memoria principal se divide en particiones estáticas en tiempo de generación del sistema. Un proceso se puede cargar en una partición con igual o superior tamaño.	Sencilla de implementar; poca sobrecarga para el sistema operativo.	Uso ineficiente de la memoria, debido a la fragmentación interna; debe fijarse el número máximo de procesos activos.
<b>Particionamiento dinámico</b>	Las particiones se crean de forma dinámica, de tal forma que cada proceso se carga en una partición del mismo tamaño que el proceso.	No existe fragmentación interna; uso más eficiente de memoria principal.	Uso ineficiente del procesador, debido a la necesidad de compactación para evitar la fragmentación externa.
<b>Paginación sencilla</b>	La memoria principal se divide en marcos del mismo tamaño. Cada proceso se divide en páginas del mismo tamaño que los marcos. Un proceso se carga a través de la carga de todas sus páginas en marcos disponibles, no necesariamente contiguos.	No existe fragmentación externa.	Una pequeña cantidad de fragmentación interna.
<b>Segmentación sencilla</b>	Cada proceso se divide en segmentos. Un proceso se carga cargando todos sus segmentos en particiones dinámicas, no necesariamente contiguas.	No existe fragmentación interna; mejora la utilización de la memoria y reduce la sobrecarga respecto al particionamiento dinámico.	Fragmentación externa.
<b>Paginación con memoria virtual</b>	Exactamente igual que la paginación sencilla, excepto que no es necesario cargar todas las páginas de un proceso. Las páginas no residentes se traen bajo demanda de forma automática.	No existe fragmentación externa; mayor grado de multiprogramación; gran espacio de direcciones virtuales.	Sobrecarga por la gestión compleja de la memoria.
<b>Segmentación con memoria virtual</b>	Exactamente igual que la segmentación, excepto que no es necesario cargar todos los segmentos de un proceso. Los segmentos no residentes se traen bajo demanda de forma automática.	No existe fragmentación interna; mayor grado de multiprogramación; gran espacio de direcciones virtuales; soporte a protección y compartición.	Sobrecarga por la gestión compleja de la memoria.



**Figura 7.2.** Ejemplo de particionamiento fijo de una memoria de 64 Mbytes.

es menor o igual que el tamaño de partición puede cargarse en cualquier partición disponible. Si todas las particiones están llenas y no hay ningún proceso en estado Listo o Ejecutando, el sistema operativo puede mandar a *swap* a un proceso de cualquiera de las particiones y cargar otro proceso, de forma que el procesador tenga trabajo que realizar.

Existen dos dificultades con el uso de las particiones fijas del mismo tamaño:

- Un programa podría ser demasiado grande para caber en una partición. En este caso, el programador debe diseñar el programa con el uso de *overlays*, de forma que sólo se necesite una porción del programa en memoria principal en un momento determinado. Cuando se necesita un módulo que no está presente, el programa de usuario debe cargar dicho módulo en la partición del programa, superponiéndolo (*overlaying*) a cualquier programa o datos que haya allí.
- La utilización de la memoria principal es extremadamente ineficiente. Cualquier programa, sin importar lo pequeño que sea, ocupa una partición entera. En el ejemplo, podría haber un programa cuya longitud es menor que 2 Mbytes; ocuparía una partición de 8 Mbytes cuando se lleva a la memoria. Este fenómeno, en el cual hay espacio interno malgastado debido al hecho

de que el bloque de datos cargado es menor que la partición, se conoce con el nombre de **fragmentación interna**.

Ambos problemas se pueden mejorar, aunque no resolver, utilizando particiones de tamaño diferente (Figura 7.2b). En este ejemplo, los programas de 16 Mbytes se pueden acomodar sin *overlays*. Las particiones más pequeñas de 8 Mbytes permiten que los programas más pequeños se puedan acomodar sin menor fragmentación interna.

### Algoritmo de ubicación

Con particiones del mismo tamaño, la ubicación de los procesos en memoria es trivial. En cuanto haya una partición disponible, un proceso se carga en dicha partición. Debido a que todas las particiones son del mismo tamaño, no importa qué partición se utiliza. Si todas las particiones se encuentran ocupadas por procesos que no están listos para ejecutar, entonces uno de dichos procesos debe llevarse a disco para dejar espacio para un nuevo proceso. Cuál de los procesos se lleva a disco es una decisión de planificación; este tema se describe en la Parte Cuatro.

Con particiones de diferente tamaño, hay dos formas posibles de asignar los procesos a las particiones. La forma más sencilla consiste en asignar cada proceso a la partición más pequeña dentro de la cual cabe<sup>1</sup>. En este caso, se necesita una cola de planificación para cada partición, que mantenga procesos en disco destinados a dicha partición (Figura 7.3a). La ventaja de esta técnica es que los procesos siempre se asignan de tal forma que se minimiza la memoria malgastada dentro de una partición (fragmentación interna).

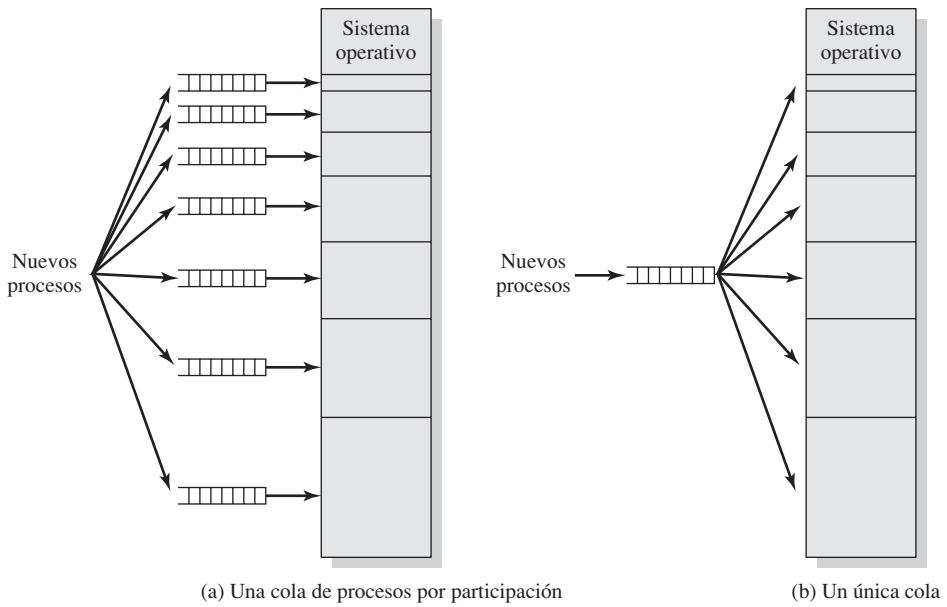
Aunque esta técnica parece óptima desde el punto de vista de una partición individual, no es óptima desde el punto de vista del sistema completo. En la Figura 7.2b, por ejemplo, se considera un caso en el que no haya procesos con un tamaño entre 12 y 16M en un determinado instante de tiempo. En este caso, la partición de 16M quedará sin utilizarse, incluso aunque se puede asignar dicha partición a algunos procesos más pequeños. Por tanto, una técnica óptima sería emplear una única cola para todos los procesos (Figura 7.3b). En el momento de cargar un proceso en la memoria principal, se selecciona la partición más pequeña disponible que puede albergar dicho proceso. Si todas las particiones están ocupadas, se debe llevar a cabo una decisión para enviar a *swap* a algún proceso. Tiene preferencia a la hora de ser expulsado a disco el proceso que ocupe la partición más pequeña que pueda albergar al proceso entrante. Es también posible considerar otros factores, como la prioridad o una preferencia por expulsar a disco procesos bloqueados frente a procesos listos.

El uso de particiones de distinto tamaño proporciona un grado de flexibilidad frente a las particiones fijas. Adicionalmente, se puede decir que los esquemas de particiones fijas son relativamente sencillos y requieren un soporte mínimo por parte del sistema operativo y una sobrecarga de procesamiento mínimo. Sin embargo, tiene una serie de desventajas:

- El número de particiones especificadas en tiempo de generación del sistema limita el número de proceso activos (no suspendidos) del sistema.
- Debido a que los tamaños de las particiones son preestablecidos en tiempo de generación del sistema, los trabajos pequeños no utilizan el espacio de las particiones eficientemente. En un entorno donde el requisito de almacenamiento principal de todos los trabajos se conoce de an-

---

<sup>1</sup> Se asume que se conoce el tamaño máximo de memoria que un proceso requerirá. No siempre es el caso. Si no se sabe lo que un proceso puede ocupar, la única alternativa es un esquema de *overlays* o el uso de memoria virtual.



**Figura 7.3.** Asignación de memoria para particionamiento fijo.

temano, esta técnica puede ser razonable, pero en la mayoría de los casos, se trata de una técnica ineficiente.

El uso de particionamiento fijo es casi desconocido hoy en día. Un ejemplo de un sistema operativo exitoso que sí utilizó esta técnica fue un sistema operativo de los primeros *mainframes* de IBM, el sistema operativo OS/MFT (*Multiprogramming with a Fixed Number of Tasks*; Multiprogramado con un número fijo de tareas).

## PARTICIONAMIENTO DINÁMICO

Para vencer algunas de las dificultades con particionamiento fijo, se desarrolló una técnica conocida como particionamiento dinámico. De nuevo, esta técnica se ha sustituido por técnicas de gestión de memoria más sofisticadas. Un sistema operativo importante que utilizó esta técnica fue el sistema operativo de *mainframes* de IBM, el sistema operativo OS/MVT (*Multiprogramming with a Variable Number of Tasks*; Multiprogramado con un número variable de tareas).

Con particionamiento dinámico, las particiones son de longitud y número variable. Cuando se lleva un proceso a la memoria principal, se le asigna exactamente tanta memoria como requiera y no más. Un ejemplo, que utiliza 64 Mbytes de memoria principal, se muestra en la Figura 7.4. Inicialmente, la memoria principal está vacía, excepto por el sistema operativo (a). Los primeros tres procesos se cargan justo donde el sistema operativo finaliza y ocupando el espacio justo para cada proceso (b, c, d). Esto deja un «hueco» al final de la memoria que es demasiado pequeño para un cuarto proceso. En algún momento, ninguno de los procesos que se encuentran en memoria está disponible. El sistema operativo lleva el proceso 2 al disco (e), que deja suficiente espacio para cargar un nuevo proceso, el proceso 4 (f). Debido a que el proceso 4 es más pequeño que el proceso 2, se crea otro pequeño hueco. Posteriormente, se alcanza un punto en el cual ninguno de los procesos de la memoria principal está listo, pero el proceso 2, en estado Listo-Suspendido, está disponible. Porque no hay es-

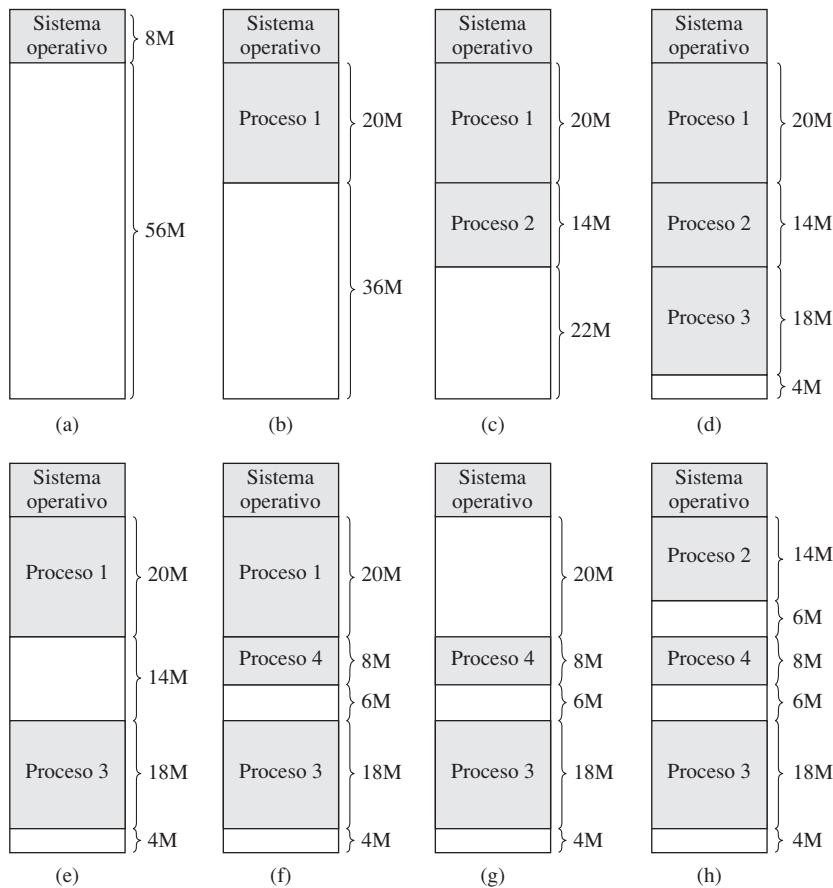


Figura 7.4. El efecto del particionamiento dinámico.

pacio suficiente en la memoria para el proceso 2, el sistema operativo lleva a disco el proceso 1 (g) y lleva a la memoria el proceso 2 (h).

Como muestra este ejemplo, el método comienza correctamente, pero finalmente lleva a una situación en la cual existen muchos huecos pequeños en la memoria. A medida que pasa el tiempo, la memoria se fragmenta cada vez más y la utilización de la memoria se decrementa. Este fenómeno se conoce como **fragmentación externa**, indicando que la memoria que es externa a todas las particiones se fragmenta de forma incremental, por contraposición a lo que ocurre con la fragmentación interna, descrita anteriormente.

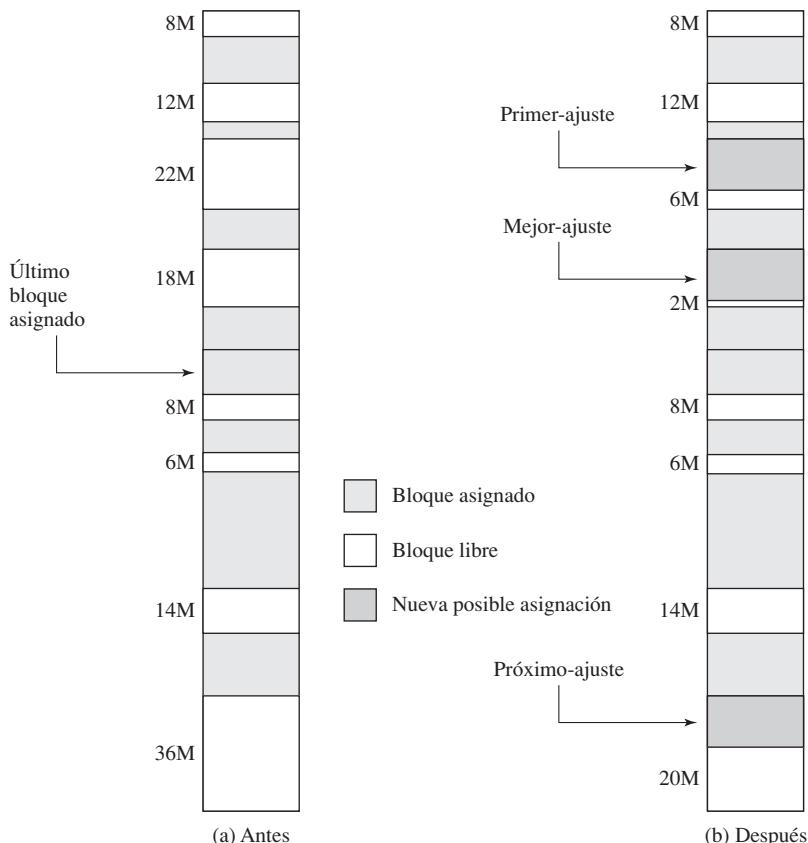
Una técnica para eliminar la fragmentación externa es la **compactación**: de vez en cuando, el sistema operativo desplaza los procesos en memoria, de forma que se encuentren contiguos y de este modo toda la memoria libre se encontrará unida en un bloque. Por ejemplo, en la Figura 7.4R, la compactación permite obtener un bloque de memoria libre de longitud 16M. Esto sería suficiente para cargar un proceso adicional. La desventaja de la compactación es el hecho de que se trata de un procedimiento que consume tiempo y malgasta tiempo de procesador. Obsérvese que la compactación requiere la capacidad de reubicación dinámica. Es decir, debe ser posible mover un programa desde una región a otra en la memoria principal sin invalidar las referencias de la memoria de cada programa (véase Apéndice 7A).

### Algoritmo de ubicación

Debido a que la compactación de memoria consume una gran cantidad de tiempo, el diseñador del sistema operativo debe ser inteligente a la hora de decidir cómo asignar la memoria a los procesos (cómo eliminar los huecos). A la hora de cargar o intercambiar un proceso a la memoria principal, y siempre que haya más de un bloque de memoria libre de suficiente tamaño, el sistema operativo debe decidir qué bloque libre asignar.

Tres algoritmos de colocación que pueden considerarse son mejor-ajuste (*best-fit*), primer-ajuste (*first-fit*) y siguiente-ajuste (*next-fit*). Todos, por supuesto, están limitados a escoger entre los bloques libres de la memoria principal que son iguales o más grandes que el proceso que va a llevarse a la memoria. **Mejor-ajuste** escoge el bloque más cercano en tamaño a la petición. **Primer-ajuste** comienza a analizar la memoria desde el principio y escoge el primer bloque disponible que sea suficientemente grande. **Siguiente-ajuste** comienza a analizar la memoria desde la última colocación y elige el siguiente bloque disponible que sea suficientemente grande.

La Figura 7.5a muestra un ejemplo de configuración de memoria después de un número de colo-  
caciones e intercambios a disco. El último bloque que se utilizó fue un bloque de 22 Mbytes del cual  
se crea una partición de 14 Mbytes. La Figura 7.5b muestra la diferencia entre los algoritmos de me-  
jor-, primer- y siguiente- ajuste a la hora de satisfacer una petición de asignación de 16 Mbytes. Me-



**Figura 7.5.** Ejemplo de configuración de la memoria antes y después de la asignación de un bloque de 16 Mbytes.

jor-ajuste busca la lista completa de bloques disponibles y hace uso del bloque de 18 Mbytes, dejando un fragmento de 2 Mbytes. *Primer-ajuste* provoca un fragmento de 6 Mbytes, y *siguiente-ajuste* provoca un fragmento de 20 Mbytes.

Cuál de estas técnicas es mejor depende de la secuencia exacta de intercambio de procesos y del tamaño de dichos procesos. Sin embargo, se pueden hacer algunos comentarios (véase también [BREN89], [SHOR75] y [BAYS77]). El algoritmo primer-ajuste no es sólo el más sencillo, sino que normalmente es también el mejor y más rápido. El algoritmo siguiente-ajuste tiende a producir resultados ligeramente peores que el primer-ajuste. El algoritmo siguiente-ajuste llevará más frecuentemente a una asignación de un bloque libre al final de la memoria. El resultado es que el bloque más grande de memoria libre, que normalmente aparece al final del espacio de la memoria, se divide rápidamente en pequeños fragmentos. Por tanto, en el caso del algoritmo siguiente-ajuste se puede requerir más frecuentemente la compactación. Por otro lado, el algoritmo primer-ajuste puede dejar el final del espacio de almacenamiento con pequeñas particiones libres que necesitan buscarse en cada paso del primer-ajuste siguiente. El algoritmo mejor-ajuste, a pesar de su nombre, su comportamiento normalmente es el peor. Debido a que el algoritmo busca el bloque más pequeño que satisfaga la petición, garantiza que el fragmento que quede sea lo más pequeño posible. Aunque cada petición de memoria siempre malgasta la cantidad más pequeña de la memoria, el resultado es que la memoria principal se queda rápidamente con bloques demasiado pequeños para satisfacer las peticiones de asignación de la memoria. Por tanto, la compactación de la memoria se debe realizar más frecuentemente que con el resto de los algoritmos.

### Algoritmo de reemplazamiento

En un sistema multiprogramado que utiliza particionamiento dinámico, puede haber un momento en el que todos los procesos de la memoria principal estén en estado bloqueado y no haya suficiente memoria para un proceso adicional, incluso después de llevar a cabo una compactación. Para evitar malgastar tiempo de procesador esperando a que un proceso se desbloquee, el sistema operativo intercambiará alguno de los procesos entre la memoria principal y disco para hacer sitio a un nuevo proceso o para un proceso que se encuentre en estado Listo-Suspendido. Por tanto, el sistema operativo debe escoger qué proceso reemplazar. Debido a que el tema de los algoritmos de reemplazo se contemplará en detalle respecto a varios esquemas de la memoria virtual, se pospone esta discusión hasta entonces.

### SISTEMA BUDDY

Ambos esquemas de particionamiento, fijo y dinámico, tienen desventajas. Un esquema de particionamiento fijo limita el número de procesos activos y puede utilizar el espacio ineficientemente si existe un mal ajuste entre los tamaños de partición disponibles y los tamaños de los procesos. Un esquema de particionamiento dinámico es más complejo de mantener e incluye la sobrecarga de la compactación. Un compromiso interesante es el sistema *buddy* ([KNUT97], [PETE77]).

En un sistema *buddy*, los bloques de memoria disponibles son de tamaño  $2^k$ ,  $L \leq K \leq U$ , donde

$2^L$  = bloque de tamaño más pequeño asignado

$2^U$  = bloque de tamaño mayor asignado; normalmente  $2^U$  es el tamaño de la memoria completa disponible

Para comenzar, el espacio completo disponible se trata como un único bloque de tamaño  $2^U$ . Si se realiza una petición de tamaño  $s$ , tal que  $2^{U-1} < s \leq 2^U$ , se asigna el bloque entero. En otro caso, el blo-

que se divide en dos bloques *buddy* iguales de tamaño  $2^{U-1}$ . Si  $2^{U-2} < s \leq 2^{U-1}$ , entonces se asigna la petición a uno de los otros dos bloques. En otro caso, uno de ellos se divide por la mitad de nuevo. Este proceso continúa hasta que el bloque más pequeño mayor o igual que  $s$  se genera y se asigna a la petición. En cualquier momento, el sistema *buddy* mantiene una lista de huecos (bloques sin asignar) de cada tamaño  $2^i$ . Un hueco puede eliminarse de la lista  $(i+1)$  dividiéndolo por la mitad para crear dos bloques de tamaño  $2^i$  en la lista  $i$ . Siempre que un par de bloques de la lista  $i$  no se encuentren asignados, son eliminados de dicha lista y unidos en un único bloque de la lista  $(i+1)$ . Si se lleva a cabo una petición de asignación de tamaño  $k$  tal que  $2^{i-1} < k \leq 2^i$ , se utiliza el siguiente algoritmo recursivo (de [LIST93]) para encontrar un hueco de tamaño  $2^i$ :

```
void obtener_hueco (int i)
{
    if (i==(U+1))
        <falla>;
    if (<lista_i vacía>)
    {
        obtener_hueco(i+1);
        <dividir hueco en dos buddies>;
        <colocar buddies en lista_i>;
    }
    <tomar primer hueco de la lista_i>;
}
```

La Figura 7.6 muestra un ejemplo que utiliza un bloque inicial de 1 Mbyte. La primera petición, A, es de 100 Kbytes, de tal forma que se necesita un bloque de 128K. El bloque inicial se divide en dos de 512K. El primero de éstos se divide en dos bloques de 256K, y el primero de éstos se divide en dos de 128K, uno de los cuales se asigna a A. La siguiente petición, B, solicita un bloque de 256K. Dicho bloque está disponible y es asignado. El proceso continúa, provocando divisiones y fusiones de bloques cuando se requiere. Obsérvese que cuando se libera E, se unen dos bloques de 128K en un bloque de 256K, que es inmediatamente unido con su bloque compañero (*buddy*).

La Figura 7.7 muestra una representación en forma de árbol binario de la asignación de bloques inmediatamente después de la petición «Liberar B». Los nodos hoja representan el particionamiento de la memoria actual. Si dos bloques son nodos hoja, entonces al menos uno de ellos debe estar asignado; en otro caso, se unen en un bloque mayor.

El sistema *buddy* es un compromiso razonable para eliminar las desventajas de ambos esquemas de particionamiento, fijo y variable, pero en los sistemas operativos contemporáneos, la memoria virtual basada en paginación y segmentación es superior. Sin embargo, el sistema *buddy* se ha utilizado en sistemas paralelos como una forma eficiente de asignar y liberar programas paralelos (por ejemplo, véase [JOHN92]). Una forma modificada del sistema *buddy* se utiliza en la asignación de memoria del núcleo UNIX (descrito en el Capítulo 8).

## REUBICACIÓN

Antes de considerar formas de tratar los problemas del particionamiento, se va a aclarar un aspecto relacionado con la colocación de los procesos en la memoria. Cuando se utiliza el esquema de particionamiento fijo de la Figura 7.3a, se espera que un proceso siempre se asigne a la misma partición.

1 bloque de 1 Mbyte	1M				
Solicitar 100K	A = 128K	128K	256K		512K
Solicitar 240K	A = 128K	128K	B = 256K		512K
Solicitar 64K	A = 128K	C = 64K	64K	B = 256K	512K
Solicitar 256K	A = 128K	C = 64K	64K	B = 256K	D = 256K
Liberar B	A = 128K	C = 64K	64K	256K	D = 256K
Liberar A	128K	C = 64K	64K	256K	D = 256K
Solicitar 75K	E = 128K	C = 64K	64K	256K	D = 256K
Liberar C	E = 128K	128K		256K	D = 256K
Liberar E		512K		D = 256K	256K
Liberar D				1M	

Figura 7.6. Ejemplo de sistema Buddy.

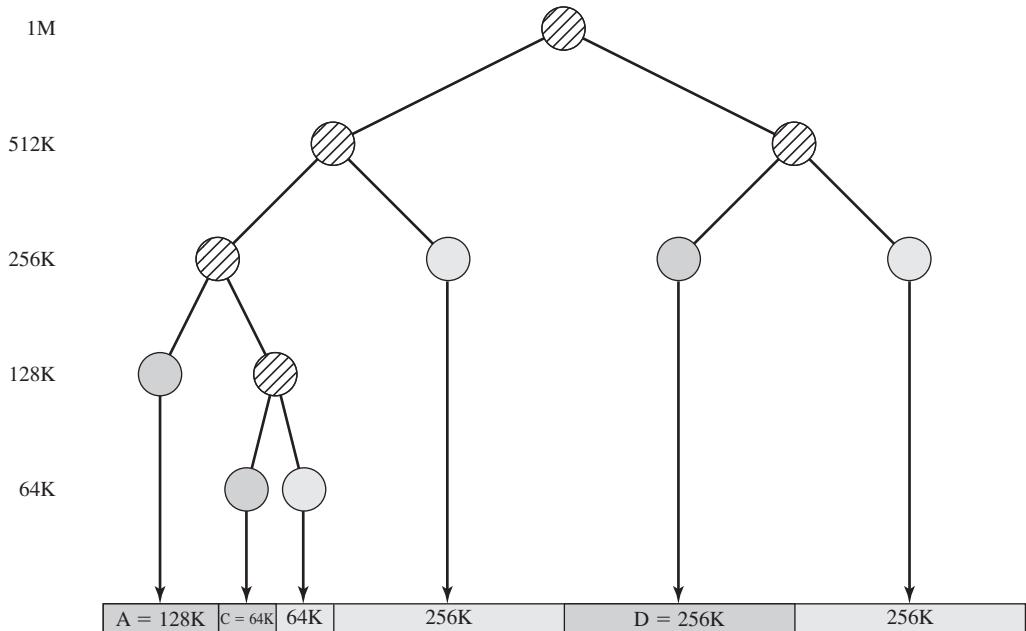


Figura 7.7. Representación en forma de árbol del sistema Buddy.

Es decir, sea cual sea la partición seleccionada cuando se carga un nuevo proceso, ésta será la utilizada para el intercambio del proceso entre la memoria y el área de *swap* en disco. En este caso, se utiliza un cargador sencillo, tal y como se describe en el Apéndice 7A: cuando el proceso se carga por

primera vez, todas las referencias de la memoria relativas del código se reemplazan por direcciones de la memoria principal absolutas, determinadas por la dirección base del proceso cargado.

En el caso de particiones de igual tamaño (Figura 7.2), y en el caso de una única cola de procesos para particiones de distinto tamaño (Figura 7.3b), un proceso puede ocupar diferentes particiones durante el transcurso de su ciclo de vida. Cuando la imagen de un proceso se crea por primera vez, se carga en la misma partición de memoria principal. Más adelante, el proceso podría llevarse a disco; cuando se trae a la memoria principal posteriormente, podría asignarse a una partición distinta de la última vez. Lo mismo ocurre en el caso del particionamiento dinámico. Obsérvese en las Figuras 7.4c y h que el proceso 2 ocupa dos regiones diferentes de memoria en las dos ocasiones que se trae a la memoria. Más aún, cuando se utiliza la compactación, los procesos se desplazan mientras están en la memoria principal. Por tanto, las ubicaciones (de las instrucciones y los datos) referenciadas por un proceso no son fijas. Cambiarán cada vez que un proceso se intercambia o se desplaza. Para resolver este problema, se realiza una distinción entre varios tipos de direcciones. Una **dirección lógica** es una referencia a una ubicación de memoria independiente de la asignación actual de datos a la memoria; se debe llevar a cabo una traducción a una dirección física antes de que se alcance el acceso a la memoria. Una **dirección relativa** es un ejemplo particular de dirección lógica, en el que la dirección se expresa como una ubicación relativa a algún punto conocido, normalmente un valor en un registro del procesador. Una **dirección física**, o dirección absoluta, es una ubicación real de la memoria principal.

Los programas que emplean direcciones relativas de memoria se cargan utilizando carga dinámica en tiempo de ejecución (véase Apéndice 7A, donde se recoge una discusión). Normalmente, todas las referencias de memoria de los procesos cargados son relativas al origen del programa. Por tanto, se necesita un mecanismo hardware para traducir las direcciones relativas a direcciones físicas de la memoria principal, en tiempo de ejecución de la instrucción que contiene dicha referencia.

La Figura 7.8 muestra la forma en la que se realiza normalmente esta traducción de direcciones. Cuando un proceso se asigna al estado Ejecutando, un registro especial del procesador, algunas veces llamado registro base, carga la dirección inicial del programa en la memoria principal. Existe también un registro «valla», que indica el final de la ubicación del programa; estos valores se establecen cuando el programa se carga en la memoria o cuando la imagen del proceso se lleva a la memoria. A lo largo de la ejecución del proceso, se encuentran direcciones relativas. Éstas incluyen los contenidos del registro de las instrucciones, las direcciones de instrucciones que ocurren en los saltos e instrucciones *call*, y direcciones de datos existentes en instrucciones de carga y almacenamiento. El procesador manipula cada dirección relativa, a través de dos pasos. Primero, el valor del registro base se suma a la dirección relativa para producir una dirección absoluta. Segundo, la dirección resultante se compara con el valor del registro «valla». Si la dirección se encuentra dentro de los límites, entonces se puede llevar a cabo la ejecución de la instrucción. En otro caso, se genera una interrupción, que debe manejar el sistema operativo de algún modo.

El esquema de la Figura 7.8 permite que se traigan a memoria los programas y que se lleven a disco, a lo largo de la ejecución. También proporciona una medida de protección: cada imagen del proceso está aislada mediante los contenidos de los registros base y valla. Además, evita accesos no autorizados por parte de otros procesos.

### 7.3. PAGINACIÓN

Tanto las particiones de tamaño fijo como variable son ineficientes en el uso de la memoria; los primeros provocan fragmentación interna, los últimos fragmentación externa. Supóngase, sin embargo, que la memoria principal se divide en porciones de tamaño fijo relativamente pequeños, y que cada proceso también se divide en porciones pequeñas del mismo tamaño fijo. A dichas porciones del pro-

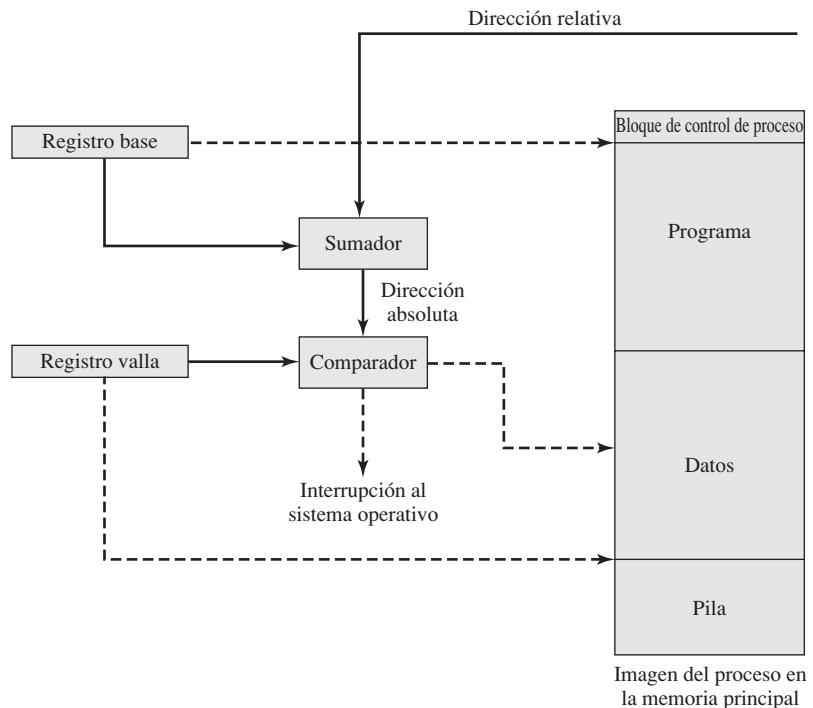
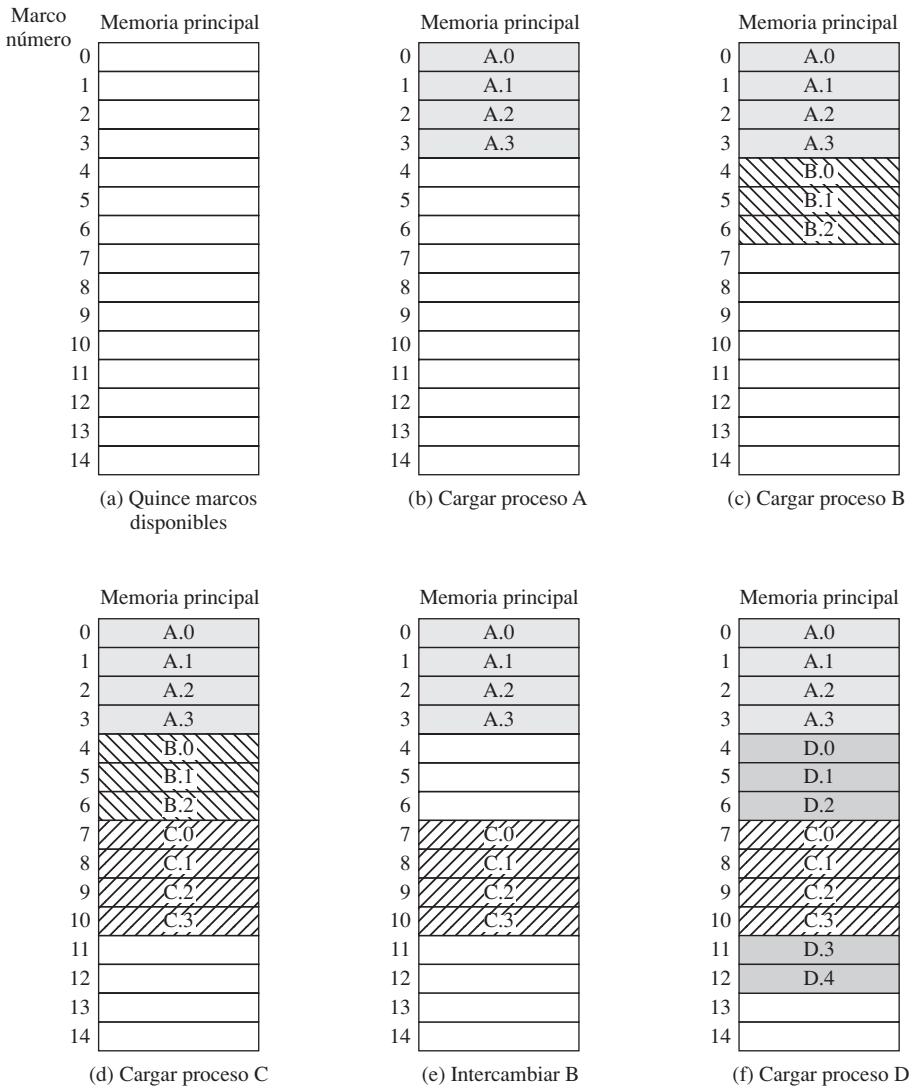


Figura 7.8. Soporte hardware para la reubicación.

ceso, conocidas como **páginas**, se les asigna porciones disponibles de memoria, conocidas como **marcos**, o marcos de páginas. Esta sección muestra que el espacio de memoria malgastado por cada proceso debido a la fragmentación interna corresponde sólo a una fracción de la última página de un proceso. No existe fragmentación externa.

La Figura 7.9 ilustra el uso de páginas y marcos. En un momento dado, algunos de los marcos de la memoria están en uso y algunos están libres. El sistema operativo mantiene una lista de marcos libres. El proceso A, almacenado en disco, está formado por cuatro páginas. En el momento de cargar este proceso, el sistema operativo encuentra cuatro marcos libres y carga las cuatro páginas del proceso A en dichos marcos (Figura 7.9b). El proceso B, formado por tres páginas, y el proceso C, formado por cuatro páginas, se cargan a continuación. En ese momento, el proceso B se suspende y deja la memoria principal. Posteriormente, todos los procesos de la memoria principal se bloquean, y el sistema operativo necesita traer un nuevo proceso, el proceso D, que está formado por cinco páginas.

Ahora supóngase, como en este ejemplo, que no hay suficientes marcos contiguos sin utilizar para ubicar un proceso. ¿Esto evitaría que el sistema operativo cargara el proceso D? La respuesta es no, porque una vez más se puede utilizar el concepto de dirección lógica. Un registro base sencillo de direcciones no basta en esta ocasión. En su lugar, el sistema operativo mantiene una **tabla de páginas** por cada proceso. La tabla de páginas muestra la ubicación del marco por cada página del proceso. Dentro del programa, cada dirección lógica está formada por un número de página y un desplazamiento dentro de la página. Es importante recordar que en el caso de una partición simple, una dirección lógica es la ubicación de una palabra relativa al comienzo del programa; el procesador la traduce en una dirección física. Con paginación, la traducción de direcciones lógicas a físicas las realiza también el hardware del procesador. Ahora el procesador debe conocer cómo acceder a la ta-



**Figura 7.9.** Asignación de páginas de proceso a marcos libres.

bla de páginas del proceso actual. Presentado como una dirección lógica (número de página, desplazamiento), el procesador utiliza la tabla de páginas para producir una dirección física (número de marco, desplazamiento).

Continuando con nuestro ejemplo, las cinco páginas del proceso D se cargan en los marcos 4, 5, 6, 11 y 12. La Figura 7.10 muestra las diferentes tablas de páginas en este momento. Una tabla de páginas contiene una entrada por cada página del proceso, de forma que la tabla se indexe fácilmente por el número de página (iniciando en la página 0). Cada entrada de la tabla de páginas contiene el número del marco en la memoria principal, si existe, que contiene la página correspondiente. Adicionalmente, el sistema operativo mantiene una única lista de marcos libres de todos los marcos de la memoria que se encuentran actualmente no ocupados y disponibles para las páginas.

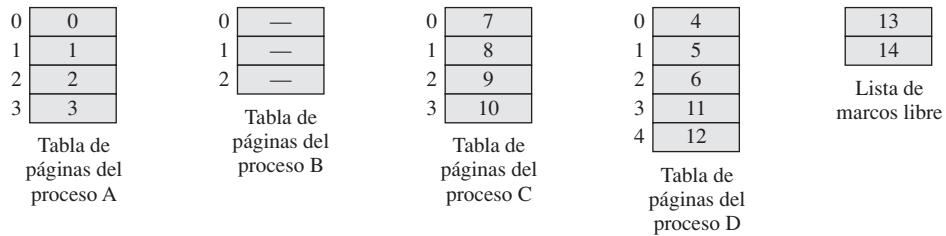


Figura 7.10. Estructuras de datos para el ejemplo de la Figura 7.9 en el instante (f).

Por tanto vemos que la paginación simple, tal y como se describe aquí, es similar al particionamiento fijo. Las diferencias son que, con la paginación, las particiones son bastante pequeñas; un programa podría ocupar más de una partición; y dichas particiones no necesitan ser contiguas.

Para hacer este esquema de paginación conveniente, el tamaño de página y por tanto el tamaño del marco debe ser una potencia de 2. Con el uso de un tamaño de página potencia de 2, es fácil demostrar que la dirección relativa, que se define con referencia al origen del programa, y la dirección lógica, expresada como un número de página y un desplazamiento, es lo mismo. Se muestra un ejemplo en la Figura 7.11. En este ejemplo, se utilizan direcciones de 16 bits, y el tamaño de la página es  $1K = 1024$  bytes. La dirección relativa 1502, en formato binario, es 0000010111011110. Con una página de tamaño 1K, se necesita un campo de desplazamiento de 10 bits, dejando 6 bits para el número de página. Por tanto, un programa puede estar compuesto por un máximo de  $2^6=64$  páginas de 1K byte cada una. Como muestra la Figura 7.11, la dirección relativa 1502 corresponde a un desplazamiento de 478 (0111011110) en la página 1 (000001), que forma el mismo número de 16 bits, 0000010111011110.

Las consecuencias de utilizar un tamaño de página que es una potencia de 2 son dobles. Primero, el esquema de direccionamiento lógico es transparente al programador, al ensamblador y al montador. Cada dirección lógica (número de página, desplazamiento) de un programa es idéntica a su dirección relativa. Segundo, es relativamente sencillo implementar una función que ejecute el hardware para llevar a cabo dinámicamente la traducción de direcciones en tiempo de ejecución. Considérese una dirección de  $n+m$  bits, donde los  $n$  bits de la izquierda corresponden al número de página y los  $m$  bits de la derecha corresponden al desplazamiento. En el ejemplo (Figura 7.11b),  $n = 6$  y  $m = 10$ . Se necesita llevar a cabo los siguientes pasos para la traducción de direcciones:

- Extraer el número de página como los  $n$  bits de la izquierda de la dirección lógica.
- Utilizar el número de página como un índice a tabla de páginas del proceso para encontrar el número de marco,  $k$ .
- La dirección física inicial del marco es  $k \times 2^m$ , y la dirección física del byte referenciado es dicho número más el desplazamiento. Esta dirección física no necesita calcularse; es fácilmente construida concatenando el número de marco al desplazamiento.

En el ejemplo, se parte de la dirección lógica 0000010111011110, que corresponde a la página número 1, desplazamiento 478. Supóngase que esta página reside en el marco de memoria principal 6 = número binario 000110. Por tanto, la dirección física corresponde al marco número 6, desplazamiento 478 = 0001100111011110 (Figura 7.12a).

Resumiendo, con la paginación simple, la memoria principal se divide en muchos marcos pequeños de igual tamaño. Cada proceso se divide en páginas de igual tamaño; los procesos más pequeños requieren menos páginas, los procesos mayores requieren más. Cuando un proceso se trae a la memo-

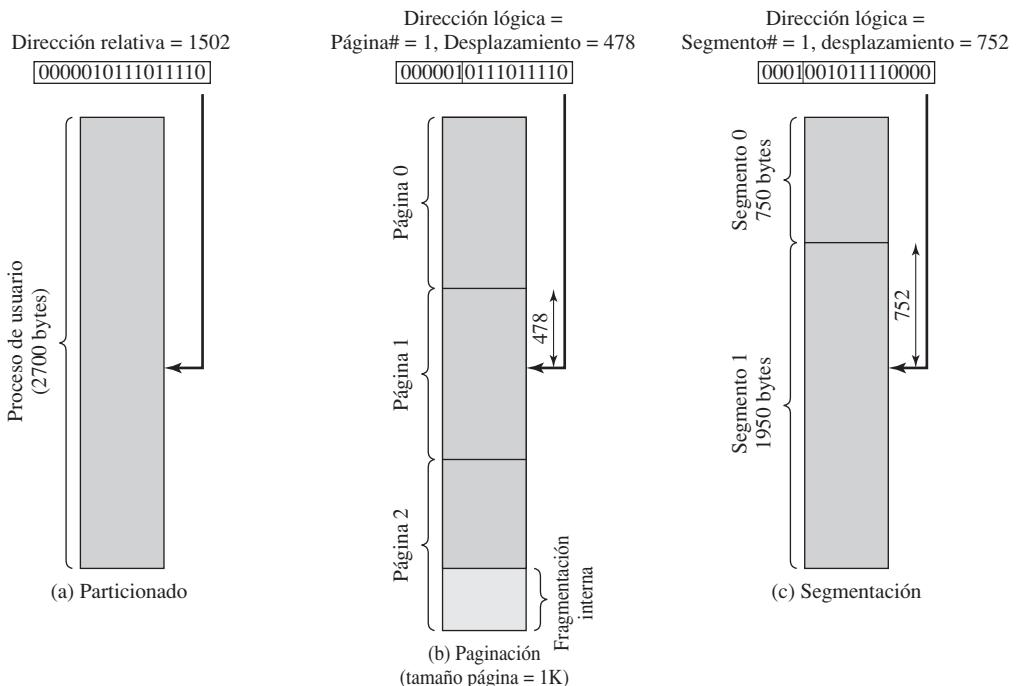


Figura 7.11. Direcciones lógicas.

ria, todas sus páginas se cargan en los marcos disponibles, y se establece una tabla de páginas. Esta técnica resuelve muchos de los problemas inherentes en el particionamiento.

## 7.4. SEGMENTACIÓN

Un programa de usuario se puede subdividir utilizando segmentación, en la cual el programa y sus datos asociados se dividen en un número de **segmentos**. No se requiere que todos los programas sean de la misma longitud, aunque hay una longitud máxima de segmento. Como en el caso de la paginación, una dirección lógica utilizando segmentación está compuesta por dos partes, en este caso un número de segmento y un desplazamiento.

Debido al uso de segmentos de distinto tamaño, la segmentación es similar al particionamiento dinámico. En la ausencia de un esquema de *overlays* o el uso de la memoria virtual, se necesitaría que todos los segmentos de un programa se cargaran en la memoria para su ejecución. La diferencia, comparada con el particionamiento dinámico, es que con la segmentación un programa podría ocupar más de una partición, y estas particiones no necesitan ser contiguas. La segmentación elimina la fragmentación interna pero, al igual que el particionamiento dinámico, sufre de fragmentación externa. Sin embargo, debido a que el proceso se divide en varias piezas más pequeñas, la fragmentación externa debería ser menor.

Mientras que la paginación es invisible al programador, la segmentación es normalmente visible y se proporciona como una utilidad para organizar programas y datos. Normalmente, el programador o compilador asignará programas y datos a diferentes segmentos. Para los propósitos de la programación modular, los programas o datos se pueden dividir posteriormente en múltiples segmentos. El in-

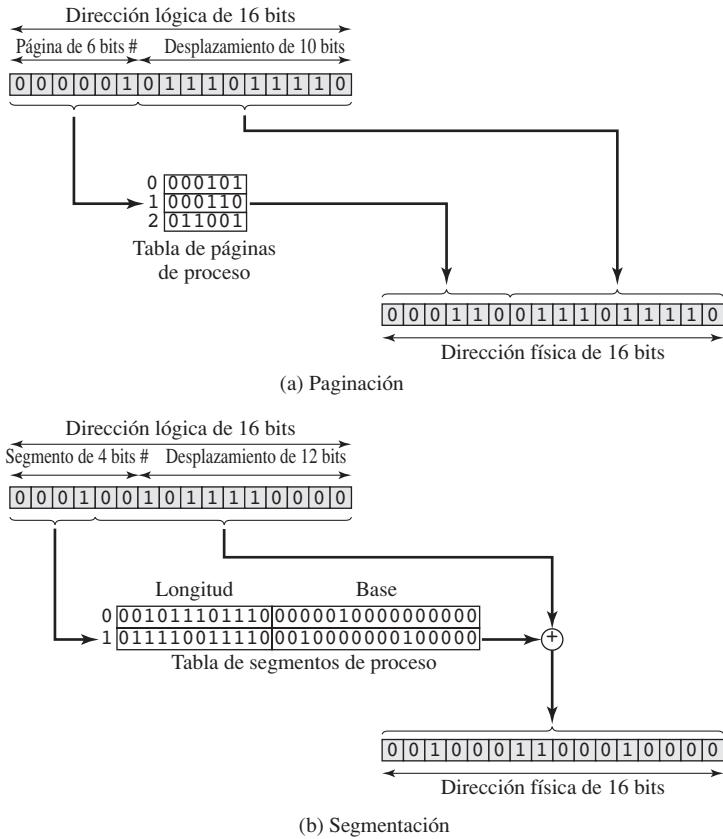


Figura 7.12. Ejemplos de traducción de direcciones lógicas a físicas.

conveniente principal de este servicio es que el programador debe ser consciente de la limitación de tamaño de segmento máximo.

Otra consecuencia de utilizar segmentos de distinto tamaño es que no hay una relación simple entre direcciones lógicas y direcciones físicas. De forma análoga a la paginación, un esquema de segmentación sencillo haría uso de una tabla de segmentos por cada proceso y una lista de bloques libre de memoria principal. Cada entrada de la tabla de segmentos tendría que proporcionar la dirección inicial de la memoria principal del correspondiente segmento. La entrada también debería proporcionar la longitud del segmento, para asegurar que no se utilizan direcciones no válidas. Cuando un proceso entra en el estado Ejecutando, la dirección de su tabla de segmentos se carga en un registro especial utilizado por el hardware de gestión de la memoria.

Considérese una dirección de  $n+m$  bits, donde los  $n$  bits de la izquierda corresponden al número de segmento y los  $m$  bits de la derecha corresponden al desplazamiento. En el ejemplo (Figura 7.11c),  $n = 4$  y  $m = 12$ . Por tanto, el tamaño de segmento máximo es  $2^{12} = 4096$ . Se necesita llevar a cabo los siguientes pasos para la traducción de direcciones:

- Extraer el número de segmento como los  $n$  bits de la izquierda de la dirección lógica.
- Utilizar el número de segmento como un índice a la tabla de segmentos del proceso para encontrar la dirección física inicial del segmento.

- Comparar el desplazamiento, expresado como los  $m$  bits de la derecha, y la longitud del segmento. Si el desplazamiento es mayor o igual que la longitud, la dirección no es válida.
- La dirección física deseada es la suma de la dirección física inicial y el desplazamiento.

En el ejemplo, se parte de la dirección lógica 0001001011110000, que corresponde al segmento número 1, desplazamiento 752. Supóngase que este segmento reside en memoria principal, comenzando en la dirección física inicial 001000000100000. Por tanto, la dirección física es  $001000000100000 + 00101110000 = 0010001100010000$  (Figura 7.12b).

Resumiendo, con la segmentación simple, un proceso se divide en un conjunto de segmentos que no tienen que ser del mismo tamaño. Cuando un proceso se trae a memoria, todos sus segmentos se cargan en regiones de memoria disponibles, y se crea la tabla de segmentos.

## 7.5. RESUMEN

Una de las tareas más importantes y complejas de un sistema operativo es la gestión de memoria. La gestión de memoria implica tratar la memoria principal como un recurso que debe asignarse y compartirse entre varios procesos activos. Para utilizar el procesador y las utilidades de E/S eficientemente, es deseable mantener tantos procesos en memoria principal como sea posible. Además, es deseable también liberar a los programadores de tener en cuenta las restricciones de tamaño en el desarrollo de los programas.

Las herramientas básicas de gestión de memoria son la paginación y la segmentación. Con la paginación, cada proceso se divide en un conjunto de páginas de tamaño fijo y de un tamaño relativamente pequeño. La segmentación permite el uso de piezas de tamaño variable. Es también posible combinar la segmentación y la paginación en un único esquema de gestión de memoria.

## 7.6. LECTURAS RECOMENDADAS

Los libros de sistema operativos recomendados en la Sección 2.9 proporcionan cobertura para la gestión de memoria.

Debido a que el sistema de particionamiento se ha suplantado por técnicas de memoria virtual, la mayoría de los libros sólo cubren superficialmente el tema. Uno de los tratamientos más completos e interesantes se encuentra en [MILE92]. Una discusión más profunda de las estrategias de particionamiento se encuentra en [KNUT97].

Los temas de enlace y carga se cubren en muchos libros de desarrollo de programas, arquitectura de computadores y sistemas operativos. Un tratamiento particularmente detallado es [BECK90]. [CLAR98] también contiene una buena discusión. Una discusión práctica en detalle de este tema, con numerosos ejemplos de sistemas operativos, es [LEVI99].

**BECK90** Beck, L. *System Software*. Reading, MA: Addison-Wesley, 1990.

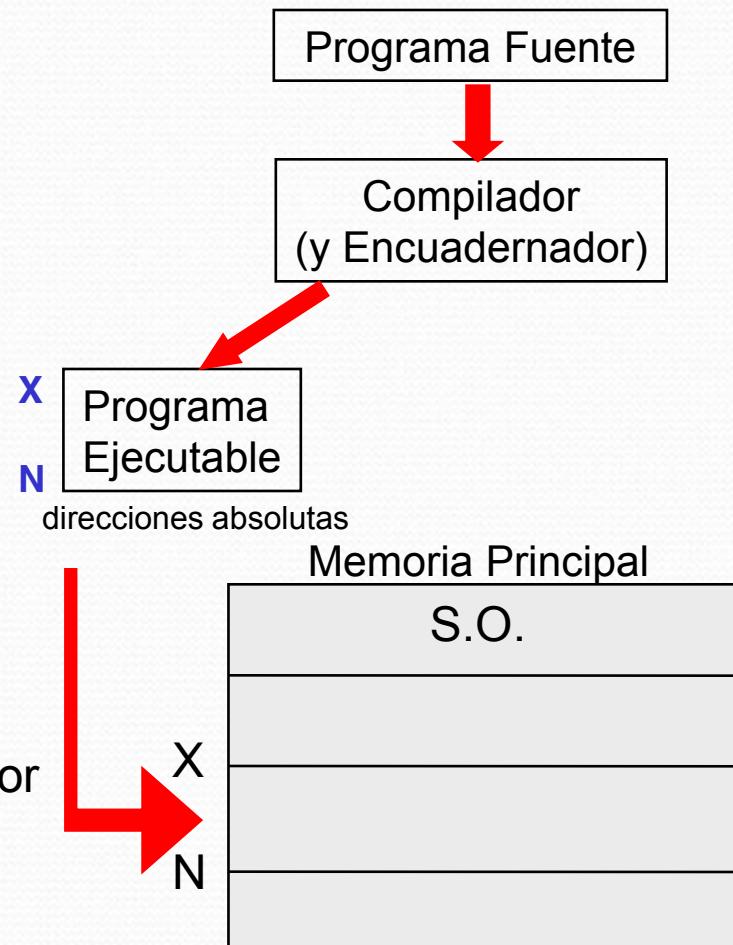
**CLAR98** Clarke, D., and Merusi, D. *System Software Programming: The Way Things Work*. Upper Saddle River, NJ: Prentice Hall, 1998.

**KNUT97** Knuth, D. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1997.

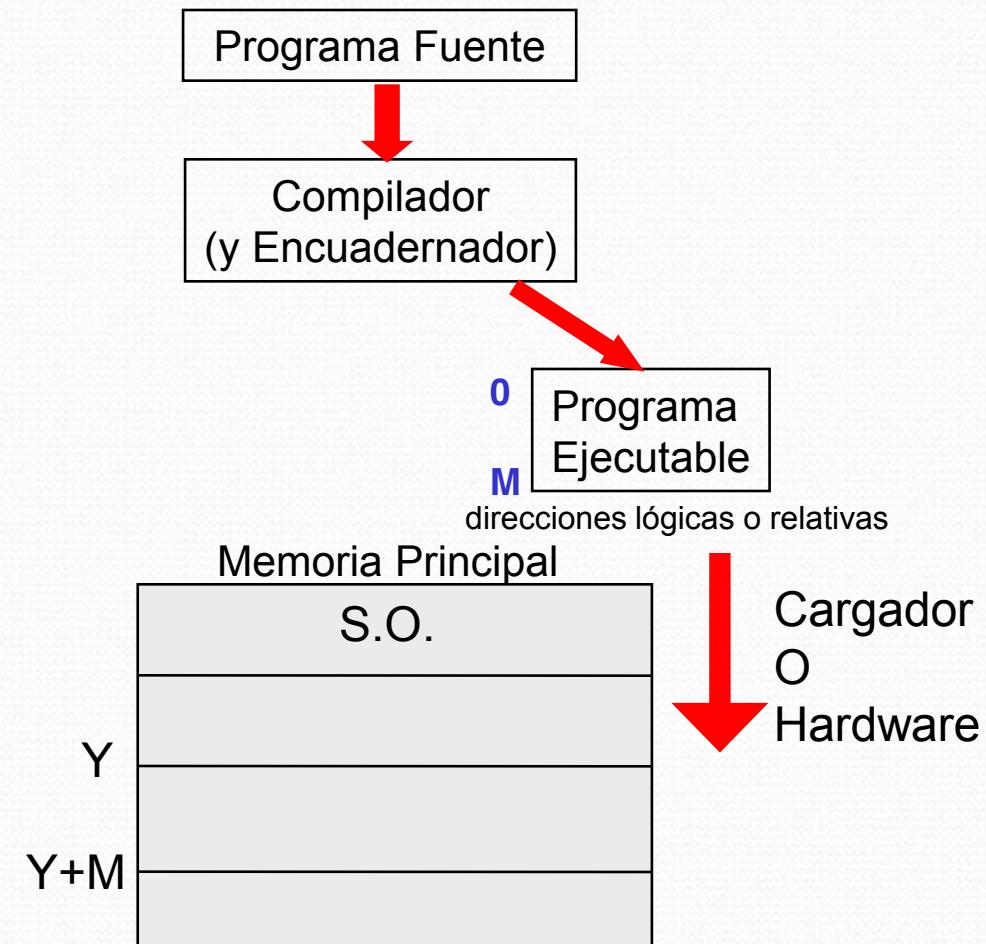
**LEVI99** Levine, J. *Linkers and Loaders*. New York: Elsevier Science and Technology, 1999.

**MILE92** Milenkovic, M. *Operating Systems: Concepts and Design*. New York: McGraw-Hill, 1992.

### Carga absoluta

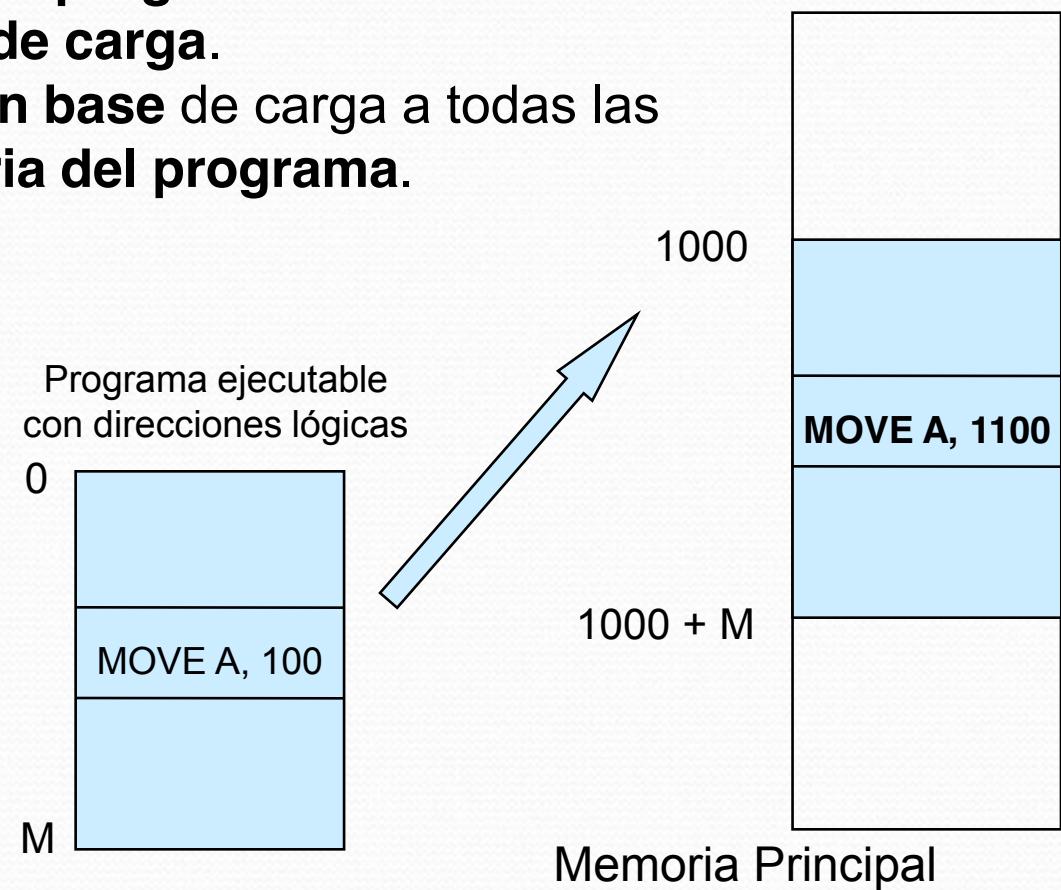


### Reubicación



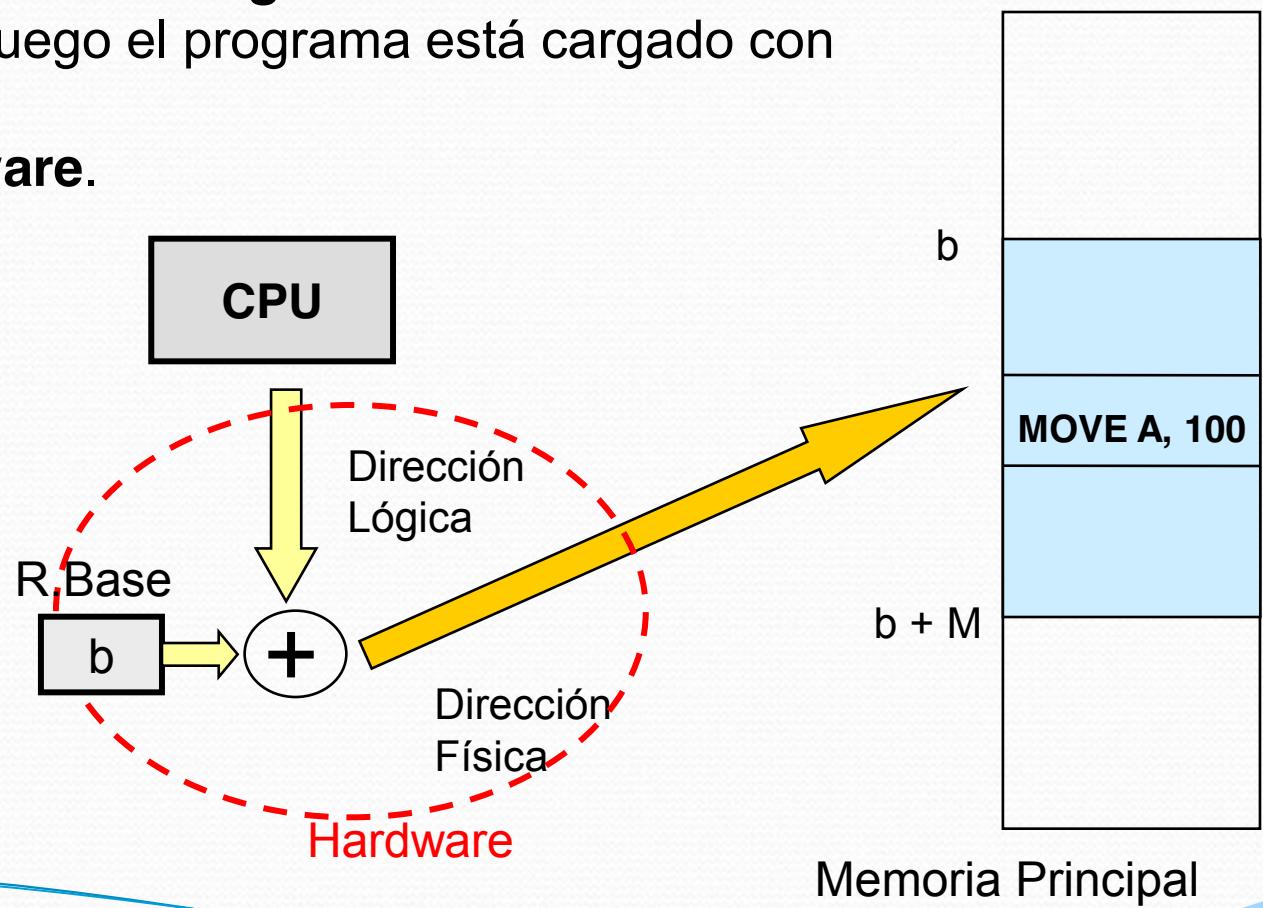
### Reubicación estática

- El **compilador** genera **direcciones lógicas** (relativas) de 0 a M.
- La **decisión de dónde ubicar el programa** en memoria principal se realiza en **tiempo de carga**.
- El **cargador** añade la **dirección base** de carga a todas las **referencias** relativas a **memoria del programa**.



## Reubicación dinámica

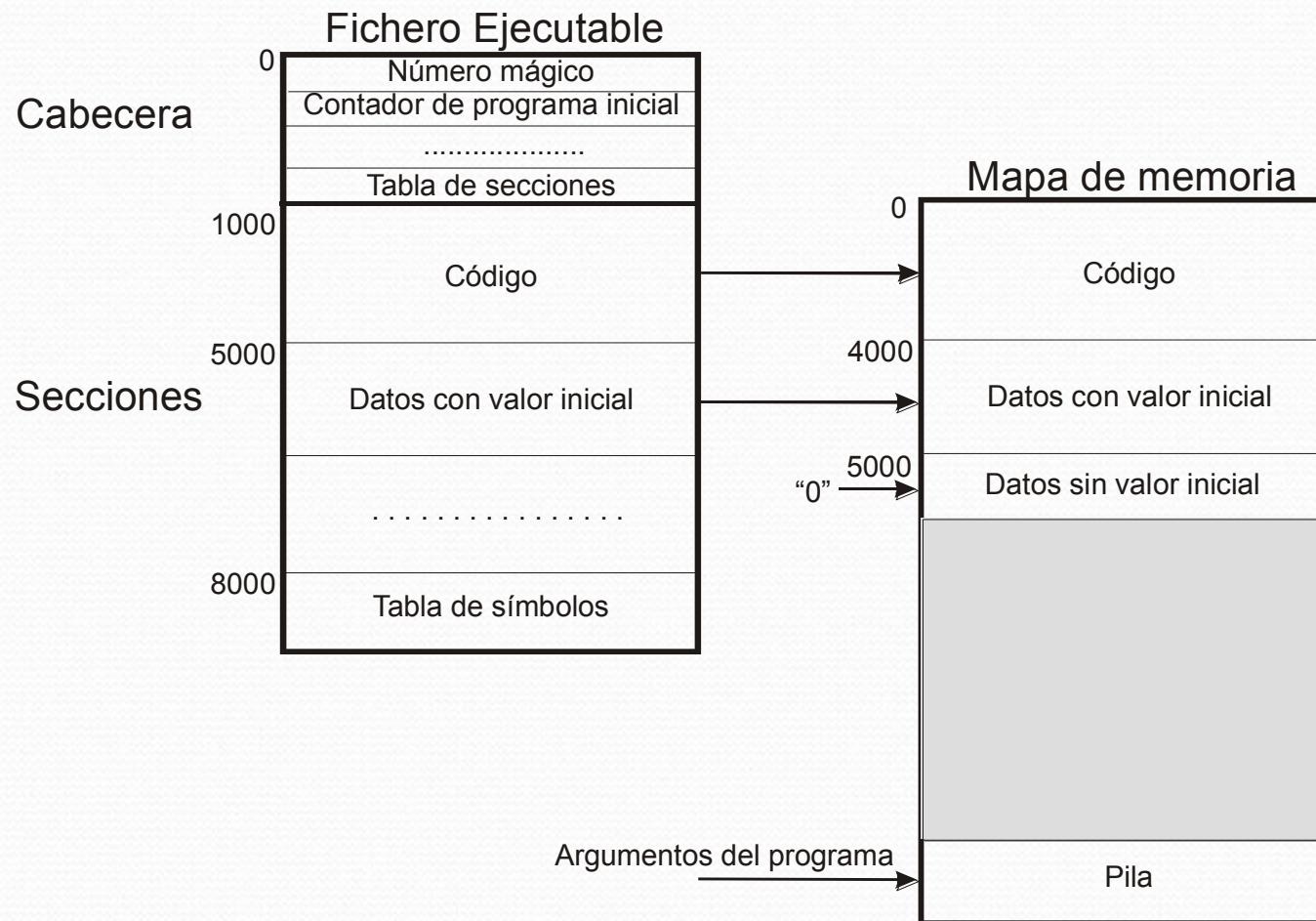
- El **compilador** genera **direcciones lógicas** (relativas) de 0 a M.
- La traducción de **direcciones lógicas a físicas** se realiza en **tiempo de ejecución** luego el programa está cargado con referencias relativas.
- Requiere apoyo **hardware**.



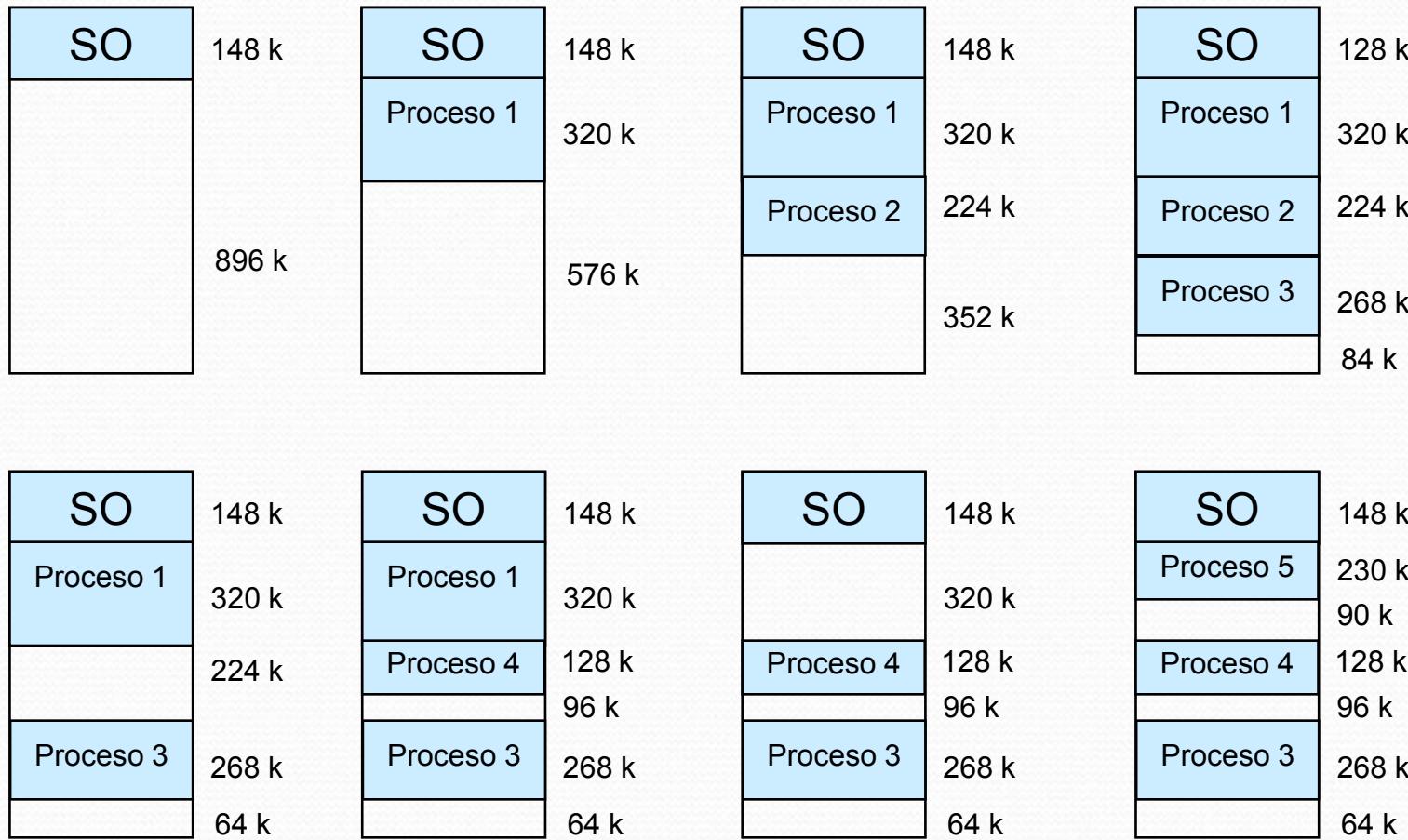
### Espacios para las direcciones de memoria

- **Espacio de direcciones lógico.** Conjunto de direcciones lógicas (o relativas) que utiliza un programa ejecutable.
- **Espacio de direcciones físico.** Conjunto de direcciones físicas (memoria principal) correspondientes a las direcciones lógicas del programa en un instante dado.
- **Mapa de memoria de un ordenador.** Todo el espacio de memoria direccionable por el ordenador. Normalmente depende del tamaño del bus de direcciones.
- **Mapa de memoria de un proceso.** Estructura de datos (que reside en memoria) que indica el tamaño total del espacio de direcciones lógico y la correspondencia entre las direcciones lógicas y las físicas.

## Ejemplo de mapa de memoria de un proceso



### Problema de la fragmentación de memoria



### Solución a la fragmentación de memoria [Stall05] (pp. 340-358)

- Trocear el espacio lógico en unidades más pequeñas: **páginas** (elementos de longitud fija), o **segmentos** (elementos de longitud variable).
- Los trozos no tienen por qué ubicarse consecutivamente en el espacio físico.
- Los esquemas de organización del espacio lógico de direcciones y de traducción de una dirección del espacio lógico al espacio físico que comentaremos son:
  - **Paginación**
  - **Segmentación**



*En el Capítulo 7 se vieron los conceptos de paginación y segmentación y se analizaron sus limitaciones. Ahora vamos a entrar a discutir el concepto de memoria virtual. Un análisis de este concepto es complicado por el hecho de que la gestión de la memoria es una interacción compleja entre el hardware del procesador y el sistema operativo. Nos centraremos primero en los aspectos hardware de la memoria virtual, observando el uso de la paginación, segmentación, y combinación de paginación y segmentación. Despues veremos los aspectos relacionados con el diseño de los servicios de la memoria virtual en el sistema operativo.*



## 8.1. HARDWARE Y ESTRUCTURAS DE CONTROL

**C**omparando la paginación sencilla y la segmentación sencilla, por un lado, tenemos una distinción entre particionamiento estático y dinámico, y por otro, tenemos los fundamentos de comienzo de la gestión de la memoria. Las dos características de la paginación y la segmentación que son la clave de este comienzo son:

1. Todas las referencias a la memoria dentro un proceso se realizan a direcciones lógicas, que se traducen dinámicamente en direcciones físicas durante la ejecución. Esto significa que un proceso puede ser llevado y traído a memoria de forma que ocupe diferentes regiones de la memoria principal en distintos instantes de tiempo durante su ejecución.
2. Un proceso puede dividirse en varias porciones (páginas o segmentos) y estas porciones no tienen que estar localizadas en la memoria de forma contigua durante la ejecución. La combinación de la traducción de direcciones dinámicas en ejecución y el uso de una tabla de páginas o segmentos lo permite.

Ahora veamos cómo comenzar con la memoria dinámica. *Si las dos características anteriores se dan, entonces es necesario que todas las páginas o todos los segmentos de un proceso se encuentren en la memoria principal durante la ejecución. Si la porción (segmento o página) en la que se encuentra la siguiente instrucción a buscar está y si la porción donde se encuentra la siguiente dirección de datos que se va a acceder también está, entonces al menos la siguiente instrucción se podrá ejecutar.*

Consideremos ahora cómo se puede realizar esto. De momento, vamos a hablar en términos generales, y usaremos el término *porción* para referirnos o bien a una página o un segmento, dependiendo si estamos empleando paginación o segmentación. Supongamos que se tiene que traer un nuevo proceso de memoria. El sistema operativo comienza trayendo únicamente una o dos porciones, que incluye la porción inicial del programa y la porción inicial de datos sobre la cual acceden las primeras instrucciones acceden. Esta parte del proceso que se encuentra realmente en la memoria principal para, cualquier instante de tiempo, se denomina **conjunto residente** del proceso. Cuando el proceso está ejecutándose, las cosas ocurren de forma suave mientras que todas las referencias a la memoria se encuentren dentro del conjunto residente. Usando una tabla de segmentos o páginas, el procesador siempre es capaz de determinar si esto es así o no. Si el procesador encuentra una dirección lógica que no se encuentra en la memoria principal, generará una interrupción indicando un fallo de acceso a la memoria. El sistema operativo coloca al proceso interrumpido en un estado de bloqueado y toma el control. Para que la ejecución de este proceso pueda reanudarse más adelante, el sistema operativo necesita traer a la memoria principal la porción del proceso que contiene la dirección lógica que ha causado el fallo de acceso. Con este fin, el sistema operativo realiza una petición de E/S, una lectura a disco. Después de realizar la petición de E/S, el sistema operativo puede activar otro proceso que se ejecute mientras el disco realiza la operación de E/S. Una vez que la porción solicitada se ha traído a

la memoria principal, una nueva interrupción de E/S se lanza, dando control de nuevo al sistema operativo, que coloca al proceso afectado de nuevo en el estado Listo.

Al lector se le puede ocurrir cuestionar la eficiencia de esta maniobra, en la cual a un proceso que se puede estar ejecutando resulta necesario interrumpirlo sin otro motivo que el hecho de que no se ha llegado a cargar todas las porciones necesarias de dicho proceso. De momento, vamos a posponer esta cuestión con la garantía de que la eficiencia es verdaderamente posible. En su lugar, vamos a ponderar las implicaciones de nuestra nueva estrategia. Existen dos implicaciones, la segunda más sorprendente que la primera, y ambas dirigidas a mejorar la utilización del sistema:

1. **Pueden mantenerse un mayor número de procesos en memoria principal.** Debido a que sólo vamos a cargar algunas de las porciones de los procesos a ejecutar, existe espacio para más procesos. Esto nos lleva a una utilización más eficiente del procesador porque es más probable que haya al menos uno o más de los numerosos procesos que se encuentre en el estado Listo, en un instante de tiempo concreto.
2. **Un proceso puede ser mayor que toda la memoria principal.** Se puede superar una de las restricciones fundamentales de la programación. Sin el esquema que hemos estado discutiendo, un programador debe estar realmente atento a cuánta memoria está disponible. Si el programa que está escribiendo es demasiado grande, el programador debe buscar el modo de estructurar el programa en fragmentos que pueden cargarse de forma separada con un tipo de estrategia de superposición (*overlay*). Con la memoria virtual basada en paginación o segmentación, este trabajo se delega al sistema operativo y al hardware. En lo que concierne al programador, él está trabajando con una memoria enorme, con un tamaño asociado al almacenamiento en disco. El sistema operativo automáticamente carga porciones de un proceso en la memoria principal cuando éstas se necesitan.

Debido a que un proceso ejecuta sólo en la memoria principal, esta memoria se denomina **memoria real**. Pero el programador o el usuario perciben una memoria potencialmente mucho más grande —la cual se encuentra localizada en disco. Esta última se denomina **memoria virtual**. La memoria virtual permite una multiprogramación muy efectiva que libera al usuario de las restricciones excesivamente fuertes de la memoria principal. La Tabla 8.1 recoge las características de la paginación y la segmentación, con y sin el uso de la memoria virtual.

## PROXIMIDAD Y MEMORIA VIRTUAL

Los beneficios de la memoria virtual son atractivos, ¿pero el esquema es verdaderamente práctico? En su momento, hubo un importante debate sobre este punto, pero la experiencia de numerosos sistemas operativos ha demostrado, más allá de toda duda, que la memoria virtual realmente funciona. La memoria virtual, basada en paginación o paginación más segmentación, se ha convertido, en la actualidad, en una componente esencial de todos los sistemas operativos contemporáneos.

Para entender cuál es el aspecto clave, y por qué la memoria virtual era la causa de dicho debate, examinemos de nuevo las tareas del sistema operativo relacionadas con la memoria virtual. Se va a considerar un proceso de gran tamaño, consistente en un programa largo más un gran número de vectores de datos. A lo largo de un corto periodo de tiempo, la ejecución se puede acotar a una pequeña sección del programa (por ejemplo, una subrutina) y el acceso a uno o dos vectores de datos únicamente. Si es así, sería verdaderamente un desperdicio cargar docenas de porciones de dicho proceso cuando sólo unas pocas porciones se usarán antes de que el programa se suspenda o se mande a zona de intercambio o *swap*. Se puede hacer un mejor uso de la memoria cargando únicamente unas pocas porciones. Entonces, si el programa salta a una destrucción o hace referencia a un dato que se en-

**Tabla 8.1.** Características de la paginación y la segmentación.

<b>Paginación sencilla</b>	<b>Paginación con memoria virtual</b>	<b>Segmentación sencilla</b>	<b>Segmentación con memoria virtual</b>
Memoria principal particionada en fragmentos pequeños de un tamaño fijo llamados marcos	Memoria principal particionada en fragmentos pequeños de un tamaño fijo llamados marcos	Memoria principal no particionada	Memoria principal no particionada
Programa dividido en páginas por el compilador o el sistema de gestión de la memoria	Programa dividido en páginas por el compilador o el sistema de gestión de la memoria	Los segmentos de programa se especifican por el programador al compilador (por ejemplo, la decisión se toma por parte del programador)	Los segmentos de programa se especifican por el programador al compilador (por ejemplo, la decisión se toma por parte del programador)
Fragmentación interna dentro de los marcos	Fragmentación interna dentro de los marcos	Sin fragmentación interna	Sin fragmentación interna
Sin fragmentación externa	Sin fragmentación externa	Fragmentación externa	Fragmentación externa
El sistema operativo debe mantener una tabla de páginas por cada proceso mostrando en el marco que se encuentra cada página ocupada	El sistema operativo debe mantener una tabla de páginas por cada proceso mostrando en el marco que se encuentra cada página ocupada	El sistema operativo debe mantener una tabla de segmentos por cada proceso mostrando la dirección de carga y la longitud de cada segmento	El sistema operativo debe mantener una tabla de segmentos por cada proceso mostrando la dirección de carga y la longitud de cada segmento
El sistema operativo debe mantener una lista de marcos libres	El sistema operativo debe mantener una lista de marcos libres	El sistema operativo debe mantener una lista de huecos en la memoria principal	El sistema operativo debe mantener una lista de huecos en la memoria principal
El procesador utiliza el número de página, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de página, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de segmento, desplazamiento para calcular direcciones absolutas	El procesador utiliza el número de segmento, desplazamiento para calcular direcciones absolutas
Todas las páginas del proceso deben encontrarse en la memoria principal para que el proceso se pueda ejecutar, salvo que se utilicen solapamientos (overlays)	No se necesita mantener todas las páginas del proceso en los marcos de la memoria principal para que el proceso se ejecute. Las páginas se pueden leer bajo demanda	Todos los segmentos del proceso deben encontrarse en la memoria principal para que el proceso se pueda ejecutar, salvo que se utilicen solapamientos (overlays)	No se necesitan mantener todos los segmentos del proceso en la memoria principal para que el proceso se ejecute. Los segmentos se pueden leer bajo demanda
			La lectura de un segmento a memoria principal puede requerir la escritura de uno o más segmentos a disco

cuenta en una porción de memoria que no está en la memoria principal, entonces se dispara un fallo. Éste indica al sistema operativo que debe conseguir la porción deseada.

Así, en cualquier momento, sólo unas pocas porciones de cada proceso se encuentran en memoria, y por tanto se pueden mantener más procesos alojados en la misma. Además, se ahorra tiempo porque las porciones del proceso no usadas no se expulsarán de la memoria a *swap* y de *swap* a la memoria. Sin embargo, el sistema operativo debe ser inteligente a la hora de manejar este esquema. En estado estable, prácticamente toda la memoria principal se encontrará ocupada con porciones de procesos, de forma que el procesador y el sistema operativo tengan acceso directo al mayor número posible de procesos. Así, cuando el sistema operativo traiga una porción a la memoria, debe expulsar otra. Si elimina una porción justo antes de que vaya a ser utilizada, deberá recuperar dicha porción de nuevo casi de forma inmediata. Un abuso de esto lleva a una condición denominada **trasiego (*thrashing*)**: el sistema consume la mayor parte del tiempo enviando y trayendo porciones de *swap* en lugar de ejecutar instrucciones. Evitar el trasiego fue una de las áreas de investigación principales en la época de los años 70 que condujo una gran variedad de algoritmos complejos pero muy efectivos. En esencia, el sistema operativo trata de adivinar, en base a la historia reciente, qué porciones son menos probables de ser utilizadas en un futuro cercano.

Este razonamiento se basa en la creencia del **principio de proximidad**, que se presentó en el Capítulo 1 (véase especialmente el Apéndice 1A). Para resumir, el principio de proximidad indica que las referencias al programa y a los datos dentro de un proceso tienden a agruparse. Por tanto, se resume que sólo unas pocas porciones del proceso se necesitarán a lo largo de un periodo de tiempo corto. También, es posible hacer suposiciones inteligentes sobre cuáles son las porciones del proceso que se necesitarán en un futuro próximo, para evitar este trasiego.

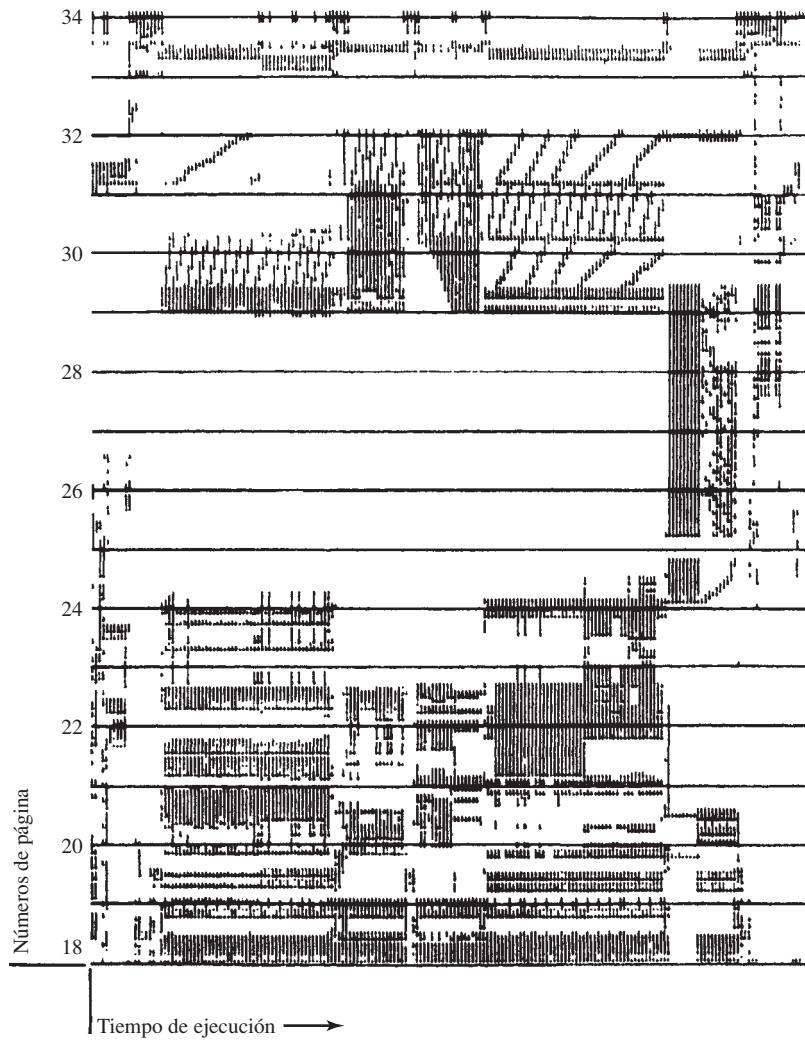
Una forma de confirmar el principio de proximidad es observar el rendimiento de los procesos en un entorno de memoria virtual. En la Figura 8.1 se muestra un famoso diagrama que ilustra de forma clara los principios de proximidad [HATF 72]. Nótese que, durante el tiempo de vida de un proceso las referencias se encuentran acotadas a un subconjunto de sus páginas.

Así pues, vemos que el principio de proximidad sugiere que el esquema de memoria virtual debe funcionar. Para que la memoria virtual resulte práctica y efectiva, se necesitan dos ingredientes. Primero, debe existir un soporte hardware para el esquema de paginación y/o segmentación. Segundo, el sistema operativo debe incluir código para gestionar el movimiento de páginas y/o segmentos entre la memoria secundaria y la memoria principal. En esta sección, examinaremos los aspectos hardware y veremos cuáles son las estructuras de control necesarias, que se crearán y mantendrán por parte del sistema operativo pero que son usadas por el hardware de gestión de la memoria. Se examinarán los aspectos correspondientes al sistema operativo en la siguiente sección.

## PAGINACIÓN

El término *memoria virtual* se asocia habitualmente con sistemas que emplean paginación, a pesar de que la memoria virtual basada en segmentación también se utiliza y será tratada más adelante. El uso de paginación para conseguir memoria virtual fue utilizado por primera vez en el computador Atlas [KILB62] y pronto se convirtió en una estrategia usada en general de forma comercial.

En la presentación de la paginación sencilla, indicamos que cada proceso dispone de su propia tabla de páginas, y que todas las páginas se encuentran localizadas en la memoria principal. Cada entrada en la tabla de páginas consiste en un número de marco de la correspondiente página en la memoria principal. Para la memoria virtual basada en el esquema de paginación también se necesita una tabla de páginas. De nuevo, normalmente se asocia una única tabla de páginas a cada proceso. En este caso, sin embargo, las entradas de la tabla de páginas son más complejas (Figura 8.2a). Debido a que



**Figura 8.1.** Comportamiento de la paginación.

sólo algunas de las páginas de proceso se encuentran en la memoria principal, se necesita que cada entrada de la tabla de páginas indique si la correspondiente página está presente (P) en memoria principal o no. Si el bit indica que la página está en memoria, la entrada también debe indicar el número de marco de dicha página.

La entrada de la tabla de páginas incluye un bit de modificado (M), que indica si los contenidos de la correspondiente página han sido alterados desde que la página se cargó por última vez en la memoria principal. Si no había ningún cambio, no es necesario escribir la página cuando llegue el momento de reemplazarla por otra página en el marco de página que actualmente ocupa. Pueden existir también otros bits de control en estas entradas. Por ejemplo, si la protección y compartición se gestiona a nivel de página, se necesitarán también los bits para este propósito.

**Estructura de la tabla de páginas.** El mecanismo básico de lectura de una palabra de la memoria implica la traducción de la dirección virtual, o lógica, consistente en un número de página y un des-

Dirección virtual

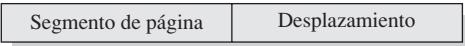


Entrada de la tabla de páginas



(a) Únicamente paginación

Dirección virtual



Entrada de la tabla de segmentos



(b) Únicamente segmentación

Dirección virtual



Entrada de la tabla de segmentos



Entrada de la tabla de páginas



P = bit de presente  
M = bit de modificado

(c) Combinación de segmentación y paginación

**Figura 8.2.** Formatos típicos de gestión de memoria.

plazamiento, a la dirección física, consistente en un número de marco y un desplazamiento, usando para ello la tabla de páginas. Debido a que la tabla de páginas es de longitud variable dependiendo del tamaño del proceso, no podemos suponer que se encuentra almacenada en los registros. En lugar de eso, debe encontrarse en la memoria principal para poder ser accedida. La Figura 8.3 sugiere una implementación hardware. Cuando un proceso en particular se encuentra ejecutando, un registro contiene la dirección de comienzo de la tabla de páginas para dicho proceso. El número de página de la dirección virtual se utiliza para indexar esa tabla y buscar el correspondiente marco de página. Éste, combinado con la parte de desplazamiento de la dirección virtual genera la dirección real deseada. Normalmente, el campo correspondiente al número de página es mayor que el campo correspondiente al número de marco de página ( $n > m$ ).

En la mayoría de sistemas, existe una única tabla de página por proceso. Pero cada proceso puede ocupar una gran cantidad de memoria virtual. Por ejemplo, en la arquitectura VAX, cada proceso puede tener hasta  $2^{31} = 2$  Gbytes de memoria virtual. Usando páginas de  $2^9 = 512$  bytes, que representa un total de  $2^{22}$  entradas de tabla de página *por cada proceso*. Evidentemente, la cantidad de memoria demandada por las tablas de página únicamente puede ser inaceptablemente grande. Para resolver este problema, la mayoría de esquemas de memoria virtual almacena las ta-

blas de páginas también en la memoria virtual, en lugar de en la memoria real. Esto representa que las tablas de páginas están sujetas a paginación igual que cualquier otra página. Cuando un proceso está en ejecución, al menos parte de su tabla de páginas debe encontrarse en memoria, incluyendo la entrada de tabla de páginas de la página actualmente en ejecución. Algunos procesadores utilizan un esquema de dos niveles para organizar las tablas de páginas de gran tamaño. En este esquema, existe un directorio de páginas, en el cual cada entrada apuntaba a una tabla de páginas. De esta forma, si la extensión del directorio de páginas es  $X$ , y si la longitud máxima de una tabla de páginas es  $Y$ , entonces un proceso consistirá en hasta  $X \geq Y$  páginas. Normalmente, la longitud máxima de la tabla de páginas se restringe para que sea igual a una página. Por ejemplo, el procesador Pentium utiliza esta estrategia.

La Figura 8.4 muestra un ejemplo de un esquema típico de dos niveles que usa 32 bits para la dirección. Asumimos un direccionamiento a nivel de byte y páginas de 4 Kbytes ( $2^{12}$ ), por tanto el espacio de direcciones virtuales de 4 Gbytes ( $2^{32}$ ) se compone de  $2^{20}$  páginas. Si cada una de estas páginas se referencia por medio de una entrada la tabla de páginas (ETP) de 4-bytes, podemos crear una tabla de página de usuario con  $2^{20}$  la ETP que requiere 4 Mbytes ( $2^{22}$  bytes). Esta enorme tabla de páginas de usuario, que ocupa  $2^{10}$  páginas, puede mantenerse en memoria virtual y hacerse referencia desde una tabla de páginas raíz con  $2^{10}$  PTE que ocuparía 4 Kbytes ( $2^{12}$ ) de memoria principal. La Figura 8.5 muestra los pasos relacionados con la traducción de direcciones para este esquema. La página raíz siempre se mantiene en la memoria principal. Los primeros 10 bits de la dirección virtual se pueden usar para indexar en la tabla de páginas raíz para encontrar la ETP para la página en la que está la tabla de páginas de usuario. Si la página no está en la memoria principal, se produce un fallo de página. Si la página está en la memoria principal, los siguientes 10 bits de la dirección virtual se usan para indexar la tabla de páginas de usuario para encontrar la ETP de la página a la cual se hace referencia desde la dirección virtual original.

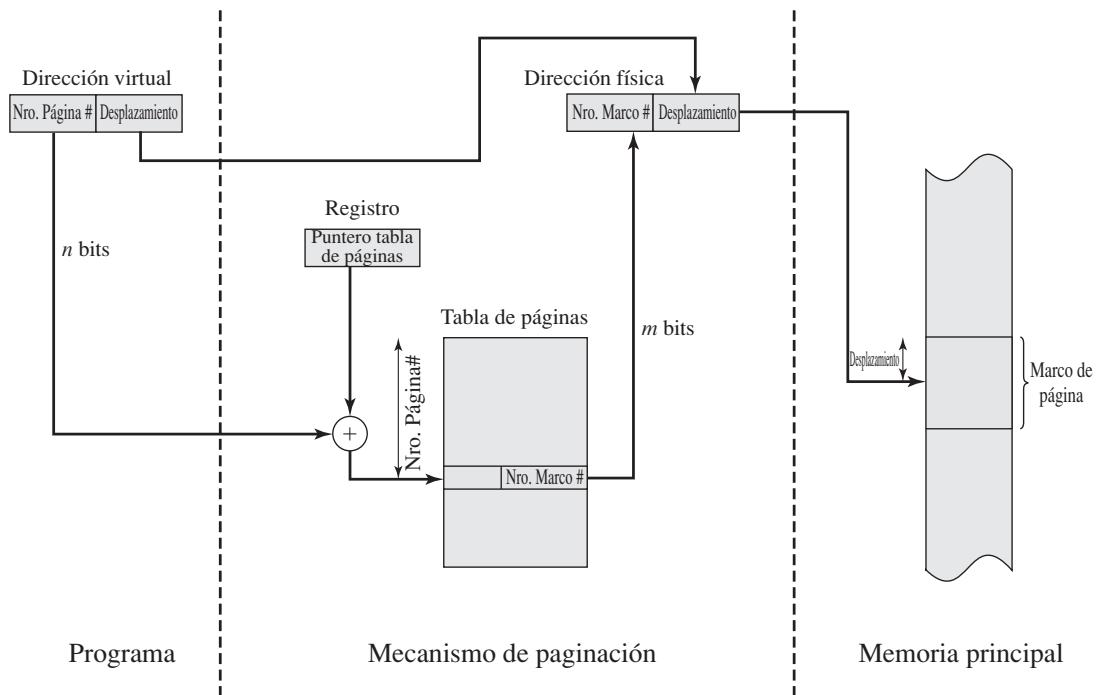


Figura 8.3. Traducción de direcciones en un sistema con paginación.

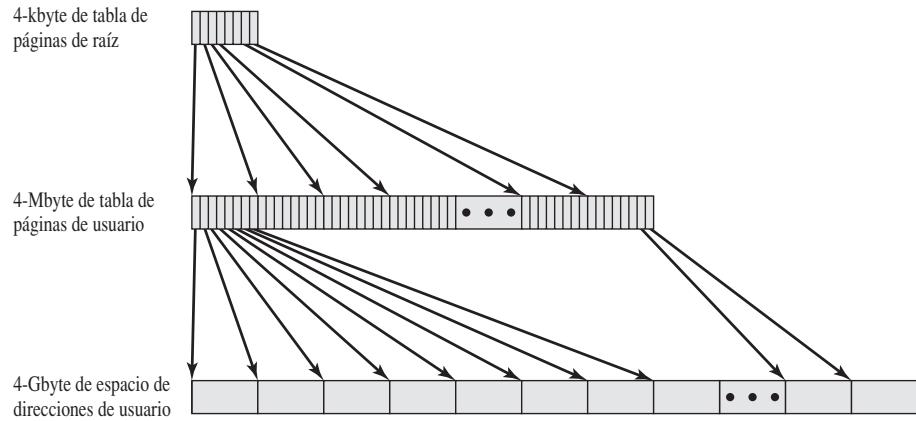


Figura 8.4. Una tabla de páginas jerárquica de dos niveles.

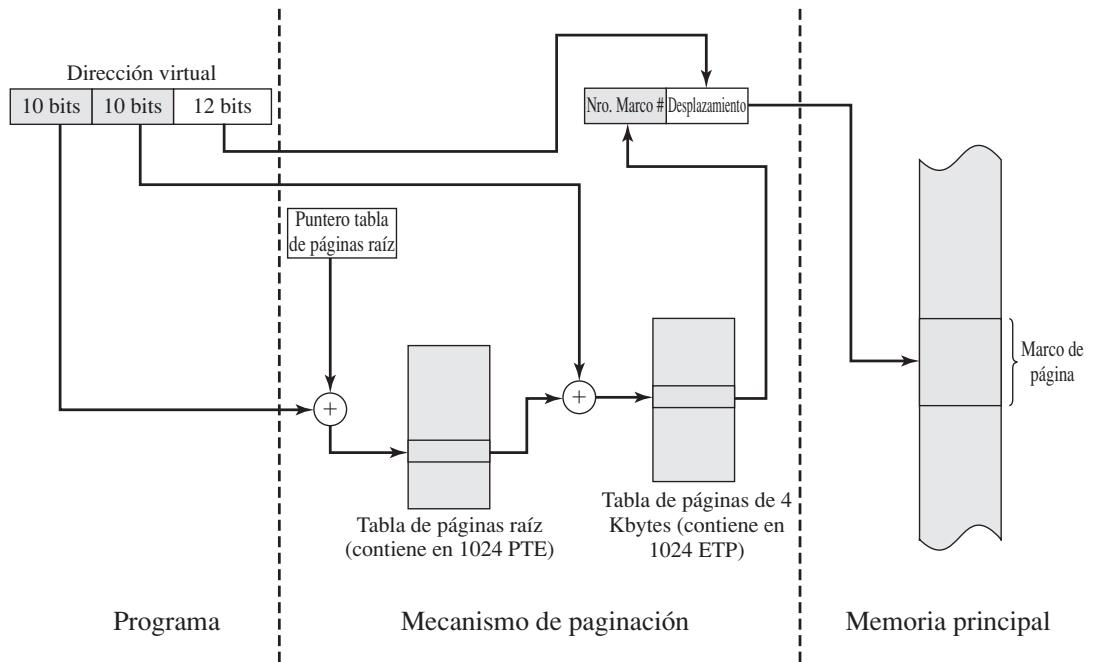
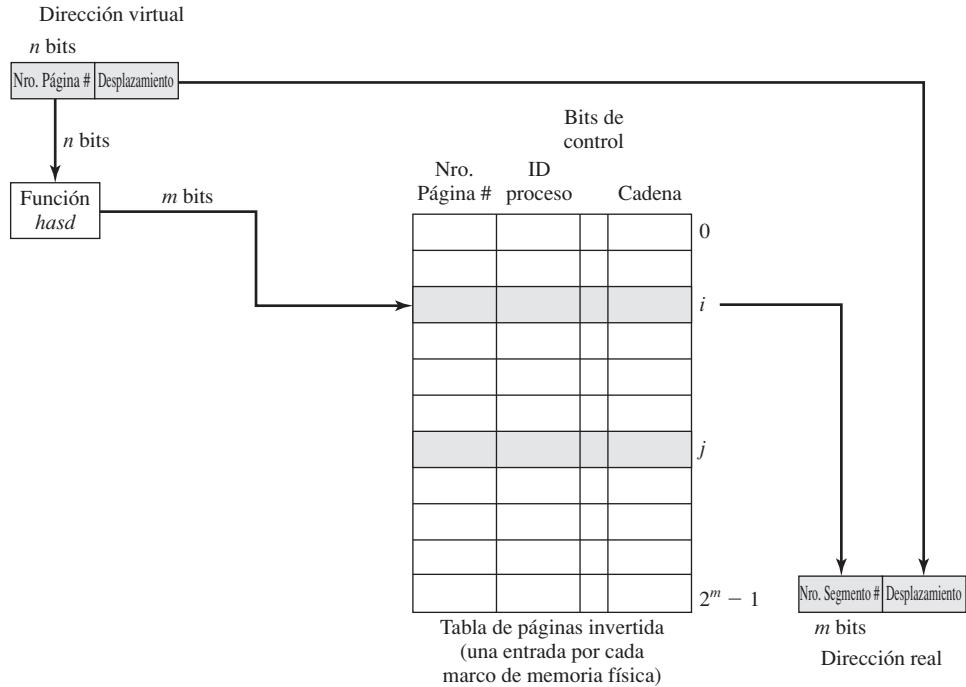


Figura 8.5. Traducción de direcciones en un sistema de paginación de dos niveles.

**Tabla de páginas invertida.** Una desventaja del tipo de tablas de páginas que hemos visto es que su tamaño es proporcional al espacio de direcciones virtuales.

Una estrategia alternativa al uso de tablas de páginas de uno o varios niveles es el uso de la estructura de **tabla de páginas invertida**. Variaciones de esta estrategia se han usado en arquitecturas como PowerPC, UltraSPARC, e IA-64. La implementación del sistema operativo Mach sobre RT-PC también la usa.



**Figura 8.6.** Estructura de tabla de páginas invertida.

En esta estrategia, la parte correspondiente al número de página de la dirección virtual se referencia por medio de un valor *hash* usando una función *hash* sencilla<sup>1</sup>. El valor *hash* es un puntero para la tabla de páginas invertida, que contiene las entradas de tablas de página. Hay una entrada en la tabla de páginas invertida por cada marco de página real en lugar de uno por cada página virtual. De esta forma, lo único que se requiere para estas tablas de página siempre es una proporción fija de la memoria real, independientemente del número de procesos o de las páginas virtuales soportadas. Debido a que más de una dirección virtual puede traducirse en la misma entrada de la tabla *hash*, una técnica de encadenamiento se utiliza para gestionar el desbordamiento. Las técnicas de *hashing* proporcionan habitualmente cadenas que no son excesivamente largas —entre una y dos entradas. La estructura de la tabla de páginas se denomina invertida debido a que se indexan sus entradas de la tabla de páginas por el número de marco en lugar de por el número de página virtual.

La Figura 8.6 muestra una implementación típica de la técnica de tabla de páginas invertida. Para un tamaño de memoria física de  $2^m$  marcos, la tabla de páginas invertida contiene  $2^m$  entradas, de forma que la entrada en la posición  $i$ -esima se refiere al marco  $i$ . La entrada en la tabla de páginas incluye la siguiente información:

- **Número de página.** Esta es la parte correspondiente al número de página de la dirección virtual.

<sup>1</sup> Véase Apéndice 8A para explicaciones sobre *hashing*.

- **Identificador del proceso.** El proceso que es propietario de esta página. La combinación de número de página e identificador del proceso identifica a una página dentro del espacio de direcciones virtuales de un proceso en particular.
- **Bits de control.** Este campo incluye los *flags*, como por ejemplo, válido, referenciado, y modificado; e información de protección y cerrojos.
- **Puntero de la cadena.** Este campo es nulo (indicado posiblemente por un bit adicional) si no hay más entradas encadenadas en esta entrada. En otro caso, este campo contiene el valor del índice (número entre 0 y  $2^{m-1}$ ) de la siguiente entrada de la cadena.

En este ejemplo, la dirección virtual incluye un número de página de  $n$  bits, con  $n > m$ . La función *hash* traduce el número de página  $n$  bits en una cantidad de  $m$  bits, que se utiliza para indexar en la tabla de páginas invertida.

**Buffer de traducción anticipada.** En principio, toda referencia a la memoria virtual puede causar dos accesos a memoria física: uno para buscar la entrada la tabla de páginas apropiada y otro para buscar los datos solicitados. De esa forma, un esquema de memoria virtual básico causaría el efecto de duplicar el tiempo de acceso a la memoria. Para solventar este problema, la mayoría de esquemas de la memoria virtual utilizan una *cache* especial de alta velocidad para las entradas de la tabla de página, habitualmente denominada *buffer de traducción anticipada* (*translation lookaside buffer - TLB*)<sup>2</sup>. Esta *cache* funciona de forma similar a una memoria *cache* general (véase Capítulo 1) y contiene aquellas entradas de la tabla de páginas que han sido usadas de forma más reciente. La organización del hardware de paginación resultante se ilustra en la Figura 8.7. Dada una dirección virtual, el procesador primero examina la TLB, si la entrada de la tabla de páginas solicitada está presente (*acuerdo en TLB*), entonces se recupera el número de marco y se construye la dirección real. Si la entrada de la tabla de páginas solicitada no se encuentra (*fallo en la TLB*), el procesador utiliza el número de página para indexar la tabla de páginas del proceso y examinar la correspondiente entrada de la tabla de páginas. Si el bit de presente está puesto a 1, entonces la página se encuentra en memoria principal, y el procesador puede recuperar el número de marco desde la entrada de la tabla de páginas para construir la dirección real. El procesador también autorizará la TLB para incluir esta nueva entrada de tabla de páginas. Finalmente, si el bit presente no está puesto a 1, entonces la página solicitada no se encuentra en la memoria principal y se produce un fallo de acceso memoria, llamado **fallo de página**. En este punto, abandonamos el dominio del hardware para invocar al sistema operativo, el cual cargará la página necesaria y actualizada de la tabla de páginas.

La Figura 8.8 muestra un diagrama de flujo del uso de la TLB. Este diagrama de flujo muestra como si una página solicitada no se encuentra en la memoria principal, una interrupción de fallo de página hace que se invoque a la rutina de tratamiento de dicho fallo de página. Para mantener la simplicidad de este diagrama, no se ha mostrado el hecho de que el sistema operativo pueda activar otro proceso mientras la operación de E/S sobre disco se está realizando. Debido al principio de proximidad, la mayoría de referencias a la memoria virtual se encontrarán situadas en una página recientemente utilizada y por tanto, la mayoría de referencias invocarán una entrada de la tabla de páginas que se encuentra en la *cache*. Los estudios sobre la TLB de los sistemas VAX han demostrado que este esquema significa una importante mejora del rendimiento [CLAR85, SATY81].

Hay numerosos detalles adicionales relativos a la organización real de la TLB. Debido a que la TLB sólo contiene algunas de las entradas de toda la tabla de páginas, no es posible indexar simple-

---

<sup>2</sup> N. de T. Aunque la traducción más apropiada del *translation lookaside buffer* quizás sea la de *buffer de traducción anticipada*, en la literatura en castellano se utilizan las siglas TLB de forma generalizada para describir dicha memoria. Por ello, y para no causar confusión con otros textos, a lo largo del presente libro utilizaremos dichas siglas para referirnos a ella.

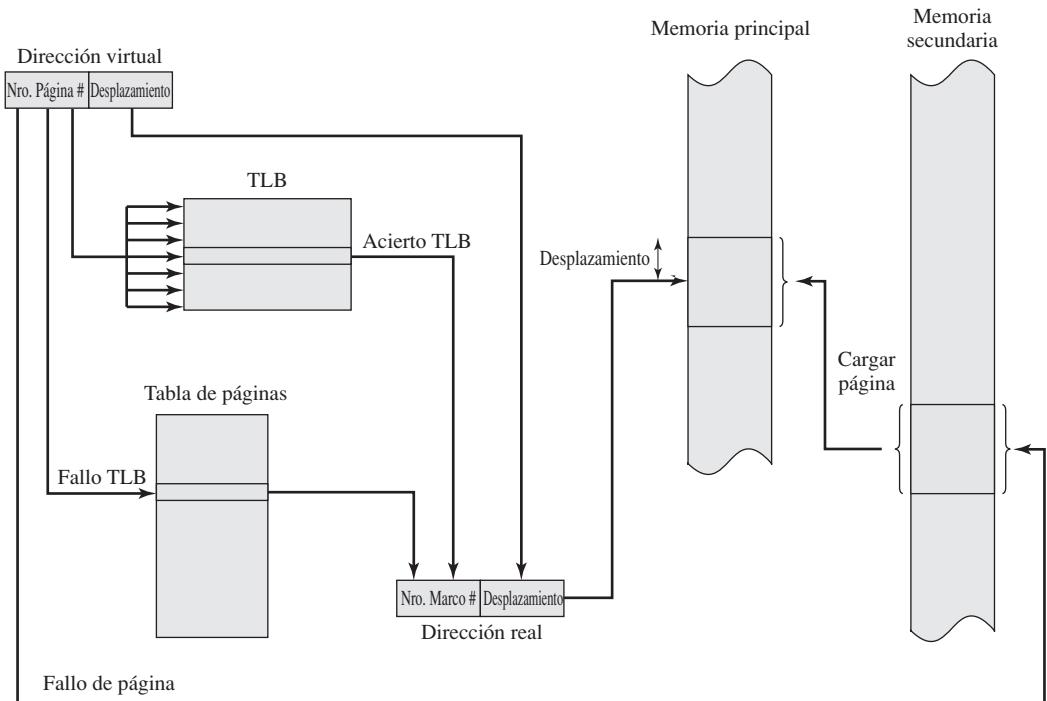


Figura 8.7. Uso de la TLB.

mente la TLB por medio de número página. En lugar de eso, cada entrada de la TLB debe incluir un número de página así como la entrada de la tabla de páginas completa. El procesador proporciona un hardware que permite consultar simultáneamente varias entradas para determinar si hay una conciencia sobre un número de página. Esta técnica se denomina **resolución asociativa (associative mapping)** que contrasta con la resolución directa, o indexación, utilizada para buscar en la tabla de páginas en la Figura 8.9. El diseño de la TLB debe considerar también la forma mediante la cual las entradas se organizan en ella y qué entrada se debe reemplazar cuando se necesite traer una nueva entrada. Estos aspectos deben considerarse en el diseño de la cache hardware. Este punto no se contempla en este libro; el lector podrá consultar el funcionamiento del diseño de una cache para más detalle en, por ejemplo, [STAL03].

Para concluir, el mecanismo de memoria virtual debe interactuar con el sistema de *cache* (no la *cache* de TLB, sino la *cache* de la memoria principal). Esto se ilustra en la Figura 8.10. Una dirección virtual tendrá generalmente el formato número de página, desplazamiento. Primero, el sistema de memoria consulta la TLB para ver si se encuentra presente una entrada de tabla de páginas que coincide. Si es así, la dirección real (física) se genera combinando el número de marco con el desplazamiento. Si no, la entrada se busca en la tabla de páginas. Una vez se ha generado la dirección real, que mantiene el formato de etiqueta (*tag*)<sup>3</sup> y resto (*remainder*), se consulta la cache para ver si el bloque que contiene esa palabra se encuentra ahí. Si es así, se le devuelve a la CPU. Si no, la palabra se busca en la memoria principal.

<sup>3</sup> Véase en la Figura 1.17. Normalmente, una etiqueta son los bits situados más a la izquierda de una dirección real. Una vez más, para un estudio más detallado sobre las caches, se refiere al lector a [STAL03].

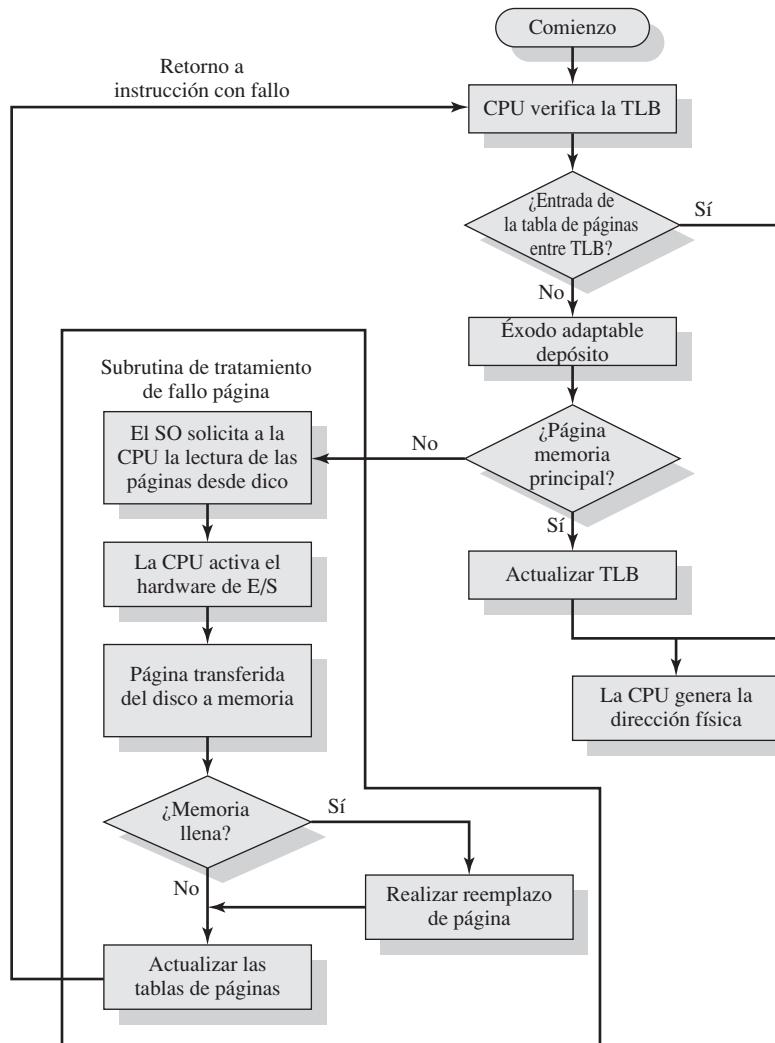


Figura 8.8. Operación de paginación y TLB [FURH87].

El lector podrá apreciar la complejidad del hardware de la CPU que participa en una referencia a memoria sencilla. La dirección virtual se traduce a una dirección real lo cual implica una referencia a la entrada de la tabla de páginas, que puede estar en la TLB, en la memoria principal, o en disco. La palabra referenciada puede estar en la cache, en la memoria principal, o en disco. Si dicha palabra referenciada se encuentra únicamente en disco, la página que contiene dicha palabra debe cargarse en la memoria principal y su bloque en la cache. Adicionalmente, la entrada en la tabla de páginas para dicha página debe actualizarse.

**Tamaño de página.** Una decisión de diseño hardware importante es el tamaño de página a usar. Hay varios factores a considerar. Por un lado, está la fragmentación interna. Evidentemente, cuanto mayor es el tamaño de la página, menor cantidad de fragmentación interna. Para optimizar el uso de la memoria principal, sería beneficioso reducir la fragmentación interna. Por otro lado, cuanto menor es la página, mayor número de páginas son necesarias para cada proceso. Un mayor número de páginas por proceso significa también mayores tablas de páginas. Para programas grandes en un entorno altamente multipro-

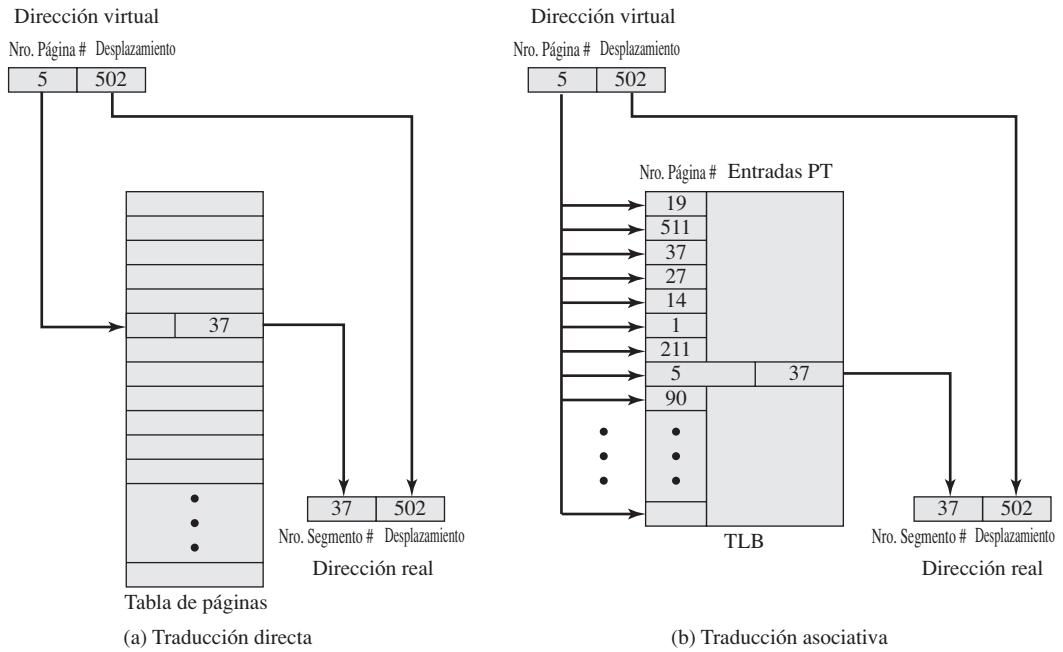


Figura 8.9. Resolución directa vs. asociativa para las entradas en la tabla de páginas.

gramado, esto significa que determinadas partes de las tablas de página de los procesos activos deben encontrarse en la memoria virtual, no en la memoria principal. Por tanto, puede haber un fallo de página doble para una referencia sencilla a memoria: el primero para atraer la tabla de página de la parte solicitada y el segundo para atraer la página del propio proceso. Otro factor importante son las características físicas de la mayoría de los dispositivos de la memoria secundaria, que son de tipo giratorio, favoreciendo tamaños de página grandes para mejorar la eficiencia de transferencia de bloques de datos.

Aumentando la complejidad de estos aspectos se encuentra el efecto que el tamaño de página tiene en relación a la posibilidad de que ocurra un fallo de página. Este comportamiento en términos generales, se encuentra recogido en la Figura 8.11a que se basa en el principio de proximidad. Si el tamaño de página es muy pequeño, de forma habitual habrá un número relativamente alto de páginas disponibles en la memoria principal para cada proceso. Después de un tiempo, las páginas en memoria contendrán las partes de los procesos a las que se ha hecho referencia de forma reciente. De esta forma, la tasa de fallos de página debería ser baja. A medida que el tamaño de páginas se incrementa, la página en particular contendrá información más lejos de la última referencia realizada. Así pues, el efecto del principio de proximidad se debilita y la tasa de fallos de página comienza a crecer. En algún momento, sin embargo, la tasa de fallos de página comenzará a caer a medida que el tamaño de la página se aproxima al tamaño del proceso completo (punto  $P$  en el diagrama). Cuando una única página contiene el proceso completo, no habrá fallos de página.

Una complicación adicional es que la tasa de fallos de página también viene determinada por el número de marcos asociados a cada proceso. La Figura 8.11b muestra que, para un tamaño de página fijo, la tasa de fallos cae a medida que el número de páginas mantenidas en la memoria principal crece<sup>4</sup>. Por

<sup>4</sup> El parámetro  $W$  representa el conjunto de trabajo, un concepto que se analizará en la Sección 8.2.

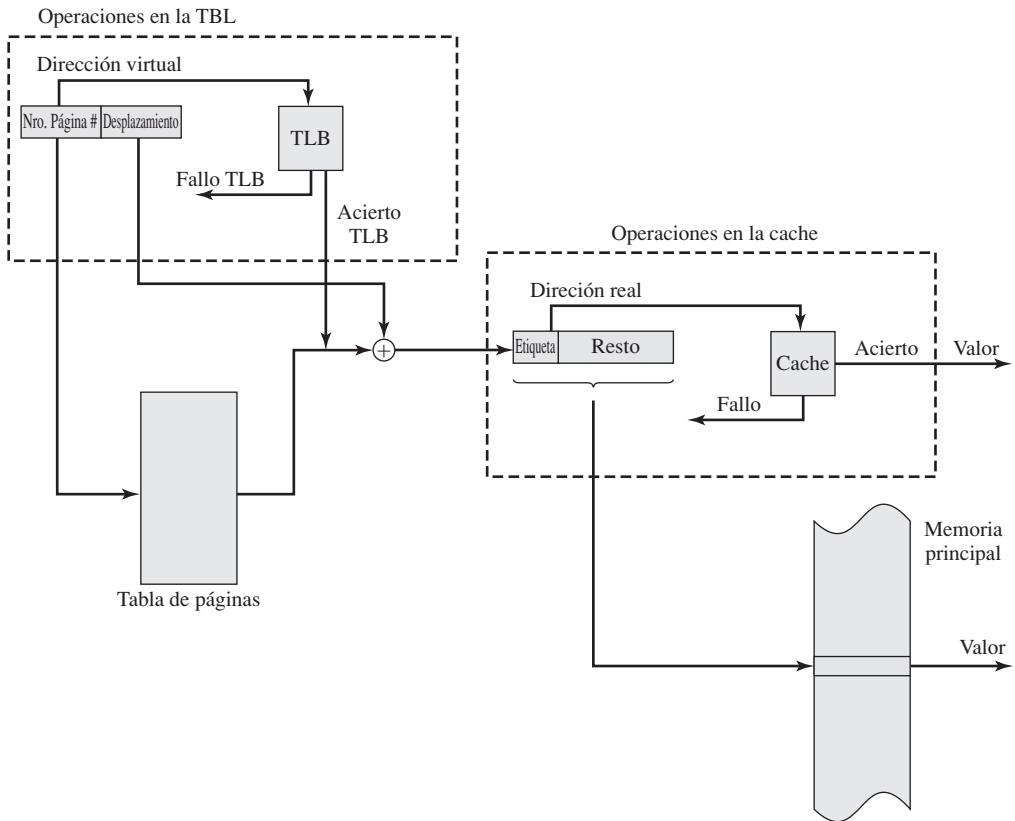
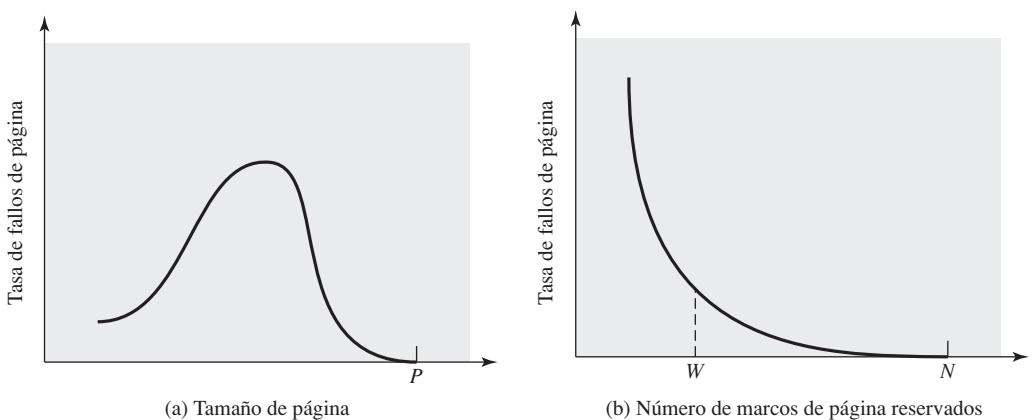


Figura 8.10. Operaciones en la TBL y en la cache.



$P$  = tamaño del proceso entero  
 $W$  = conjunto de trabajo  
 $N$  = número total de páginas en proceso

Figura 8.11. Comportamiento típico de la paginación de un programa.

tanto, una política software (la cantidad de memoria reservada por cada proceso) interactúa con decisiones de diseño del propio hardware (tamaño de página).

La Tabla 8.2 contiene un listado de los tamaños de páginas que tienen determinadas arquitecturas.

Para concluir, el aspecto de diseño del tamaño página se encuentra relacionado con el tamaño de la memoria física y el tamaño del programa. Al mismo tiempo que la memoria principal está siendo cada vez más grande, el espacio de direcciones utilizado por las aplicaciones también crece. Esta tendencia resulta más evidente en ordenadores personales y estaciones de trabajo, donde las aplicaciones tienen una complejidad creciente. Por contra, diversas técnicas de programación actuales usadas para programas de gran tamaño tienden a reducir el efecto de la proximidad de referencias dentro un proceso [HUCK93]. Por ejemplo,

- Las técnicas de programación orientada a objetos motivan el uso de muchos módulos de datos y programas de pequeño tamaño con referencias repartidas sobre un número relativamente alto de objetos en un periodo de tiempo bastante corto.
- Las aplicaciones multihilo (*multithreaded*) pueden presentar cambios abruptos en el flujo de instrucciones y referencias a la memoria fraccionadas.

**Tabla 8.2.** Ejemplo de tamaños de página.

Computer	Tamaño de página
Atlas	512 palabras de 48-bits
Honeywell-Multics	1024 palabras de 36-bits
IBM 370/XA y 370/ESA	4 Kbytes
Familia VAX	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes hasta 16 Mbytes
UltraSPARC	8 Kbytes hasta 4 Mbytes
Pentium	4 Kbytes o 4 Mbytes
PowerPc	4 Kbytes
Itanium	4 Kbytes hasta 256 Mbytes

Para un tamaño determinado de una TLB, a medida que el tamaño del proceso crece y la proximidad de referencias decrece, el índice de aciertos en TLB se va reduciendo. Bajo estas circunstancias, la TLB se puede convertir en el cuello de botella del rendimiento (por ejemplo, véase [CHEN92]).

Una forma de incrementar el rendimiento en la TLB es utilizar una TLB de gran tamaño, con más entradas. Sin embargo, el tamaño de TLB interactúa con otros aspectos del diseño hardware, por ejemplo la *cache* de memoria principal o el número de accesos a memoria por ciclo de instrucción [TALL92]. Una de las principales pegas es que el tamaño de la TLB no tiene la misma tendencia de crecimiento que el tamaño de la memoria principal, en velocidad de crecimiento. Como alternativa se encuentra el uso de tamaños de página mayores de forma que cada entrada en la tabla de páginas referenciada en la TLB apunte a un bloque de memoria relativamente grande. Pero acabamos de ver que el uso de tamaños de página muy grandes puede significar la degradación del rendimiento.

Sobre estas consideraciones, un gran número de diseñadores han investigado la posibilidad de utilizar múltiples tamaños de página [TALL92, KHAL93], y diferentes arquitecturas de microprocesadores dan soporte a diversos tamaños de página, incluyendo MIPS R4000, Alpha, UltraSPARC, Pentium, e IA-64. Los tamaños de página múltiples proporcionan la flexibilidad necesaria para el uso de la TLB de forma eficiente. Por ejemplo, regiones contiguas de memoria de gran tamaño dentro del espacio direcciones del proceso, como las instrucciones del programa, se pueden proyectar sobre un reducido número de páginas de gran tamaño en lugar de un gran número de páginas de tamaño más pequeño, mientras que las pilas de los diferentes hilos se pueden alojar utilizando tamaños de página relativamente pequeños. Sin embargo, la mayoría de sistemas operativos comerciales aún soportan únicamente un tamaño de página, independientemente de las capacidades del hardware sobre el que están ejecutando. El motivo de esto se debe a que el tamaño de página afecta a diferentes aspectos del sistema operativo; por tanto, un cambio a un modelo de diferentes tamaños de páginas representa una tarea significativamente compleja (véase [GANA98] para más detalle).

## SEGMENTACIÓN

**Las implicaciones en la memoria virtual.** La segmentación permite al programador ver la memoria como si se tratase de diferentes espacios de direcciones o segmentos. Los segmentos pueden ser de tamaños diferentes, en realidad de tamaño dinámico. Una referencia a la memoria consiste en un formato de dirección del tipo (número de segmento, desplazamiento).

Esta organización tiene un gran número de ventajas para el programador sobre los espacios de direcciones no segmentados:

1. Simplifica el tratamiento de estructuras de datos que pueden crecer. Si el programador no conoce a priori el tamaño que una estructura de datos en particular puede alcanzar es necesario hacer una estimación salvo que se utilicen tamaños de segmento dinámicos. Con la memoria virtual segmentada, a una estructura de datos se le puede asignar su propio segmento, y el sistema operativo expandirá o reducirá el segmento bajo demanda. Si un segmento que necesita expandirse se encuentre en la memoria principal y no hay suficiente tamaño, el sistema operativo podrá mover el segmento a un área de la memoria principal mayor, si se encuentra disponible, o enviarlo a *swap*. En este último caso el segmento al que se ha incrementado el tamaño volverá a la memoria principal en la siguiente oportunidad que tenga.
2. Permite programas que se modifican o recopilan de forma independiente, sin requerir que el conjunto completo de programas se re-enlacen y se vuelvan a cargar. De nuevo, esta posibilidad se puede articular por medio de la utilización de múltiples segmentos.
3. Da soporte a la compartición entre procesos. El programador puede situar un programa de utilidad o una tabla de datos que resulte útil en un segmento al que pueda hacerse referencia desde otros procesos.
4. Soporta los mecanismos de protección. Esto es debido a que un segmento puede definirse para contener un conjunto de programas o datos bien descritos, el programador o el administrador de sistemas puede asignar privilegios de acceso de una forma apropiada.

**Organización.** En la exposición de la segmentación sencilla, indicamos que cada proceso tiene su propia tabla de segmentos, y que cuando todos estos segmentos se han cargado en la memoria principal, la tabla de segmentos del proceso se crea y se carga también en la memoria principal. Cada entrada de la tabla de segmentos contiene la dirección de comienzo del correspondiente segmento en la

memoria principal, así como la longitud del mismo. El mismo mecanismo, una tabla segmentos, se necesita cuando se están tratando esquemas de memoria virtual basados en segmentación. De nuevo, lo habitual es que haya una única tabla de segmentos por cada uno de los procesos. En este caso sin embargo, las entradas en la tabla de segmentos son un poco más complejas (Figura 8.2b). Debido a que sólo algunos de los segmentos del proceso pueden encontrarse en la memoria principal, se necesita un bit en cada entrada de la tabla de segmentos para indicar si el correspondiente segmento se encuentra presente en la memoria principal o no. Si indica que el segmento está en memoria, la entrada también debe incluir la dirección de comienzo y la longitud del mismo.

Otro bit de control en la entrada de la tabla de segmentos es el bit de modificado, que indica si los contenidos del segmento correspondiente se han modificado desde que se cargó por última vez en la memoria principal. Si no hay ningún cambio, no es necesario escribir el segmento cuando se reemplaza de la memoria principal. También pueden darse otros bits de control. Por ejemplo, si la gestión de protección y compartición se gestiona a nivel de segmento, se necesitarán los bits correspondientes a estos fines.

El mecanismo básico para la lectura de una palabra de memoria implica la traducción de una dirección virtual, o lógica, consistente en un número de segmento y un desplazamiento, en una dirección física, usando la tabla de segmentos. Debido a que la tabla de segmentos es de tamaño variable, dependiendo del tamaño del proceso, no se puede suponer que se encuentra almacenada en un registro. En su lugar, debe encontrarse en la memoria principal para poder accederse. La Figura 8.12 sugiere una implementación hardware de este esquema (nótese la similitud con la Figura 8.3). Cuando un proceso en particular está en ejecución, un registro mantiene la dirección de comienzo de la tabla de segmentos para dicho proceso. El número de segmento de la dirección virtual se utiliza para indexar esta tabla y para buscar la dirección de la memoria principal donde comienza dicho segmento. Ésta es añadida a la parte de desplazamiento de la dirección virtual para producir la dirección real solicitada.

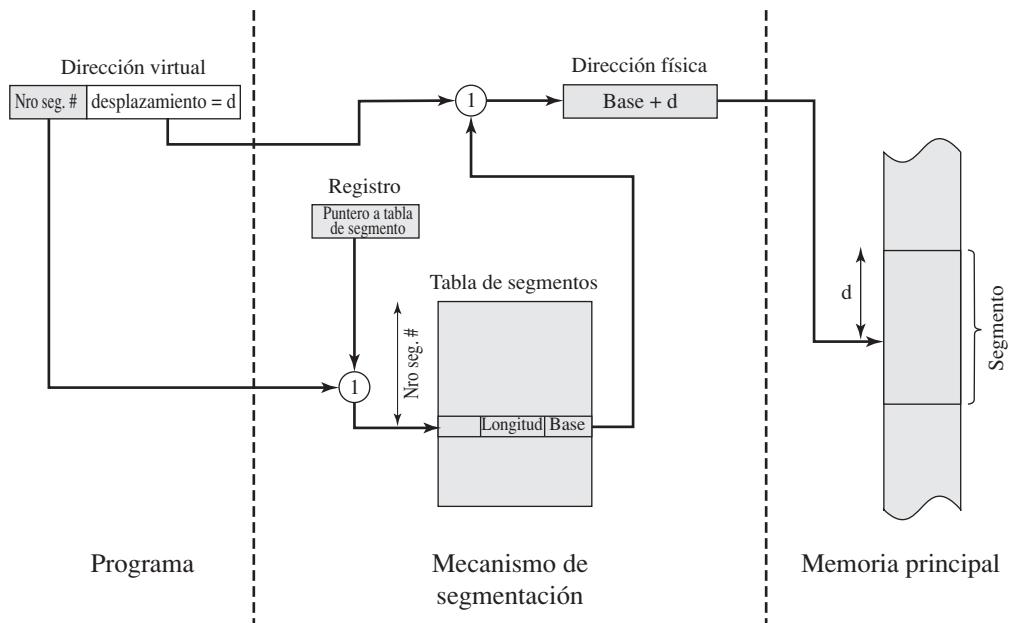


Figura 8.12. Traducción de direcciones en un sistema con segmentación.

## PAGINACIÓN Y SEGMENTACIÓN COMBINADAS

Paginación y segmentación, cada una tiene sus propias ventajas. La paginación es transparente al programador y elimina la fragmentación externa, y por tanto proporciona un uso eficiente de la memoria principal. Adicionalmente, debido a que los fragmentos que se mueven entre la memoria y el disco son de un tamaño igual y prefijado, es posible desarrollar algoritmos de gestión de la memoria más sofisticados que exploten el comportamiento de los programas, como veremos más adelante. La segmentación sí es visible al programador y tiene los beneficios que hemos visto anteriormente, incluyendo la posibilidad de manejar estructuras de datos que crecen, modularidad, y dar soporte a la compartición y a la protección. Para combinar las ventajas de ambos, algunos sistemas por medio del hardware del procesador y del soporte del sistema operativo son capaces de proporcionar ambos.

En un sistema combinado de paginación/segmentación, el espacio de direcciones del usuario se divide en un número de segmentos, a discreción del programador. Cada segmento es, por su parte, dividido en un número de páginas de tamaño fijo, que son del tamaño de los marcos de la memoria principal. Si un segmento tiene longitud inferior a una página, el segmento ocupará únicamente una página. Desde el punto de vista del programador, una dirección lógica sigue conteniendo un número de segmento y un desplazamiento dentro de dicho segmento. Desde el punto de vista del sistema, el desplazamiento dentro del segmento es visto como un número de página y un desplazamiento dentro de la página incluida en el segmento.

La Figura 8.13 sugiere la estructura para proporcionar soporte o la combinación de paginación y segmentación (nótese la similitud con la Figura 8.5). Asociada a cada proceso existe una tabla de segmentos y varias tablas de páginas, una por cada uno de los segmentos. Cuando un proceso está en ejecución, un registro mantiene la dirección de comienzo de la tabla de segmentos de dicho proceso. A partir de la dirección virtual, el procesador utiliza la parte correspondiente al número de segmento para indexar dentro de la tabla de segmentos del proceso para encontrar la tabla de pági-

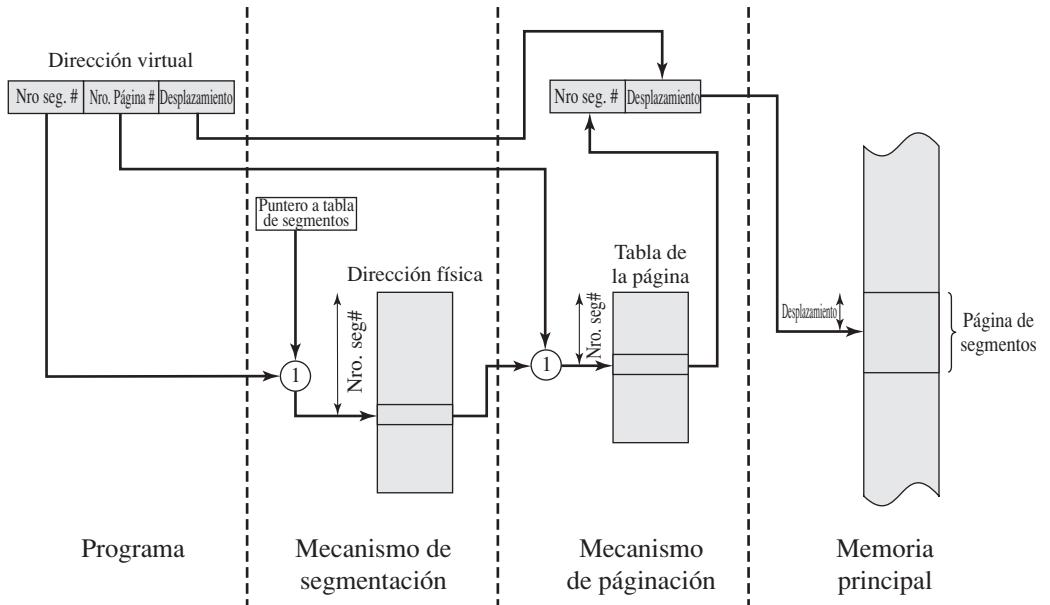


Figura 8.13. Traducción de direcciones en un sistema con segmentación/paginación.

nas de dicho segmento. Después, la parte correspondiente al número de página de la dirección virtual original se utiliza para indexar la tabla de páginas y buscar el correspondiente número de marco. Éste se combina con el desplazamiento correspondiente de la dirección virtual para generar la dirección real requerida.

En la Figura 8.2c se muestran los formatos de la entrada en la tabla de segmentos y de la entrada en la tabla de páginas. Como antes, la entrada en la tabla de segmentos contiene la longitud del segmento. También contiene el campo base, que ahora hace referencia a la tabla de páginas. Los bits de presente y modificado no se necesitan debido a que estos aspectos se gestionan a nivel de página. Otros bits de control sí pueden utilizarse, a efectos de compartición y protección. La entrada en la tabla de páginas es esencialmente la misma que para el sistema de paginación puro. El número de página se proyecta en su número de marco correspondiente si la página se encuentra presente en la memoria. El bit de modificado indica si la página necesita escribirse cuando se expulse del marco de página actual. Puede haber otros bits de control relacionados con la protección u otros aspectos de la gestión de la memoria.

## PROTECCIÓN Y COMPARTICIÓN

La segmentación proporciona una vía para la implementación de las políticas de protección y compartición. Debido a que cada entrada en la tabla de segmentos incluye la longitud así como la dirección base, un programa no puede, de forma descontrolada, acceder a una posición de memoria principal más allá de los límites del segmento. Para conseguir compartición, es posible que un segmento se encuentre referenciado desde las tablas de segmentos de más de un proceso. Los mecanismos están, por supuesto, disponibles en los sistemas de paginación. Sin embargo, en este caso la estructura de páginas de un programa y los datos no son visibles para el programador, haciendo que la especificación de la protección y los requisitos de compartición sean menos cómodos. La Figura 8.14 ilustra los tipos de relaciones de protección que se pueden definir en dicho sistema.

También es posible proporcionar mecanismos más sofisticados. Un esquema habitual es utilizar la estructura de protección en anillo, del tipo que indicamos en el Capítulo 3 (Problema 3.7). En este esquema, los anillos con números bajos, o interiores, disfrutan de mayores privilegios que los anillos con numeraciones más altas, o exteriores. Normalmente, el anillo 0 se reserva para funciones del núcleo del sistema operativo, con las aplicaciones en niveles superiores. Algunas utilidades o servicios de sistema operativo pueden ocupar un anillo intermedio. Los principios básicos de los sistemas en anillo son los siguientes:

- Un programa pueda acceder sólo a los datos residentes en el mismo anillo o en anillos con menos privilegios.
- Un programa puede invocar servicios residentes en el mismo anillo o anillos con más privilegios.

## 8.2. SOFTWARE DEL SISTEMA OPERATIVO

El diseño de la parte de la gestión de la memoria del sistema operativo depende de tres opciones fundamentales a elegir:

- Si el sistema usa o no técnicas de memoria virtual.
- El uso de paginación o segmentación o ambas.
- Los algoritmos utilizados para los diferentes aspectos de la gestión de la memoria.

### 2.4.1 Paginación

- El espacio de direcciones físicas de un proceso puede ser no contiguo
- La memoria física se divide en bloques de tamaño fijo, denominados *marcos de página*. El tamaño es potencia de dos, de 512 B a 8 KB
- El espacio lógico de un proceso se divide conceptualmente en bloques del mismo tamaño, denominados *páginas*
- Los marcos de página contendrán páginas de los procesos

### 2.4.1 Paginación

Las **direcciones lógicas**, que son las que genera la CPU se dividen en **número de página (p)** y **desplazamiento (d)**



Las **direcciones físicas** se dividen en **número de marco (m, dirección base del marco donde está almacenada la página)** y **desplazamiento (d)**



### 2.4.1 Paginación

Cuando la CPU genere una dirección lógica será necesario traducirla a la dirección física correspondiente, la *tabla de páginas* mantiene información necesaria para realizar dicha traducción. *Existe una tabla de páginas por proceso*

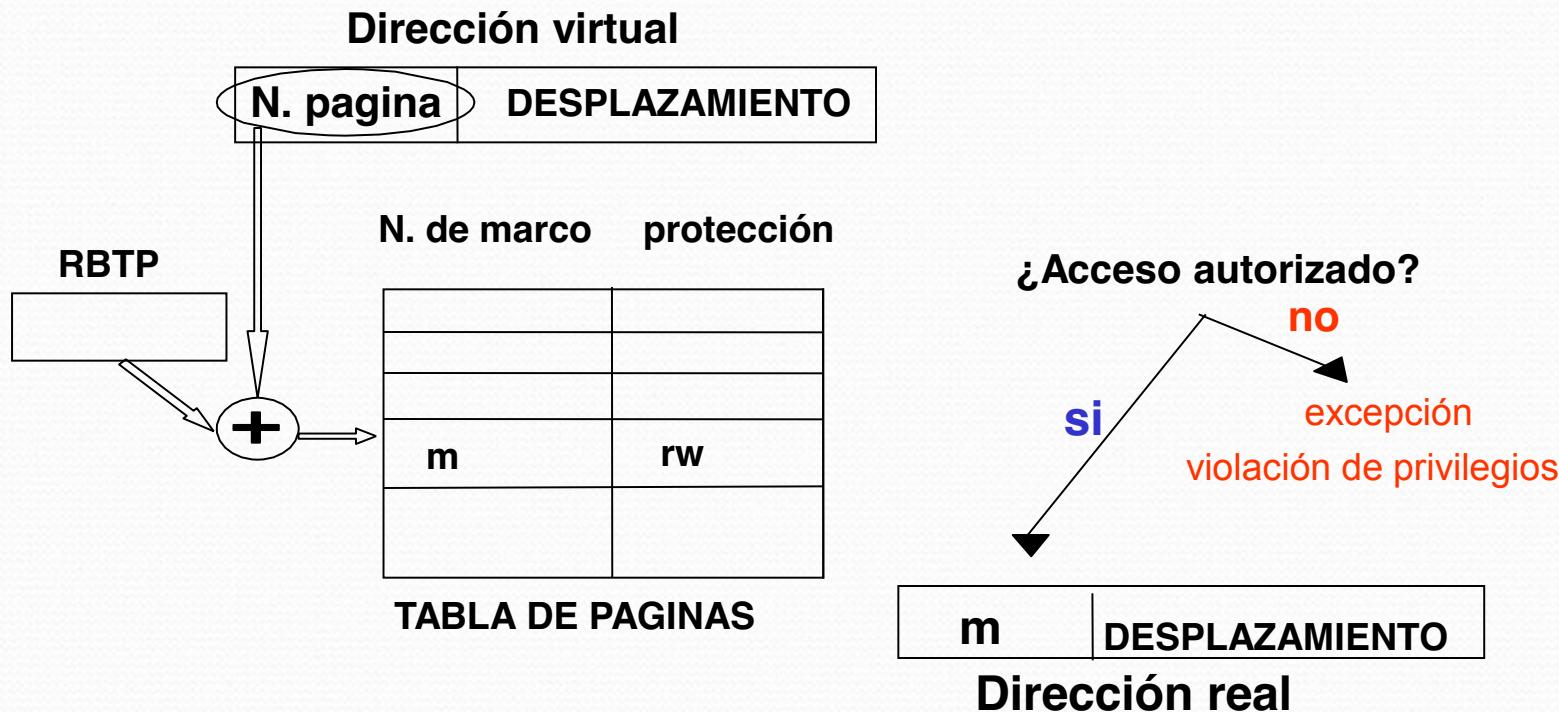
*Tabla de marcos de página*, usada por el S.O. y contiene información sobre cada marco de página

### Contenido de la tabla de páginas

Una entrada por cada página del proceso:

- Número de marco (dirección base del marco) en el que está almacenada la página si está en MP
- Modo de acceso autorizado a la página (bits de protección)

### Esquema de traducción



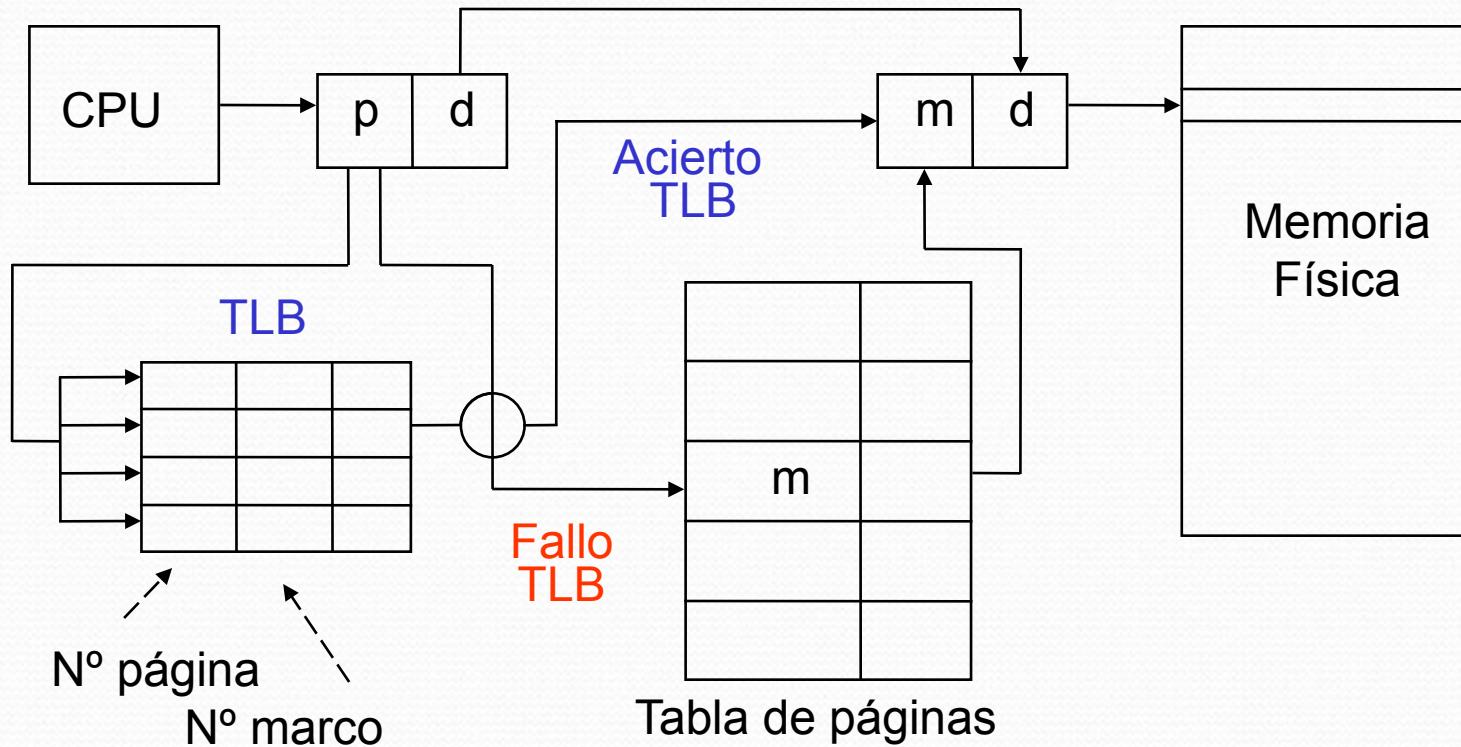
### Implementación de la Tabla de Páginas

- La tabla de páginas se mantiene en memoria principal
- El *registro base de la tabla de páginas (RBTP)* apunta a la tabla de páginas (suele almacenarse en el PCB del proceso)
- En este esquema cada acceso a una instrucción o dato requiere **dos accesos a memoria**, uno a la tabla de páginas y otro a memoria.

### Búfer de Traducción Adelantada (TLB)

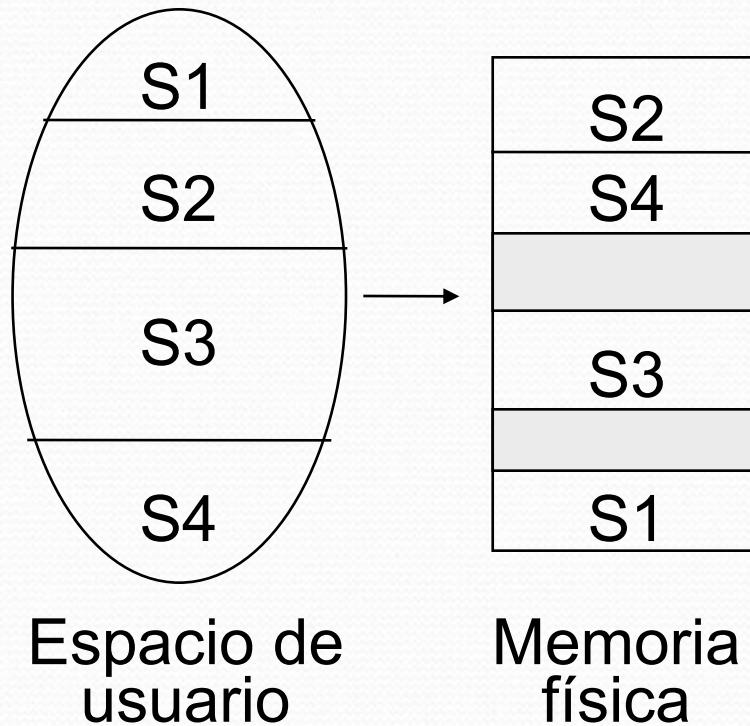
- El problema de los dos accesos a memoria se resuelve con una caché hardware de consulta rápida denominada *búfer de traducción adelantada* o **TLB** (*Translation Look-aside Buffer*)
- El TLB se implementa como un conjunto de *registros asociativos* que permiten una búsqueda en paralelo
- De esta forma, para traducir una dirección:
  - 1 Si existe ya en el registro asociativo, obtenemos el marco
  - 2 Si no, la buscamos en la tabla de páginas y se actualiza el TLB con esta nueva entrada

## Esquema de traducción con TLB



### 2.4.2 Segmentación

Esquema de organización de memoria que soporta mejor la visión de memoria del usuario: un programa es una colección de unidades lógicas -segmentos-, p. ej. procedimientos, funciones, pila, tabla de símbolos, matrices, etc.



### Tabla de Segmentos

Una dirección lógica es una tupla:

<número\_de\_segmento, desplazamiento>

La *Tabla de Segmentos* aplica direcciones bidimensionales definidas por el usuario en direcciones físicas de una dimensión. Cada entrada de la tabla tiene los siguientes elementos (aparte de presencia, modificación y protección):

- » **base** - dirección física donde reside el inicio del segmento en memoria.
- » **tamaño** - longitud del segmento.

### Implementación de la Tabla de Segmentos

- La tabla de segmentos se mantiene en memoria principal.
- El *Registro Base de la Tabla de Segmentos* (*RBTS*) apunta a la tabla de segmentos (suele almacenarse en el PCB del proceso)
- El *Registro Longitud de la Tabla de Segmentos* (*STLR*) indica el número de segmentos del proceso; el nº de segmento  $s$ , generado en una dirección lógica, es legal si  $s < STLR$  (suele almacenarse en el PCB del proceso)

### Esquema de traducción

