CAPÍTULO 9

Memoria virtual

En el Capítulo 8, hemos expuesto diversas estrategias de gestión de memoria utilizadas en los sistemas informáticos. Todas estas estrategias tienen el mismo objetivo: mantener numerosos procesos en memoria simultáneamente, con el fin de permitir la multiprogramación. Sin embargo, esas estrategias tienden a requerir que el proceso completo esté en memoria para poder ejecutarse.

La técnica de memoria virtual es un mecanismo que permite la ejecución de procesos que no se encuentren completamente en la memoria. Una de las principales ventajas de este esquema es que los programas pueden tener un tamaño mayor que la propia memoria física. Además, la memoria virtual abstrae la memoria principal, transformándola conceptualmente en una matriz uniforme y extremadamente grande de posiciones de almacenamiento, separando así la memoria lógica (tal como la ve el usuario) de la memoria física. Esta técnica libera a los programadores de las preocupaciones relativas a las limitaciones del espacio de almacenamiento de memoria. La memoria virtual también permite que los procesos compartan los archivos e implementen mecanismos de memoria compartida. Además, proporciona un mecanismo eficiente para la creación de procesos. Sin embargo, la memoria virtual no resulta fácil de implementar y puede reducir sustancialmente el rendimiento del sistema si se la utiliza sin el debido cuidado. En este capítulo, vamos a analizar los mecanismos de memoria virtual basados en paginación bajo demanda y examinaremos la complejidad y el coste asociados.

OBJETIVOS DEL CAPÍTULO

- Describir las ventajas de un sistema de memoria virtual.
- Explicar los conceptos de paginación bajo demanda, algoritmos de sustitución de páginas y asignación de marcos de páginas.
- Analizar los principios en que se basa el modelo de conjunto de trabajo.

9.1 Fundamentos

Los algoritmos de gestión de memoria esbozados en el Capítulo 8 resultan necesarios debido a un requerimiento básico: las instrucciones que se estén ejecutando deben estar en la memoria física. El primer enfoque para tratar de satisfacer este requisito consiste en colocar el espacio completo de direcciones lógicas dentro de la memoria física. Los mecanismos de carga dinámica pueden ayudar a aliviar esta restricción, pero requieren, por lo general, que el programador tome precauciones especiales y lleve a cabo un trabajo adicional.

El requisito de que las instrucciones deban encontrarse en la memoria física para poderlas ejecutar parece, a la vez, tanto necesario como razonable; pero también es un requisito poco deseable, ya que limita el tamaño de los programas, de manera que éstos no pueden exceder del tamaño

de la propia memoria física. De hecho, un examen de los programas reales nos muestra que, en muchos casos, no es necesario tener el programa completo para poderlo ejecutar. Por ejemplo, considere lo siguiente:

- Los programas incluyen a menudo código para tratar las condiciones de error poco usuales. Puesto que estos errores raramente ocurren en la práctica (si es que ocurren alguna vez), este código no se ejecuta prácticamente nunca.
- A las matrices, a las listas y a las tablas se las suele asignar más memoria de la que realmente necesitan. Por ejemplo, puede declararse una matriz como compuesta de 100 por 100 elementos, aunque raramente vaya a tener un tamaño superior a 10 por 10 elementos. Una tabla de símbolos de ensamblador puede tener espacio para 3000 símbolos, aunque un programa típico suela tener menos de 200 símbolos.
- Puede que ciertas opciones y características de un programa se utilicen raramente. Por ejemplo, las rutinas que ejecutan funciones avanzadas de cálculo dentro de determinados programas de hoja de cálculo no suelen ser utilizadas por muchos usuarios.

Incluso en aquellos casos en que se necesite el programa completo, puede suceder que no todo el programa sea necesario al mismo tiempo.

La posibilidad de ejecutar un programa que sólo se encontrara parcialmente en la memoria proporcionaría muchas ventajas:

- Los programas ya no estarían restringidos por la cantidad de memoria física disponible. Los usuarios podrían escribir programas para un espacio de direcciones *virtual* extremadamente grande, simplificándose así la tarea de programación.
- Puesto que cada programa de usuario podría ocupar menos memoria física, se podrían ejecutar más programas al mismo tiempo, con el correspondiente incremento en la tasa de utilización del procesador y en la tasa de procesamiento, y sin incrementar el tiempo de respuesta ni el tiempo de ejecución.
- Se necesitarían menos operaciones de E/S para cargar o intercambiar cada programa de usuario con el fin de almacenarlo en memoria, de manera que cada programa de usuario se ejecutaría más rápido.

Por tanto, ejecutar programas que no se encuentren completamente en memoria proporciona ventajas tanto para el sistema como para el usuario.

La **memoria virtual** incluye la separación de la memoria lógica, tal como la percibe el usuario, con respecto a la memoria física. Esta separación permite proporcionar a los programadores una memoria virtual extremadamente grande, cuando sólo está disponible una memoria física de menor tamaño (Figura 9.1).

La memoria virtual facilita enormemente la tarea de programación, porque el programador ya no tiene por qué preocuparse de la cantidad de memoria física disponible; en lugar de ello, puede concentrarse en el problema que deba resolver mediante su programa.

El **espacio de direcciones virtuales** de un proceso hace referencia a la forma lógica (o virtual) de almacenar un proceso en la memoria. Típicamente, esta forma consiste en que el proceso comienza en una cierta dirección lógica (por ejemplo, la dirección 0) y está almacenado de forma contigua en la memoria, como se muestra en la Figura 9.2. Recuerde del Capítulo 8, sin embargo, que de hecho la memoria física puede estar organizada en marcos de página y que los marcos de página física asignados a un proceso pueden no ser contiguos. Es responsabilidad de la unidad de gestión de memoria (MMU, memory-management unit) establecer la correspondencia entre las páginas lógicas y los marcos de página física de la memoria.

Observe que, en la Figura 9.2, dejamos que el cúmulo crezca hacia arriba en la memoria, ya que se utiliza para la asignación dinámica de memoria. De forma similar, permitimos que la pila crezca hacia abajo en la memoria con las sucesivas llamadas a función. El gran espacio vacío (o hueco) entre el cúmulo y la pila forma parte del espacio de direcciones virtual, pero sólo consumirá páginas físicas reales cuando crezcan el cúmulo o la pila. Los espacios de direcciones virtuales que

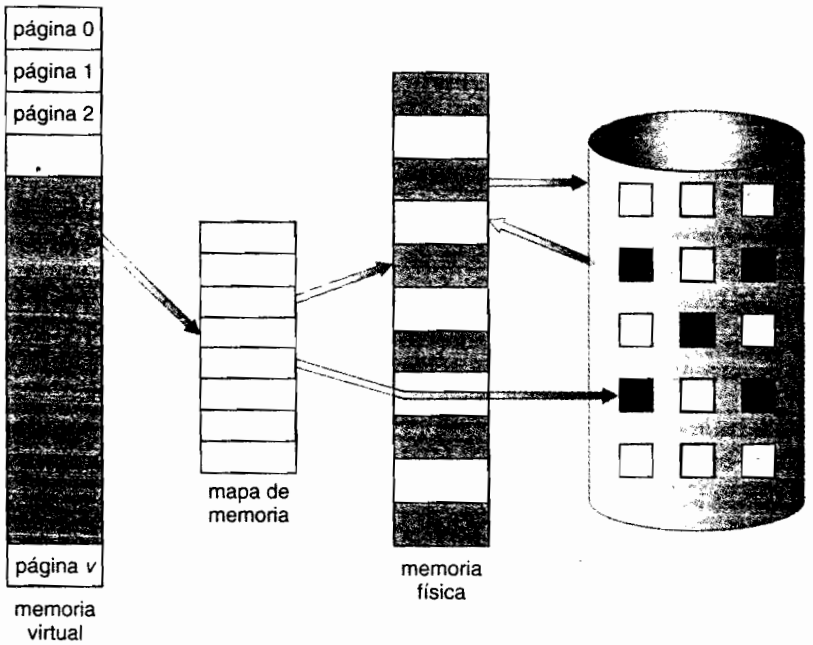

Figura 9.1 Diagrama que muestra una memoria virtual de tamaño mayor que la memoria física.

incluyen este tipo de huecos se denominan espacios de direcciones **dispersos**. Resulta ventajoso utilizar un espacio de direcciones disperso, porque los huecos pueden rellenarse a medida que crecen los segmentos de pila o de cúmulo, o también si queremos montar dinámicamente una serie de bibliotecas (o posiblemente otros objetos compartidos) durante la ejecución del programa.

Además de separar la memoria lógica de la memoria física, la memoria virtual también permite que dos o más procesos compartan los archivos y la memoria mediante mecanismos de compartición de páginas (Sección 8.4.4). Esto proporciona las siguientes ventajas:

- Las bibliotecas del sistema pueden ser compartidas por numerosos procesos, mapeando el objeto compartido sobre un espacio de direcciones virtual. Aunque cada proceso considera

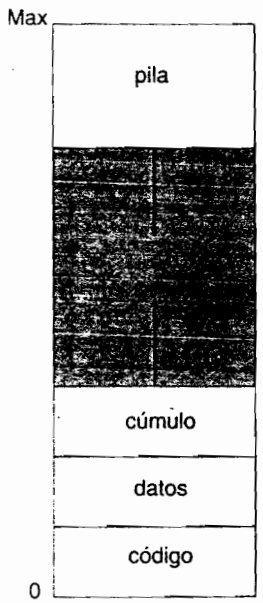

Figura 9.2 Espacio de direcciones virtual.

que las bibliotecas compartidas forman parte de su propio espacio de direcciones vi... las páginas reales de memoria física en las que residen las bibliotecas estarán compar... por todos los procesos (Figura 9.3). Normalmente, las bibliotecas se mapean en mod... sólo lectura dentro del espacio de cada proceso con el cual se monten.

- De forma similar, la memoria virtual permite a los procesos compartir memoria. Recu... del Capítulo 3 que dos o más procesos pueden comunicarse utilizando memoria comp... da. La memoria virtual permite que un proceso cree una región de memoria que lu... pueda compartir con otro proceso. Los procesos que compartan esta región la considera... parte de su espacio de direcciones virtual, aunque en realidad las páginas físicas reale... la memoria estarán compartidas entre los distintos procesos, de forma similar a como... ilustra en la Figura 9.3.

- La memoria virtual puede permitir que se compartan páginas durante la creación de proc... sos mediante la llamada al sistema `fork()`, acelerando así la tarea de creación de p... cesos.

Exploraremos estas y otras ventajas de la memoria virtual con más detalle posteriormente en... capítulo; pero antes de ello, veamos cómo puede implementarse la memoria virtual median... mecanismos de paginación bajo demanda.

9.2 Paginación bajo demanda

Considere cómo podría cargarse un programa ejecutable desde el disco a la memoria. Una opción consiste en cargar el programa completo en memoria física en el momento de ejecutar el progra... ma. Sin embargo, esta técnica presenta el problema de que puede que no *necesitemos* inicialmente todo el programa en la memoria. Considere un programa que comience con una lista de acciones disponibles, de entre las cuales el usuario debe seleccionar una. Cargar el programa completo en memoria hace que se cargue el código ejecutable de *todas* las opciones, independientemente de si el usuario selecciona o no una determinada opción. Una estrategia alternativa consiste en cargar inicialmente las páginas únicamente cuando sean necesarias. Esta técnica se denomina **paginación bajo demanda** y se utiliza comúnmente en los sistemas de memoria virtual. Con la memoria virtual basada en paginación bajo demanda, sólo se cargan las páginas cuando así se solicita durante la ejecución del programa; de este modo, las páginas a las que nunca se acceda no llegarán a cargarse en la memoria física.

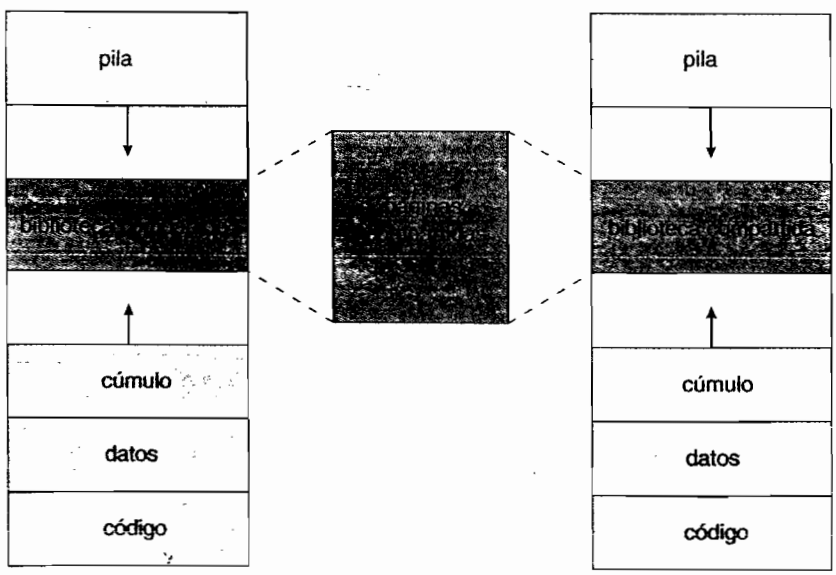

Figura 9.3 Biblioteca compartida mediante memoria virtual.

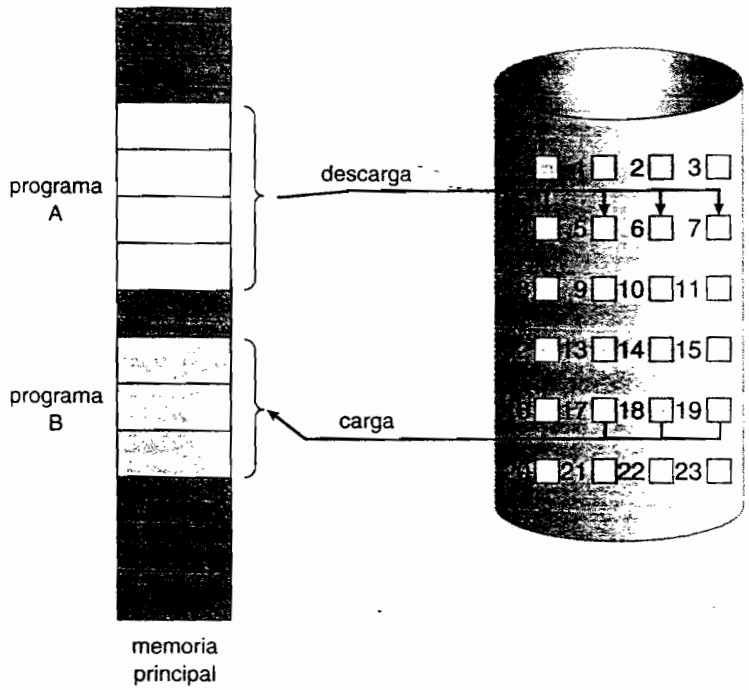

Figura 9.4 Transferencia de una memoria paginada a un espacio contiguo de disco.

Un sistema de paginación bajo demanda es similar a un sistema de paginación con intercambio (Figura 9.4) en el que los procesos residen en memoria secundaria (usualmente en disco). Cuando queremos ejecutar un proceso, realizamos un intercambio para cargarlo en memoria. Sin embargo, en lugar de intercambiar el proceso completo con la memoria, lo que hacemos en este caso es utilizar un **intercambiador perezoso**. El intercambiador perezoso jamás intercambia una página con la memoria a menos que esa página vaya a ser necesaria. Puesto que ahora estamos contemplando los procesos como secuencias de páginas, en lugar de como un único espacio de direcciones contiguas de gran tamaño, la utilización del término *intercambiador* es técnicamente incorrecta. Un intercambiador manipula procesos completos, mientras que un **paginador** sólo se ocupa de las páginas individuales de un proceso. Por tanto, utilizaremos el término *paginador* en lugar de *intercambiador*, cuando hablemos de la paginación bajo demanda.

9.2.1 Conceptos básicos

Cuando hay que cargar un proceso, el paginador realiza una estimación de qué páginas serán utilizadas antes de descargar de nuevo el proceso. En lugar de cargar el proceso completo, el paginador sólo carga en la memoria las páginas necesarias; de este modo, evita cargar en la memoria las páginas que no vayan a ser utilizadas, reduciendo así el tiempo de carga y la cantidad de memoria física necesaria.

Con este esquema, necesitamos algún tipo de soporte hardware para distinguir entre las páginas que se encuentran en memoria y las páginas que residen en el disco. Podemos usar para este propósito el esquema descrito en la Sección 8.5, basado en un bit válido-inválido. Sin embargo, esta vez, cuando se configura este bit como "válido", la página asociada será legal y además se encontrará en memoria. Si el bit se configura como "inválido", querrá decir que o bien la página no es válida (es decir, no se encuentra en el espacio lógico de direcciones del proceso) o es válida pero está actualmente en el disco. La entrada de la tabla de páginas correspondiente a una página que se cargue en memoria se configurará de la forma usual, pero la entrada de la tabla de páginas correspondiente a una página que no se encuentre actualmente en la memoria se marcará simplemente como inválida o contendrá la dirección de la página dentro del disco. Esta situación se ilustra en la Figura 9.5.

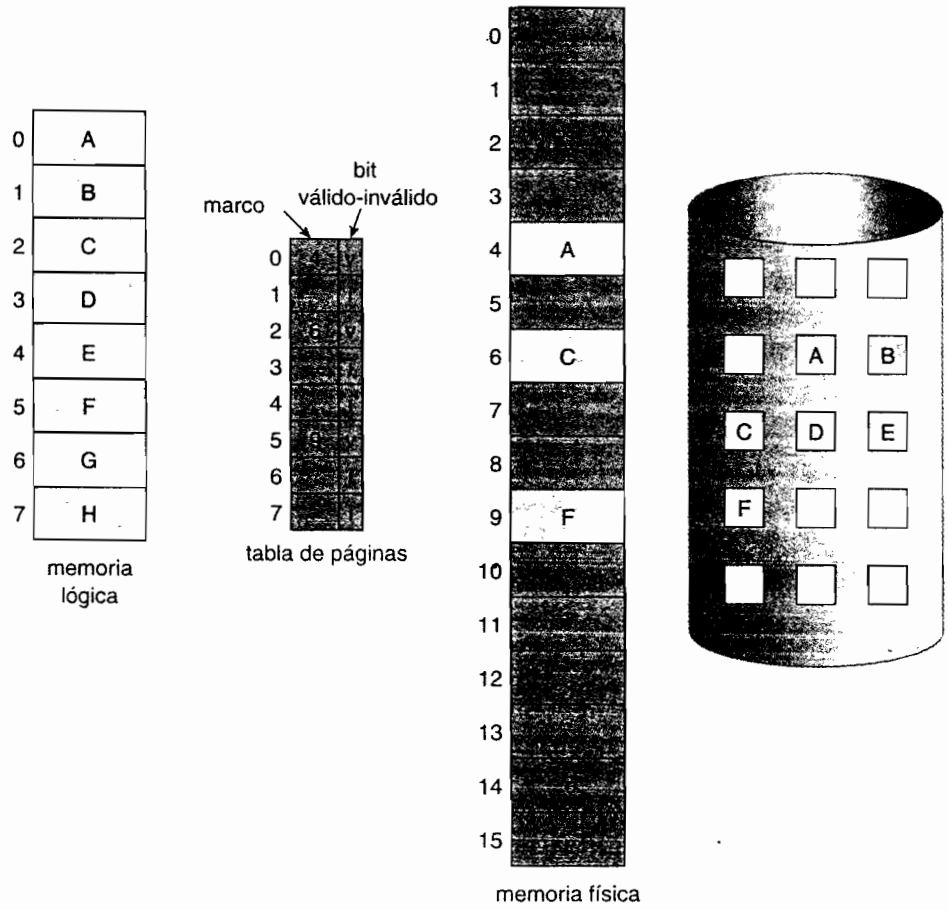

Figura 9.5 Tabla de páginas cuando algunas páginas no se encuentran en memoria principal.

Observe que marcar una página como inválida no tendrá ningún efecto si el proceso no intenta nunca acceder a dicha página. Por tanto, si nuestra estimación inicial es correcta y cargamos en la memoria todas las páginas que sean verdaderamente necesarias (y sólo esas páginas), el proceso se ejecutará exactamente igual que si hubiéramos cargado en memoria todas las páginas. Mientras que el proceso se ejecute y acceda a páginas que sean **residentes en memoria**, la ejecución se llevará a cabo normalmente.

Pero ¿qué sucede si el proceso trata de acceder a una página que no haya sido cargada en memoria? El acceso a una página marcada como inválida provoca una **interrupción de fallo de página**. El hardware de paginación, al traducir la dirección mediante la tabla de páginas, detectará que el bit de página inválida está activado, provocando una interrupción dirigida al sistema operativo. Esta interrupción se produce como resultado de que el sistema operativo no ha cargado anteriormente en memoria la página deseada. El procedimiento para tratar este fallo de página es muy sencillo (Figura 9.6):

1. Comprobamos una tabla interna (que usualmente se mantiene con el bloque del control del proceso) correspondiente a este proceso, para determinar si la referencia era un acceso de memoria válido o inválido.

2. Si la referencia era inválida, terminamos el proceso. Si era válida pero esa página todavía no ha sido cargada, la cargamos en memoria.

3. Buscamos un marco libre (por ejemplo, tomando uno de la lista de marcos libres).

4. Ordenamos una operación de disco para leer la página deseada en el marco recién asignado.

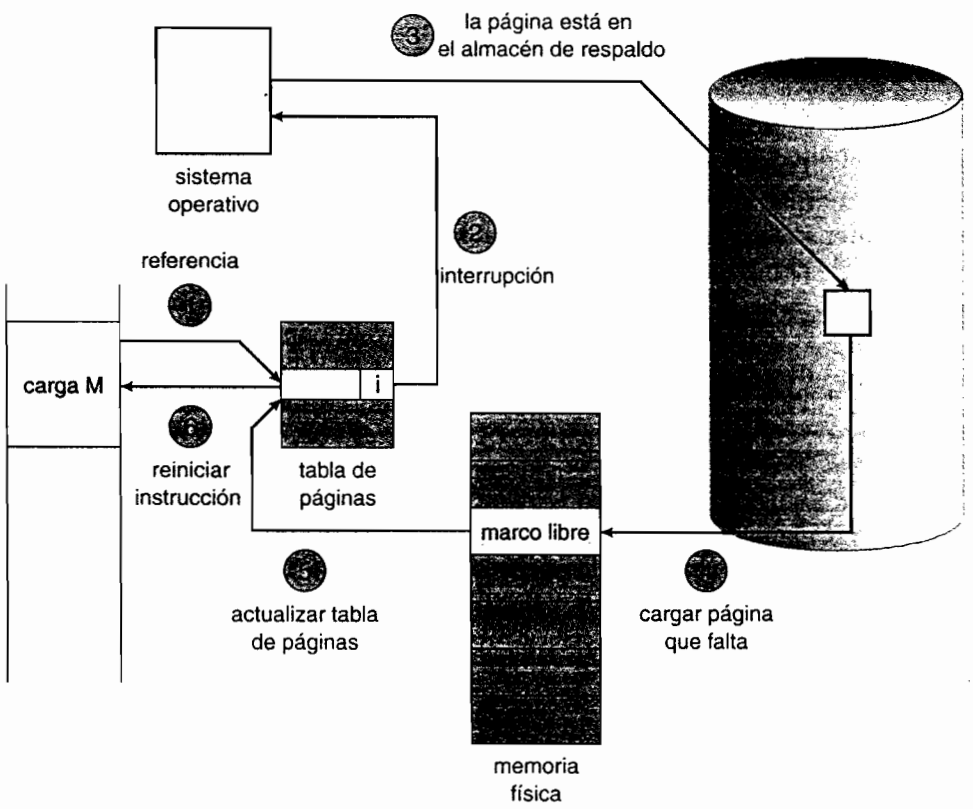

Figura 9.6 Pasos para tratar un fallo de página.

5. Una vez completada la lectura de disco, modificamos la tabla interna que se mantiene con los datos del proceso y la tabla de páginas para indicar que dicha página se encuentra ahora en memoria.
6. Reiniciamos la instrucción que fue interrumpida. El proceso podrá ahora acceder a la página como si siempre hubiera estado en memoria.

En el caso extremo, podríamos empezar a ejecutar un proceso *sin ninguna* página en memoria. Cuando el sistema operativo hiciera que el puntero de instrucciones apuntara a la primera instrucción del proceso, que se encontraría en una página no residente en memoria, se produciría inmediatamente un fallo de página. Después de cargar dicha página en memoria, el proceso continuaría ejecutándose, generando los fallos de página necesarios hasta que todas las páginas requeridas se encontraran en memoria. A partir de ahí, podría ejecutarse sin ningún fallo de página adicional. Este esquema sería una **paginación bajo demanda pura**: nunca cargar una página en memoria hasta que sea requerida.

En teoría, algunos programas podrían acceder a varias nuevas páginas de memoria con cada ejecución de una instrucción (una página para la instrucción y muchas para los datos), posiblemente generando múltiples fallos de página por cada instrucción. Esta situación provocaría una degradación inaceptable en el rendimiento del sistema. Afortunadamente, el análisis de la ejecución de los procesos muestra que este comportamiento es bastante improbable. Los programas tienden a tener lo que se denomina **localidad de referencia**, que se describe en la Sección 9.6.1, por lo que las prestaciones que pueden obtenerse con el mecanismo de paginación bajo demanda son razonables.

El hardware necesario para soportar la paginación bajo demanda es el mismo que para los mecanismos de paginación e intercambio:

- Una **tabla de páginas**. Esta tabla permite marcar una entrada como inválida mediante un bit válido-inválido o mediante un valor especial de una serie de bits de protección.

- **Memoria secundaria.** Esta memoria almacena aquellas páginas que no están presentes en memoria principal. La memoria secundaria es usualmente un disco de alta velocidad. Se le conoce como dispositivo de intercambio y la sección del disco utilizada para este propósito se llama **espacio de intercambio**. La cuestión de la asignación del espacio de intercambio se analiza en el Capítulo 12.

Un requisito fundamental para la paginación bajo demanda es la necesidad de poder reiniciar cualquier instrucción después de un fallo de página. Puesto que guardamos el estado (registros, código de condición, contador de instrucciones) del proceso interrumpido en el momento de producirse el fallo de página, debemos poder reiniciar el proceso en *exactamente* el mismo lugar y con el mismo estado, salvo porque ahora la página deseada se encontrará en memoria y será accesible. En la mayoría de los casos, este requisito resulta fácil de satisfacer. Los fallos de página pueden producirse para cualquier referencia a memoria. Si el fallo de página se produce al extraer una instrucción, podemos reiniciar las operaciones volviendo a extraer dicha instrucción. Si el fallo de página se produce mientras estamos leyendo un operando, deberemos extraer y decodificar de nuevo la instrucción y luego volver a leer el operando.

Como ejemplo más desfavorable, considere una instrucción con tres direcciones, como por ejemplo una que realizara la suma del contenido de A con B, colocando el resultado en C. Estos serían los pasos para ejecutar esta instrucción:

1. Extraer y decodificar la instrucción (ADD).
2. Extraer A.
3. Extraer B.
4. Sumar A y B.
5. Almacenar la suma en C.

Si el fallo de página se produce cuando estamos tratando de almacenar el resultado en C (porque C se encuentra en una página que no está actualmente en memoria), tendremos que obtener la página deseada, cargarla, corregir la tabla de páginas y reiniciar la instrucción. Ese reinicio requerirá volver a extraer la instrucción, decodificarla de nuevo, extraer otra vez los dos operandos y luego realizar de nuevo la suma. Sin embargo, el trabajo que hay que repetir no es excesivo (menos de una instrucción completa) y esa repetición sólo es necesaria cuando se produce un fallo de página.

La principal dificultad surge cuando una instrucción puede modificar varias ubicaciones diferentes. Por ejemplo, considere la instrucción MVC (mover carácter) de IBM System 360/370, que puede mover hasta 256 bytes de una ubicación a otra, pudiendo estar solapados los dos rangos de direcciones. Si alguno de los bloques (el de origen o el de destino) atraviesa una frontera de página, podría producirse un fallo de página después de que esa operación de desplazamiento se hubiera completado de manera parcial. Además, si los bloques de origen y de destino se solapan, el bloque de origen puede haber sido modificado, en cuyo caso no podemos simplemente reiniciar la instrucción.

Este problema puede resolverse de dos formas distintas. En una de las soluciones, el microcódigo calcula y trata de acceder a los dos extremos de ambos bloques. Si va a producirse un fallo de página, se producirá en este punto, antes de que se haya modificado nada. Entonces, podremos realizar el desplazamiento (después de cargar páginas según sea necesario) porque sabemos que ya no puede volver a producirse ningún fallo de página, ya que todas las páginas relevantes están en la memoria. La otra solución utiliza registros temporales para almacenar los valores de las ubicaciones sobreescritas. Si se produce un fallo de página, vuelven a escribirse en memoria los antiguos valores antes de que se produzca la interrupción. Esta acción restaura la memoria al estado que tenía antes de iniciar la instrucción, así que la instrucción puede repetirse.

Éste no es, en modo alguno, el único problema relativo a la arquitectura que surge como resultado de añadir el mecanismo de paginación a una arquitectura existente para permitir la paginación bajo demanda. Aunque este ejemplo sí que ilustra muy bien algunas de las dificultades

implicadas. El mecanismo de paginación se añade entre la CPU y la memoria en un sistema informático; debe ser completamente transparente para el proceso de usuario. Debido a ello, todos tendemos a pensar que se pueden añadir mecanismos de paginación a cualquier sistema. Pero, aunque esta suposición es cierta para los entornos de paginación que no son bajo demanda, en los que los fallos de página representan errores fatales, no es cierta cuando un fallo de página sólo significa que es necesario cargar en memoria una página adicional y luego reiniciar el proceso.

9.2.2 Rendimiento de la paginación bajo demanda

La paginación bajo demanda puede afectar significativamente al rendimiento de un sistema informático. Para ver por qué, vamos a calcular el **tiempo de acceso efectivo** a una memoria con paginación bajo demanda. Para la mayoría de los sistemas informáticos, el tiempo de acceso a memoria, que designaremos *ma*, va de 10 a 200 ns. Mientras no tengamos fallos de página, el tiempo de acceso efectivo será igual al tiempo de acceso a memoria. Sin embargo, si se produce un fallo de página, deberemos primero leer la página relevante desde el disco y luego acceder a la página deseada.

Sea p la probabilidad de que se produzca un fallo de página ($0 \leq p \leq 1$). Cabe esperar que p esté próxima a cero, es decir, que sólo vayan a producirse unos cuantos fallos de página. El **tiempo de acceso efectivo** será entonces

$$\text{tiempo de acceso efectivo} = (1 - p) \times ma + p \times \text{tiempo de fallo de página.}$$

Para calcular el tiempo de acceso efectivo, debemos conocer cuánto tiempo se requiere para dar servicio a un fallo de página. Cada fallo de página hace que se produzca la siguiente secuencia:

1. Interrupción al sistema operativo.
2. Guardar los registros de usuario y el estado del proceso.
3. Determinar que la interrupción se debe a un fallo de página.
4. Comprobar que la referencia de página era legal y determinar la ubicación de la página en el disco.
5. Ordenar una lectura desde el disco para cargar la página en un marco libre:
    a. Esperar en la cola correspondiente a este dispositivo hasta que se dé servicio a la solicitud de lectura.
    b. Esperar el tiempo de búsqueda y/o latencia del dispositivo.
    c. Comenzar la transferencia de la página hacia un marco libre.
6. Mientras estamos esperando, asignar la CPU a algún otro usuario (planificación de la CPU, que será opcional).
7. Recibir una interrupción del subsistema de E/S de disco (E/S completada).
8. Guardar los registros y el estado del proceso para el otro usuario (si se ejecuta el paso 6).
9. Determinar que la interrupción corresponde al disco.
10. Corregir la tabla de páginas y otras tablas para mostrar que la página deseada se encuentra ahora en memoria.
11. Esperar a que se vuelva a asignar la CPU a este proceso.
12. Restaurar los registros de usuario, el estado del proceso y la nueva tabla de páginas y reanudar la ejecución de la instrucción interrumpida.

No todos estos pasos son necesarios en todos los casos. Por ejemplo, estamos suponiendo que, en el paso 6, la CPU se asigna a otro proceso mientras que tiene lugar la E/S. Esta operación permite que los mecanismos de multiprogramación mantengan una alta tasa de utilización de la CPU,

pero requiere un tiempo adicional para reanudar la rutina de servicio del fallo de página después de completarse la transferencia de E/S.

En cualquier caso, nos enfrentamos con tres componentes principales del tiempo de servicio de fallo de página:

1. Servir la interrupción de fallo de página.
2. Leer la página.
3. Reiniciar el proceso.

Las tareas primera y tercera pueden reducirse, codificando cuidadosamente el software, a unos cuantos cientos de instrucciones. Estas tareas pueden requerir entre 1 y 100 microsegundos cada una. Sin embargo, el tiempo de conmutación de página estará cerca, probablemente, de los 8 milisegundos. Un disco duro típico tiene una latencia media de 3 milisegundos, un tiempo de búsqueda de 5 milisegundos y un tiempo de transferencia de 0,05 milisegundos. Por tanto, el tiempo total de paginación es de unos 8 milisegundos, incluyendo los retardos hardware y software. Recuerde también que sólo estamos examinando el tiempo de servicio del dispositivo. Si hay una cola de procesos esperando a que el dispositivo les de servicio (otros procesos que hayan causado fallos de páginas), tendremos que añadir el tiempo de espera en cola para el dispositivo, ya que habrá que esperar a que el dispositivo de paginación esté libre para dar servicio a nuestra solicitud, incrementando todavía más el tiempo necesario para el intercambio.

Si tomamos un tiempo medio de servicio de fallo de página de 8 milisegundos y un tiempo de acceso a memoria de 200 nanosegundos, el tiempo efectivo de acceso en nanosegundos será:

$$\begin{aligned}\text{tiempo efectivo de acceso} &= (1 - p) \times (200) + p\ (8 \text{ milisegundos}) \\ &= (1 - p) \times 200 + p \times 8.000.000 \\ &= 200 + 7.999.800 \times p.\end{aligned}$$

Podemos ver, entonces, que el tiempo de acceso efectivo es directamente proporcional a la **tasa de fallos de página**. Si sólo un acceso de cada 1000 provoca un fallo de página, el tiempo efectivo de acceso será de 8,2 microsegundos. La computadora se ralentizará, por tanto, según el factor de 40 debido a la paginación bajo demanda. Si queremos que la degradación del rendimiento sea inferior al 10 por ciento, necesitaremos

$$\begin{aligned}220 &> 200 + 7.999.800 \times p, \\ 20 &> 7.999.800 \times p, \\ p &< 0{,}0000025.\end{aligned}$$

Es decir, para que la reducción de velocidad debida a la paginación sea razonable, sólo podemos permitir que provoque un fallo de página un acceso a memoria de cada 399.990. En resumen, resulta crucial mantener una tasa de fallos de página baja en los sistemas de paginación bajo demanda. Si no se hace así, el tiempo efectivo de acceso se incrementa, ralentizando enormemente la ejecución de los procesos.

Un aspecto adicional de la paginación bajo demanda es la gestión y el uso global del espacio de intercambio. Las operaciones de E/S de disco dirigidas al espacio de intercambio son generalmente más rápidas que las correspondientes al sistema de archivos, porque el espacio de intercambios se asigna en bloques mucho mayores y no se utilizan mecanismos de búsqueda de archivos ni métodos de asignación indirecta (Capítulo 12). El sistema puede, por tanto, conseguir una mayor tasa de transferencia para paginación copiando la imagen completa de un archivo en el espacio de intercambio en el momento de iniciar el proceso y luego realizando una paginación bajo demanda a partir del espacio de intercambio. Otra opción consiste en demandar las páginas inicialmente desde el sistema de archivos, pero irlas escribiendo en el espacio de intercambio a medida que se las sustituye. Esta técnica garantiza que sólo se lean desde el sistema de archivos las páginas necesarias y que todas las operaciones subsiguientes de paginación se lleven a cabo desde el espacio de intercambio.

Algunos sistemas tratan de limitar la cantidad de espacio de intercambio utilizado mediante mecanismos de paginación bajo demanda de archivos binarios. Las páginas demandadas de tales

archivos se cargan directamente desde el sistema de archivos; sin embargo, cuando hace falta sustituir páginas, estos marcos pueden simplemente sobreescribirse (porque nunca se han modificado) y las páginas pueden volver a leerse desde el sistema de archivos en caso necesario. Utilizando esta técnica, el propio sistema de archivos sirve como almacén de respaldo. Sin embargo, seguirá siendo necesario utilizar espacio de intercambio para las páginas que no estén asociadas con un archivo. Estas páginas incluyen la pila y el cúmulo de cada proceso. Este método parecer ser un buen compromiso y es el que se utiliza en varios sistemas, incluyendo Solaris y BSD UNIX.

9.3 Copia durante la escritura

En la Sección 9.2 hemos indicado cómo podría un proceso comenzar rápidamente, cargando únicamente en memoria la página que contuviera la primera instrucción. Sin embargo, la creación de un proceso mediante la llamada al sistema `fork()` puede inicialmente evitar que se tenga que cargar ninguna página, utilizando una técnica similar a la de compartición de páginas que se ha descrito en la Sección 8.4.4. Esta técnica permite la creación rápida de procesos y minimiza el número de nuevas páginas que deben asignarse al proceso recién creado.

Recuerde que la llamada al sistema `fork()` crea un proceso hijo como duplicado de su padre. Tradicionalmente, `fork()` trabajaba creando una copia del espacio de direcciones del padre para el hijo, duplicando las páginas que pertenecen al padre. Sin embargo, considerando que muchos procesos hijos invocan la llamada al sistema `exec()` inmediatamente después de su creación, puede que sea innecesaria la copia del espacio de direcciones del padre. Como alternativa, podemos utilizar una técnica conocida con el nombre de **copia durante la escritura**, que funciona permitiendo que los procesos padre e hijo compartan inicialmente las mismas páginas. Estas páginas compartidas se marcan como páginas de copia durante la escritura, lo que significa que si cualquiera de los procesos escribe en una de las páginas compartidas, se creará una copia de esa página compartida. El proceso de copia durante la escritura se ilustra en las Figuras 9.7 y 9.8, que muestran el contenido de la memoria física antes y después de que el proceso 1 modifique la página C.

Por ejemplo, suponga que el proceso hijo trata de modificar una página que contiene parte de la pila, estando las páginas definidas como de copia durante la escritura. El sistema operativo creará entonces una copia de esta página, y la mapeará sobre el espacio de direcciones del proceso hijo. El proceso hijo modificará entonces su página copiada y no la página que pertenece al proceso padre. Obviamente, cuando se utiliza la técnica de copia durante la escritura, sólo se copian las páginas que sean modificadas por algunos de los procesos; todas las páginas no modificadas podrán ser compartidas por los procesos padre e hijo. Observe también que sólo es necesario marcar como de copia durante la escritura aquellas páginas que puedan ser modificadas. Las páginas que no puedan modificarse (páginas que contengan el código ejecutable) pueden compartirse entre el padre y el hijo. La técnica de copia durante la escritura resulta común en varios sistemas operativos, incluyendo Windows XP, Linux y Solaris.

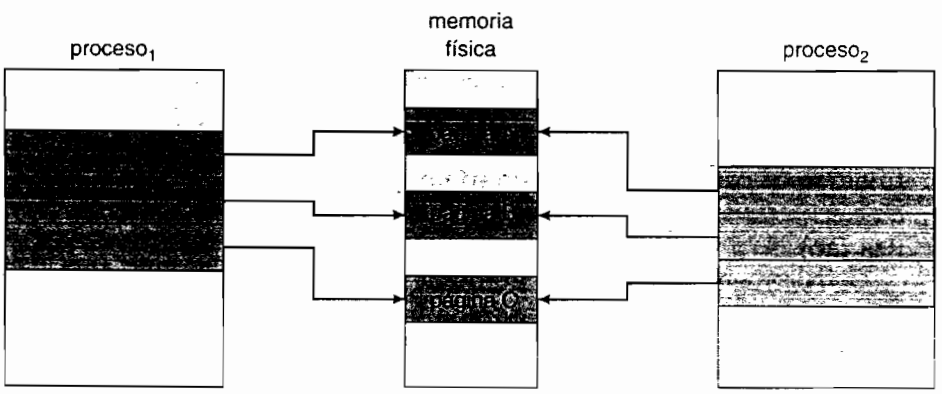

Figura 9.7 Antes de que el proceso 1 modifique la página C.