Capítulo 33

Paginación

La fragmentación externa y, por tanto, la necesidad de compactación pueden evitarse por completo empleando la *paginación*. Ésta consiste en que cada proceso está dividio en varios bloques de tamaño fijo (más pequeños que los segmentos) llamados *páginas*, dejando de requerir que la asignación sea de un área *contigua* de memoria. Claro está, esto requiere de mayor soporte por parte del hardware, y mayor información relacionada a cada uno de los procesos: no basta sólo con indicar dónde inicia y termina el área de memoria de cada proceso, sino que se debe establecer un *mapeo* entre la ubicación real (*física*) y la presentada a cada uno de los procesos (*lógica*). La memoria se presentará a cada proceso como si fuera de su uso exclusivo.

La memoria física se divide en una serie de *marcos* (*frames*), todos ellos del mismo tamaño, y el espacio para cada proceso se divide en una serie de páginas (*pages*), del mismo tamaño que los marcos. La MMU se encarga del mapeo entre páginas y marcos mediante *tablas de páginas*.

Cuando se trabaja bajo una arquitectura que maneja paginación, las direcciones que maneja el CPU ya no son presentadas de forma absoluta. Los bits de cada dirección se separan en un *identificador de página* y un *desplazamiento*, de forma similar a lo presentado al hablar de resolución de instrucciones en tiempo de ejecución. La principal diferencia con lo entonces abordado es que cada proceso tendrá ya no un único espacio en memoria, sino una multitud de páginas.

El tamaño de los marcos (y, por tanto, las páginas) debe ser una *potencia de dos*, de modo que la MMU pueda discernir fácilmente la porción de una dirección de memoria que se refiere a la *página* del *desplazamiento*. El rango

varía, según el hardware, entre los 512 bytes (2^9) y 16 MB (2^{24}); al ser una potencia de dos, la MMU puede separar la dirección en memoria entre los primeros *m* bits (referentes a la página) y los últimos *n* bits (referentes al desplazamiento).

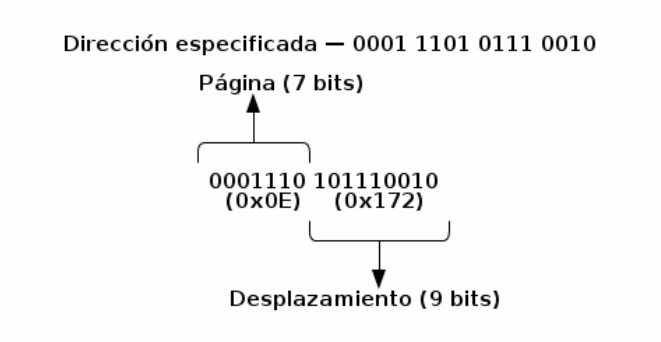

Figura 33.1: Página y desplazamiento, en un esquema de direccionamiento de 16 bits y páginas de 512 bytes.

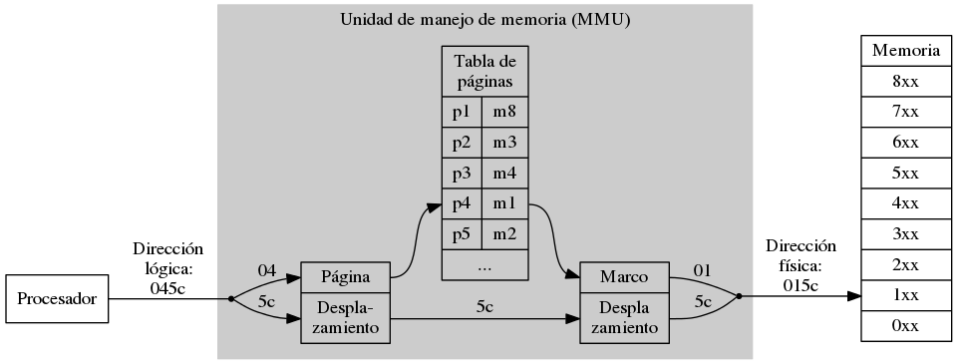

Figura 33.2: Esquema del proceso de paginación, ilustrando el papel de la MMU.

Para poder realizar este mapeo, la MMU requiere de una estructura de datos denominada *tabla de páginas* (*page table*), que *resuelve* la relación entre páginas y marcos, convirtiendo una *dirección lógica* (en el espacio del proceso) en la *dirección física* (la ubicación en que *realmente* se encuentra en la memoria del sistema).

Se puede tomar como ejemplo para explicar este mecanismo el esquema

presentado en la figura 33.3 (Silberschatz, Galvin y Gagne 2010: 292). Éste muestra un esquema minúsculo de paginación: un *espacio de direccionamiento* de 32 bytes (cinco bits), organizado en ocho páginas de cuatro bytes cada una (esto es, la página es representada con los tres bits *más significativos* de la dirección, y el desplazamiento con los dos bits *menos significativos*).

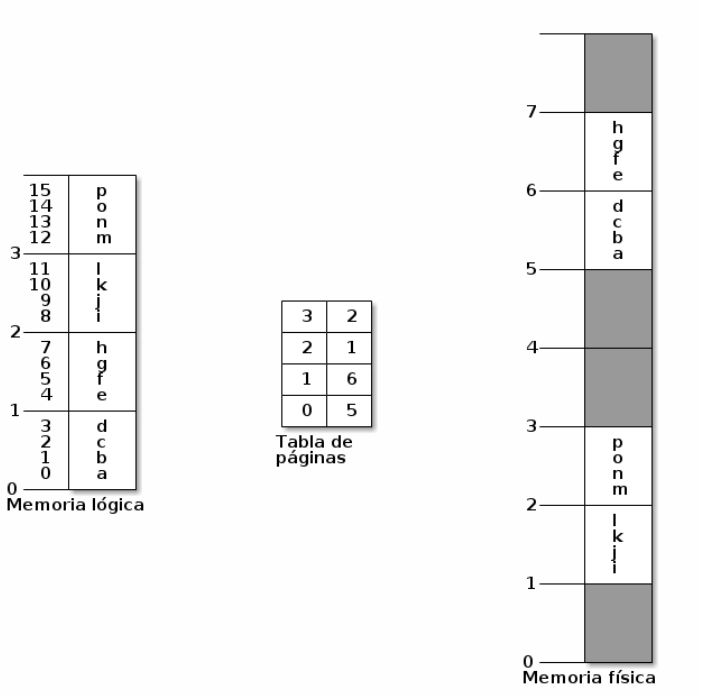

Figura 33.3: Ejemplo (minúsculo) de paginación, con un espacio de direccionamiento de 32 bytes y páginas de cuatro bytes.

El proceso que se presenta tiene una visión de la memoria como la columna del lado izquierdo: para el proceso hay cuatro páginas, y tiene sus datos distribuidos en orden desde la dirección 00 000 (0) hasta la 01 111 (15), aunque en realidad en el sistema éstas se encuentren desordenadas y ubicadas en posiciones no contiguas.

Cuando el proceso quiere referirse a la letra *f*, lo hace indicando la dirección 00 101 (5). De esta dirección, los tres bits más significativos (001, 1 —y para la computadora, lo *natural* es comenzar a contar por el 0) se refieren a la página uno, y los dos bits menos significativos (01, 1) indican al *desplazamiento* dentro de ésta.

La MMU verifica en la tabla de páginas, y encuentra que la página 1 corresponde al marco número 6 (110), por lo que traduce la dirección lógica 00 101 (5) a la física 11 001 (26).

Se puede tomar la paginación como una suerte de resolución o traducción de direcciones en tiempo de ejecución, pero con una *base* distinta para cada una de las páginas.

33.1. Tamaño de la página

Ahora, si bien la fragmentación externa se resuelve al emplear paginación, el problema de la fragmentación interna persiste: al dividir la memoria en bloques de longitud preestablecida de 2^n bytes, un proceso en promedio desperdiciará $\frac{2^n}{2}$ (y, en el peor de los casos, hasta $2^n - 1$). Multiplicando esto por el número de procesos que están en ejecución en todo momento en el sistema, para evitar que una proporción sensible de la memoria se pierda en fragmentación interna, se podría tomar como estrategia emplear un tamaño de página tan pequeño como fuera posible.

Sin embargo, la sobrecarga administrativa (el tamaño de la tabla de paginación) en que se incurre por gestionar demasiadas páginas pequeñas se vuelve una limitante en sentido opuesto:

- Las transferencias entre unidades de disco y memoria son mucho más eficientes si pueden mantenerse como recorridos continuos. El controlador de disco puede responder a solicitudes de acceso directo a memoria (DMA) siempre que tanto los fragmentos en disco como en memoria sean continuos; fragmentar la memoria demasiado jugaría en contra de la eficiencia de estas solicitudes.

- El bloque de control de proceso (PCB) incluye la información de memoria. Entre más páginas tenga un proceso (aunque éstas fueran muy pequeñas), más grande es su PCB, y más información requerirá intercambiar en un cambio de contexto.

Estas consideraciones opuestas apuntan a que se debe mantener el tamaño de página más grande, y se regulan con las primeras expuestas en esta sección.

Hoy en día, el tamaño habitual de las páginas es de 4 u 8 KB (2^{12} o 2^{13} bytes). Hay algunos sistemas operativos que soportan múltiples tamaños

de página — por ejemplo, Solaris puede emplear páginas de 8 KB y 4 MB (2^{13} o 2^{22} bytes), dependiendo del tipo de información que se declare que almacenarán.

33.2. Almacenamiento de la tabla de páginas

Algunos de los primeros equipos en manejar memoria paginada empleaban un conjunto especial de registros para representar la tabla de páginas. Esto era posible dado que eran sistemas de 16 bits, con páginas de 8 KB (2^{13}). Esto significa que contaban únicamente con ocho páginas posibles ($16 - 13 = 3; 2^3 = 8$), por lo que resultaba sensato dedicar un registro a cada una.

En los sistemas actuales, mantener la tabla de páginas en registros resultaría claramente imposible: teniendo un procesador de 32 bits, e incluso si se definiera un tamaño de página *muy* grande (por ejemplo, 4 MB), existirían 1 024 páginas posibles;[1] con un tamaño de páginas mucho más común (4 KB, 2^{12} bytes), la tabla de páginas llega a ocupar 5 MB.[2] Los registros son muy rápidos, sin embargo, son en correspondencia muy caros. El manejo de páginas más pequeñas (que es lo normal), y muy especialmente el uso de espacios de direccionamiento de 64 bits, harían prohibitivo este enfoque. Además, nuevamente, cada proceso tiene una tabla de páginas distinta —se haría necesario hacer una transferencia de información muy grande en cada cambio de contexto.

Otra estrategia para enfrentar esta situación es almacenar la propia tabla de páginas en memoria, y apuntar al inicio de la tabla con un juego de registros especiales: el *registro de base de la tabla de páginas* (PTBR, *page table base register*) y el *registro de longitud de la tabla de páginas* (PTLR, *page table length register*).[3] De esta manera, en el cambio de contexto sólo hay que cambiar estos dos registros, y además se cuenta con un espacio muy amplio para guardar las tablas de páginas que se necesiten. El problema con este mecanismo es la velocidad: Se estaría penalizando a *cada acceso a memoria*

[1] 4 MB es 2^{22} bytes; $\frac{2^{32}}{2^{22}} = 2^{10} = 1\,024$.

[2] $\frac{2^{32}}{2^{12}} = 2^{20} = 1\,048\,576$, cada entrada con un mínimo de 20 bits para la página y otros 20 para el marco. ¡La tabla de páginas misma ocuparía 1˜280 páginas!

[3] ¿Por qué es necesario el segundo? Porque es prácticamente imposible que un proceso emplee su espacio de direccionamiento completo; al indicar el límite máximo de su tabla de páginas por medio del PTLR se evita desperdiciar grandes cantidades de memoria indicando todo el espacio no utiilzado.

con uno adicional —si para resolver una dirección lógica a su correspondiente dirección física hace falta consultar la tabla de páginas en memoria, el tiempo efectivo de acceso a memoria se duplica.

33.2.1. El buffer de traducción adelantada (TLB)

La salida obvia a este dilema es el uso de un caché. Sin embargo, más que un caché genérico, la MMU utiliza un caché especializado en el tipo de información que maneja: el *buffer de traducción adelantada* o *anticipada*. (*Translation lookaside buffer*. El TLB es una tabla asociativa (un *hash*) en memoria de alta velocidad, una suerte de registros residentes dentro de la MMU, donde las *llaves* son las páginas y los *valores* son los marcos correspondientes. De este modo, las búsquedas se efectúan en tiempo constante.

El TLB típicamente tiene entre 64 y 1 024 entradas. Cuando el procesador efectúa un acceso a memoria, si la página solicitada está en el TLB, la MMU tiene la dirección física de inmediato.[4] En caso de no encontrarse la página en el TLB, la MMU lanza un *fallo de página* (*page fault*), con lo cual consulta de la memoria principal cuál es el marco correspondiente. Esta nueva entrada es agregada al TLB; por las propiedades de *localidad de referencia* que se presentaron anteriormente, la probabilidad de que las regiones más empleadas de la memoria durante un área específica de ejecución del programa sean cubiertas por relativamente pocas entradas del TLB son muy altas.

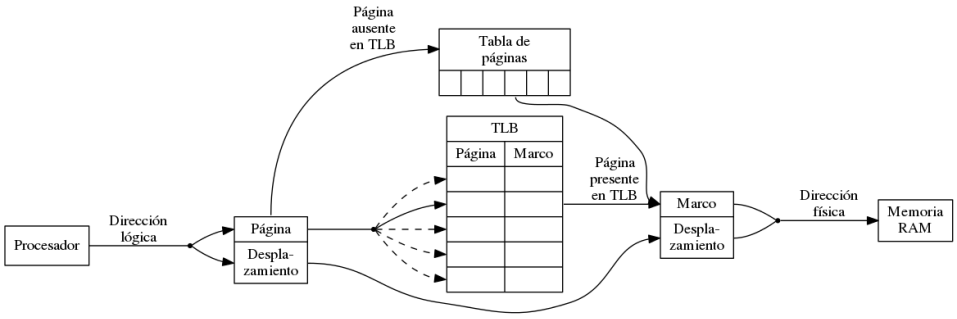

Figura 33.4: Esquema de paginación empleando un *buffer de traducción adelantada* (TLB).

[4] El tiempo efectivo de acceso puede ser 10% superior al que tomaría sin emplear paginación (Silberschatz, Galvin y Gagne 2010: 295).

Como sea, dado que el TLB es limitado en tamaño, es necesario explicitar una política que indique dónde guardar las nuevas entradas (esto es, qué entrada reemplazar) una vez que el TLB está lleno y se produce un fallo de página. Uno de los esquemas más comunes es emplear la entrada *menos recientemente utilizada* (LRU, *Least Recently Used*), nuevamente apelando a la localidad de referencia. Esto tiene como consecuencia necesaria que debe haber un mecanismo que contabilice los accesos dentro del TLB (lo cual agrega tanto latencia como costo). Otro mecanismo (con obvias desventajas) es el reemplazar una página al azar. Se explicarán con mayor detalle, más adelante, algunos de los mecanismos más empleados para este fin, comparando sus puntos a favor y en contra.

33.2.2. Subdividiendo la tabla de páginas

Incluso empleando un TLB, el espacio empleado por las páginas sigue siendo demasiado grande. Si se considera un escenario más frecuente que el propuesto anteriormente: empleando un procesador con espacio de direccionamiento de 32 bits, y un tamaño de página estándar (4 KB, 2^{12}), se tendría 1 048 576 (2^{20}) páginas. Si cada entrada de la página ocupa 40 bits[5] (esto es, cinco bytes), cada proceso requeriría de 5 MB (cinco bytes por cada una de las páginas) sólamente para representar su mapeo de memoria. Esto, especialmente en procesos pequeños, resultaría más gravoso para el sistema que los beneficios obtenidos de la paginación.

Aprovechando que la mayor parte del espacio de direccionamiento de un proceso está típicamente vacío (la pila de llamadas y el heap), se puede subdividir el identificador de página en dos (o más) niveles, por ejemplo, separando una dirección de 32 bits en una *tabla externa* de 10, una *tabla interna* de 10, y el *desplazamiento* de 12 bits.

Este esquema funciona adecuadamente para computadoras con direccionamiento de hasta 32 bits. Sin embargo, se debe considerar que cada nivel de páginas conlleva un acceso adicional a memoria en caso de fallo de página —emplear paginación jerárquica con un nivel externo y uno interno implica que un fallo de página *triplica* (y no duplica, como sería con un esquema de paginación directo) el tiempo de acceso a memoria. Para obtener una tabla de páginas manejable bajo los parámetros aquí descritos en un sistema de 64 bits, se puede *septuplicar* el tiempo de acceso (cinco ac-

[5] 20 bits identificando a la página y otros 20 bits al marco; omitiendo aquí la necesidad de alinear los accesos a memoria a *bytes* individuales, que lo aumentarían a 24.

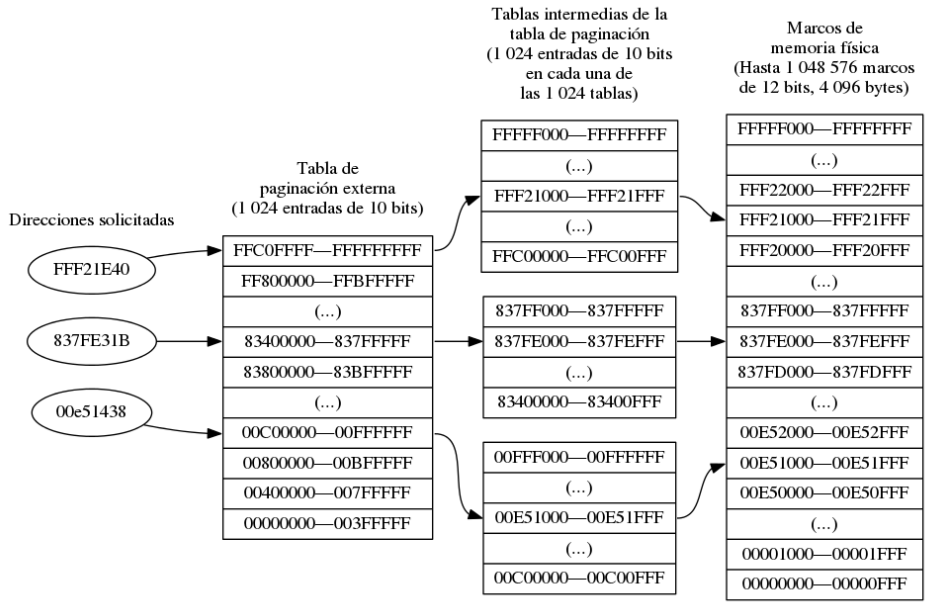

Figura 33.5: Paginación en dos niveles: una tabla externa de 10 bits, tablas intermedias de 10 bits, y marcos de 12 bits (esquema común para procesadores de 32 bits).

cesos *en cascada* para fragmentos de 10 bits, y un tamaño de página de 14 bits, más el acceso a la página destino).

Otra alternativa es emplear *funciones digestoras* (*hash functions*)[6] para mapear cada una de las páginas a un *espacio muestral* mucho más pequeño. Cada página es mapeada a una lista de correspondencias simples.[7]

Un esquema basado en funciones digestoras ofrece características muy deseables: el tamaño de la tabla de páginas puede variar según crece el uso de memoria de un proceso (aunque esto requiera recalcular la tabla con diferentes parámetros) y el número de accesos a memoria en espacios tan grandes como el de un procesador de 64 bits se mantiene mucho más

[6]Una función digestora puede definirse como $H : U \rightarrow M$, una función que *mapea* o *proyecta* al conjunto U en un conjunto M mucho menor; una característica muy deseable de toda función hash es que la *distribución resultante* en M sea homogénea y tan poco dependiente de la secuencialidad de la entrada como sea posible.

[7]A una lista y no a un valor único dado que una función digestora es necesariamente proclive a presentar *colisiones*; el sistema debe poder resolver dichas colisiones sin pérdida de información.

tratable. Sin embargo, por la alta frecuencia de accesos a esta tabla, debe elegirse un algoritmo digestor muy ágil para evitar que el tiempo que tome calcular la posición en la tabla resulte significativo frente a las alternativas.

33.3. Memoria compartida

Hay muchos escenarios en que diferentes procesos pueden beneficiarse de compartir áreas de su memoria. Uno de ellos es como mecanismo de comunicación entre procesos (IPC, *inter process communication*), en que dos o más procesos pueden intercambiar estructuras de datos complejas sin incurrir en el costo de copiado que implicaría copiarlas por medio del sistema operativo.

Otro caso, mucho más frecuente, es el de *compartir código*. Si un mismo programa es ejecutado varias veces, y dicho programa no emplea mecanismos de *código auto-modificable*, no tiene sentido que las páginas en que se representa cada una de dichas instancias ocupe un marco independiente —el sistema operativo puede asignar a páginas de diversos procesos *el mismo conjunto de marcos*, con lo cual puede aumentar la capacidad percibida de memoria.

Y si bien es muy común compartir los *segmentos de texto* de los diversos programas que están en un momento dado en ejecución en la computadora, este mecanismo es todavía más útil cuando se usan *bibliotecas del sistema*: hay bibliotecas que son empleadas por una gran cantidad de programas[8].

Claro está, para ofrecer este modelo, el sistema operativo debe garantizar que las páginas correspondientes a las *secciones de texto* (el código del programa) sean de sólo lectura.

Un programa que está desarrollado y compilado de forma que permita que todo su código sea de sólo lectura posibilita que diversos procesos entren a su espacio en memoria sin tener que sincronizarse con otros procesos que lo estén empleando.

[8] Algunos ejemplos sobresalientes podrían ser la `libc` o `glibc`, que proporciona las funcinoes estándar del lenguaje C y es, por tanto, requerida por casi todos los programas del sistema; los diferentes entornos gráficos (en los Unixes modernos, los principales son `Qt` y `Gtk`); bibliotecas para el manejo de cifrado (`openssl`), compresión (`zlib`), imágenes (`libpng`, `libjpeg`), etcétera.

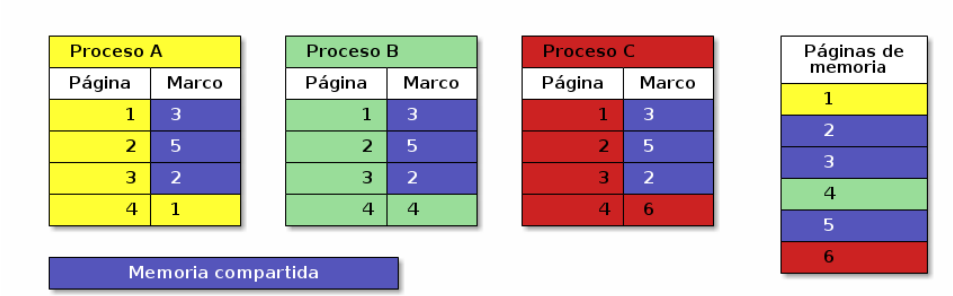

Figura 33.6: Uso de memoria compartida: tres procesos comparten la memoria ocupada por el texto del programa (azul), difieren sólo en los datos.

33.3.1. Copiar al escribir (*copy on write, CoW*)

En los sistemas Unix, el mecanismo más frecuentemente utilizado para crear un nuevo proceso es el empleo de la llamada al sistema `fork()`. Cuando es invocado por un proceso, el sistema operativo crea un nuevo proceso *idéntico* al que lo llamó, diferenciándose únicamente en *el valor entregado* por la llamada a `fork()`. Si ocurre algún error, el sistema entrega un número negativo (indicando la causa del error). En caso de ser exitoso, el proceso nuevo (o proceso *hijo*) recibe el valor `0`, mientras que el preexistente (o proceso padre) recibe el PID (número identificador de proceso) del hijo. Es frecuente encontrar el siguiente código:

```
/* (...) */
int pid;
/* (...) */
pid = fork();
if (pid == 0) {
  /* Soy el proceso hijo */
  /* (...) */
 } else if (pid < 0) {
  /* Ocurrió un error, no se creó un proceso hijo */
 } else {
  /* Soy el proceso padre */
  /* La variable 'pid' tiene el PID del proceso hijo */
  /* (...) */
 }
```

Este método es incluso utilizado normalmente para crear nuevos procesos, transfiriendo el *ambiente* (variables, por ejemplo, que incluyen cuál es la *entrada* y *salida* estándar de un proceso, esto es, a qué terminal están conectados, indispensable en un sistema multiusuario). Frecuentemente, la siguiente instrucción que ejecuta un proceso hijo es `execve()`, que carga a un nuevo programa sobre el actual y transfiere la ejecución a su primera instrucción.

Cuesta trabajo comprender el por qué de esta lógica si no es por el empleo de la memoria compartida: el costo de `fork()` en un sistema Unix es muy bajo, se limita a crear las estructuras necesarias en la memoria del núcleo. Tanto el proceso padre como el proceso hijo comparten *todas* sus páginas de memoria, como lo ilustra la figura 33.7(a), sin embargo, siendo dos procesos independientes, no deben poder modificarse más que por los canales explícitos de comunicación entre procesos.

Esto ocurre así gracias al mecanismo llamado *copiar al escribir* (frecuentemente referido por sus siglas en inglés, *CoW*). Las páginas de memoria de ambos procesos son las mismas *mientras sean sólo leídas*. Sin embargo, si uno de los procesos modifica cualquier dato en una de estas páginas, ésta se copia a un nuevo marco, y deja de ser una página compartida, como se puede ver en la figura 33.7(b). El resto de las páginas seguirá siendo compartido. Esto se puede lograr marcando *todas* las páginas compartidas como *sólo lectura*, con lo cual cuando uno de los dos procesos intente modificar la información de alguna página se generará un fallo. El sistema operativo, al notar que esto ocurre sobre un espacio CoW, en vez de responder al fallo terminando al proceso, copiará sólo la página en la cual se encuentra la dirección de memoria que causó el fallo, y esta vez marcará la página como *lectura y escritura*.

Incluso cuando se ejecutan nuevos programas mediante `execve()`, es posible que una buena parte de la memoria se mantenga compartida, por ejemplo, al referirse a copias de bibliotecas de sistema.