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

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

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

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

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

3.4 Comunicación interprocesos

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

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

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

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

jes, la comunicación tiene lugar mediante el intercambio de mensajes entre los procesos cooperativos. En la Figura 3.13 se comparan los dos modelos de comunicación.

Los dos modelos que acabamos de presentar son bastante comunes en los distintos sistemas operativos y muchos sistemas implementan ambos. El paso de mensajes resulta útil para intercambiar pequeñas cantidades de datos, ya que no existe la necesidad de evitar conflictos. El paso de mensajes también es más fácil de implementar que el modelo de memoria compartida como mecanismo de comunicación entre computadoras. La memoria compartida permite una velocidad máxima y una mejor comunicación, ya que puede realizarse a velocidades de memoria cuando se hace en una misma computadora. La memoria compartida es más rápida que el paso de mensajes, ya que este último método se implementa normalmente usando llamadas al sistema y, por tanto, requiere que intervenga el *kernel*, lo que consume más tiempo. Por el contrario, en los sistemas de memoria compartida, las llamadas al sistema sólo son necesarias para establecer las zonas de memoria compartida. Una vez establecida la memoria compartida, todos los accesos se tratan como accesos a memoria rutinarios y no se precisa la ayuda del *kernel*. En el resto de esta sección, nos ocupamos en detalle de cada uno de estos modelos de comunicación IPC.

3.4.1 Sistemas de memoria compartida

La comunicación interprocesos que emplea memoria compartida requiere que los procesos que se estén comunicando establezcan una región de memoria compartida. Normalmente, una región de memoria compartida reside en el espacio de direcciones del proceso que crea el segmento de memoria compartida. Otros procesos que deseen comunicarse usando este segmento de memoria compartida deben conectarse a su espacio de direcciones. Recuerde que, habitualmente, el sistema operativo intenta evitar que un proceso acceda a la memoria de otro proceso. La memoria compartida requiere que dos o más procesos acuerden eliminar esta restricción. Entonces podrán intercambiar información leyendo y escribiendo datos en las áreas compartidas. El formato de los datos y su ubicación están determinados por estos procesos, y no se encuentran bajo el control del sistema operativo. Los procesos también son responsables de verificar que no escriben en la misma posición simultáneamente.

Para ilustrar el concepto de procesos cooperativos, consideremos el problema del productor-consumidor, el cual es un paradigma comúnmente utilizado para los procesos cooperativos. Un proceso **productor** genera información que consume un proceso **consumidor**. Por ejemplo, un compilador puede generar código ensamblado, que consume un ensamblador. El ensamblador, a su vez, puede generar módulos objeto, que consume el cargador. El problema del productor-consumidor también proporciona una metáfora muy útil para el paradigma cliente-servidor.

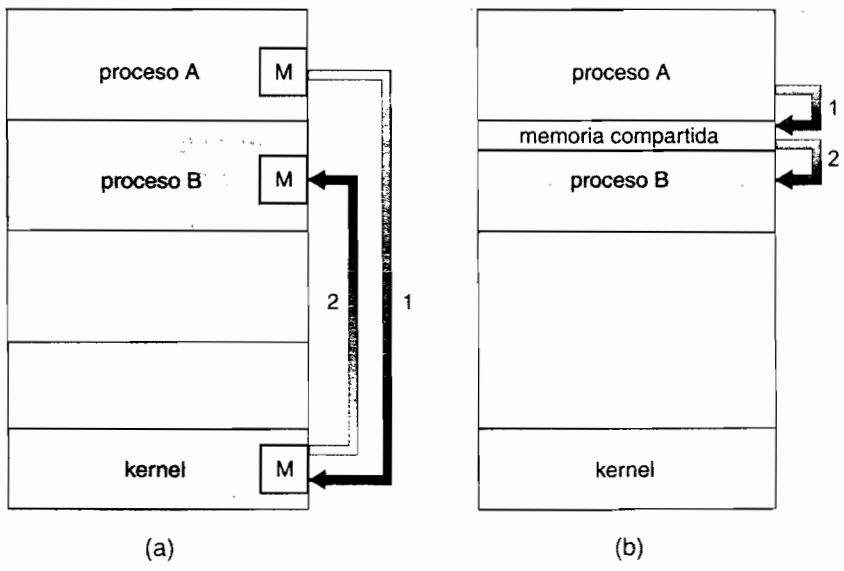

Figura 3.13 Modelos de comunicación. (a) Paso de mensajes. (b) Memoria compartida.

Generalmente, pensamos en un servidor como en un productor y en un cliente como en un consumidor. Por ejemplo, un servidor web produce (es decir, proporciona) archivos HTML e imágenes, que consume (es decir, lee) el explorador web cliente que solicita el recurso.

Una solución para el problema del productor-consumidor es utilizar mecanismos de memoria compartida. Para permitir que los procesos productor y consumidor se ejecuten de forma concurrente, debemos tener disponible un búfer de elementos que pueda rellenar el productor y vaciar el consumidor. Este búfer residirá en una región de memoria que será compartida por ambos procesos, consumidor y productor. Un productor puede generar un elemento mientras que el consumidor consume otro. El productor y el consumidor deben estar sincronizados, de modo que el consumidor no intente consumir un elemento que todavía no haya sido producido.

Pueden emplearse dos tipos de búferes. El sistema de **búfer no limitado** no pone límites al tamaño de esa memoria compartida. El consumidor puede tener que esperar para obtener elementos nuevos, pero el productor siempre puede generar nuevos elementos. El sistema de **búfer limitado** establece un tamaño de búfer fijo. En este caso, el consumidor tiene que esperar si el búfer está vacío y el productor tiene que esperar si el búfer está lleno.

Veamos más en detalle cómo puede emplearse un búfer limitado para permitir que los procesos compartan la memoria. Las siguientes variables residen en una zona de la memoria compartida por los procesos consumidor y productor:

```
#define BUFFER_SIZE 10

typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

El búfer compartido se implementa como una matriz circular con dos punteros lógicos: `in` y `out`. La variable `in` apunta a la siguiente posición libre en el búfer; `out` apunta a la primera posición ocupada del búfer. El búfer está vacío cuando `in == out`; el búfer está lleno cuando `((in + 1) %BUFFER_SIZE) == out`.

El código para los procesos productor y consumidor se muestra en las Figuras 3.14 y 3.15, respectivamente. El proceso productor tiene una variable local, `nextProduced`, en la que se almacena el elemento nuevo que se va a generar. El proceso consumidor tiene una variable local, `nextConsumed`, en la que se almacena el elemento que se va a consumir.

Este esquema permite tener como máximo BUFFER_SIZE − 1 elementos en el búfer al mismo tiempo. Dejamos como ejercicio para el lector proporcionar una solución en la que BUFFER_SIZE elementos puedan estar en el búfer al mismo tiempo. En la Sección 3.5.1 se ilustra la API de POSIX para los sistemas de memoria compartida.

Un problema del que no se ocupa este ejemplo es la situación en la que tanto el proceso productor como el consumidor intentan acceder al búfer compartido de forma concurrente. En el Capítulo 6 veremos cómo puede implementarse la sincronización entre procesos cooperativos de forma efectiva en un entorno de memoria compartida.

```
item nextProduced;

while (true) {
     * produce e inserta un elemento en nextProduced*/
    while ((in+1) % BUFFER_SIZE) == out)
      ;  /*no hacer nada*/
    buffer[in]=nextProduced;
    in = (in+1) % BUFFER_SIZE;
}
```

Figura 3.14 El proceso productor.

```
item nextConsumed;

while (true) {
     while (in == out)
       ;  /*no hacer nada*/

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume el elemento almacenado en nextConsumed */
}
```

Figura 3.15 El proceso consumidor.

3.4.2 Sistemas de paso de mensajes

En la Sección 3.4.1 hemos mostrado cómo pueden comunicarse procesos cooperativos en un entorno de memoria compartida. El esquema requiere que dichos procesos compartan una zona de la memoria y que el programador de la aplicación escriba explícitamente el código para acceder y manipular la memoria compartida. Otra forma de conseguir el mismo efecto es que el sistema operativo proporcione los medios para que los procesos cooperativos se comuniquen entre sí a través de una facilidad de paso de mensajes.

El paso de mensajes proporciona un mecanismo que permite a los procesos comunicarse y sincronizar sus acciones sin compartir el mismo espacio de direcciones, y es especialmente útil en un entorno distribuido, en el que los procesos que se comunican pueden residir en diferentes computadoras conectadas en red. Por ejemplo, un programa de **chat** utilizado en la World Wide Web podría diseñarse de modo que los participantes en la conversación se comunicaran entre sí intercambiando mensajes.

Una facilidad de paso de mensajes proporciona al menos dos operaciones: envío de mensajes (`send`) y recepción de mensajes (`receive`). Los mensajes enviados por un proceso pueden tener un tamaño fijo o variable. Si sólo se pueden enviar mensajes de tamaño fijo, la implementación en el nivel de sistema es directa. Sin embargo, esta restricción hace que la tarea de programación sea más complicada. Por el contrario, los mensajes de tamaño variable requieren una implementación más compleja en el nivel de sistema, pero la tarea de programación es más sencilla. Éste es un tipo de compromiso que se encuentra muy habitualmente en el diseño de sistemas operativos.

Si los procesos *P* y *Q* desean comunicarse, tienen que enviarse mensajes entre sí; debe existir un **enlace de comunicaciones** entre ellos. Este enlace se puede implementar de diferentes formas. No vamos a ocuparnos aquí de la implementación física del enlace (memoria compartida, bus hardware o red), que se verá en el Capítulo 16, sino de su implementación lógica. Existen varios métodos para implementar lógicamente un enlace y las operaciones de envío y recepción:

- Comunicación directa o indirecta.
- Comunicación síncrona o asíncrona.
- Almacenamiento en búfer explícito o automático.

Veamos ahora los problemas relacionados con cada una de estas funcionalidades.

3.4.2.1 Nombrado

Los procesos que se van a comunicar deben disponer de un modo de referenciarse entre sí. Pueden usar comunicación directa o indirecta.

En el caso de la **comunicación directa**, cada proceso que desea establecer una comunicación debe nombrar de forma explícita al receptor o transmisor de la comunicación. En este esquema, las primitivas `send()` y `receive()` se definen del siguiente modo:

- `send(P, mensaje)`— Envía un `mensaje` al proceso P.

- `receive(Q, mensaje)`— Recibe un `mensaje` del proceso Q.

Un enlace de comunicaciones, según este esquema, tiene las siguientes propiedades:

- Los enlaces se establecen de forma automática entre cada par de procesos que quieran comunicarse. Los procesos sólo tienen que conocer la identidad del otro para comunicarse.
- Cada enlace se asocia con exactamente dos procesos.
- Entre cada par de procesos existe exactamente un enlace.

Este esquema presenta *simetría* en lo que se refiere al direccionamiento, es decir, tanto el proceso transmisor como el proceso receptor deben nombrar al otro para comunicarse. Existe una variante de este esquema que emplea *asimetría* en el direccionamiento. En este caso, sólo el transmisor nombra al receptor; el receptor no tiene que nombrar al transmisor. En este esquema, las primitivas `send()` y `receive()` se definen del siguiente modo:

- `send(P, mensaje)`— Envía un `mensaje` al proceso P
- `receive(id, mensaje)`— Recibe un `mensaje` de cualquier proceso; a la variable *id* se le asigna el nombre del proceso con el que se ha llevado a cabo la comunicación.

La desventaja de estos dos esquemas (simétrico y asimétrico) es la limitada modularidad de las definiciones de procesos resultantes. Cambiar el identificador de un proceso puede requerir que se modifiquen todas las restantes definiciones de procesos. Deben localizarse todas las referencias al identificador antiguo, para poder sustituirlas por el nuevo identificador. En general, cualquier técnica de **precodificación**, en la que los identificadores deban establecerse explícitamente, es menos deseable que las técnicas basadas en la indirección, como se describe a continuación.

Con el modelo de comunicación indirecta, los mensajes se envían y reciben en **buzones de correo** o **puertos**. Un buzón de correo puede verse de forma abstracta como un objeto en el que los procesos pueden colocar mensajes y del que pueden eliminar mensajes. Cada buzón de correo tiene asociada una identificación unívoca. Por ejemplo, las colas de mensajes de POSIX usan un valor entero para identificar cada buzón de correo. En este esquema, un proceso puede comunicarse con otros procesos a través de una serie de buzones de correo diferentes. Sin embargo, dos procesos sólo se pueden comunicar si tienen un buzón de correo compartido. Las primitivas `send()` y `receive()` se definen del siguiente modo:

- `send(A, mensaje)`— Envía un `mensaje` al buzón de correo `A`.
- `receive(A, mensaje)`— Recibe un `mensaje` del buzón de correo `A`.

En este esquema, un enlace de comunicaciones tiene las siguientes propiedades:

- Puede establecerse un enlace entre un par de procesos sólo si ambos tienen un buzón de correo compartido.
- Un enlace puede asociarse con más de dos procesos.
- Entre cada par de procesos en comunicación, puede haber una serie de enlaces diferentes, correspondiendo cada enlace a un buzón de correo.

Ahora supongamos que los procesos P_1, P_2 y P_3 comparten el buzón de correo A. EL proceso P_1 envía un mensaje a A, mientras que los procesos P_2 y P_3 ejecutan una instrucción `receive()` de A. ¿Qué procesos recibirán el mensaje enviado por P_1? La respuesta depende de cuál de los métodos siguientes elijamos:

- Permitir que cada enlace esté asociado como máximo con dos procesos.
- Permitir que sólo un proceso, como máximo, ejecute una operación de recepción en cada momento.
- Permitir que el sistema seleccione arbitrariamente qué proceso recibirá el mensaje (es decir, P_2 o P_3, pero no ambos). El sistema también puede definir un algoritmo para seleccionar qué

proceso recibirá el mensaje (por ejemplo, que los procesos reciban por turnos los mensajes). El sistema puede identificar al receptor ante el transmisor.

Un buzón de correo puede ser propiedad de un proceso o del sistema operativo. Si es propiedad de un proceso, es decir, si el buzón de correo forma parte del espacio de direcciones del proceso, entonces podemos diferenciar entre el propietario (aquél que sólo recibe mensajes a través de este buzón) y el usuario (aquél que sólo puede enviar mensajes a dicho buzón de correo). Puesto que cada buzón de correo tiene un único propietario, no puede haber confusión acerca de quién recibirá un mensaje enviado a ese buzón de correo. Cuando un proceso que posee un buzón de correo termina, dicho buzón desaparece. A cualquier proceso que con posterioridad envíe un mensaje a ese buzón debe notificársele que dicho buzón ya no existe.

Por el contrario, un buzón de correo que sea propiedad del sistema operativo tiene existencia propia: es independiente y no está asociado a ningún proceso concreto. El sistema operativo debe proporcionar un mecanismo que permita a un proceso hacer lo siguiente:

- Crear un buzón de correo nuevo.
- Enviar y recibir mensajes a través del buzón de correo.
- Eliminar un buzón de correo.

Por omisión, el proceso que crea un buzón de correo nuevo es el propietario del mismo. Inicialmente, el propietario es el único proceso que puede recibir mensajes a través de este buzón. Sin embargo, la propiedad y el privilegio de recepción se pueden pasar a otros procesos mediante las apropiadas llamadas al sistema. Por supuesto, esta medida puede dar como resultado que existan múltiples receptores para cada buzón de correo.

3.4.2.2 Sincronización

La comunicación entre procesos tiene lugar a través de llamadas a las primitivas `send()` y `receive()`. Existen diferentes opciones de diseño para implementar cada primitiva. El paso de mensajes puede ser **con bloqueo** o **sin bloqueo**, mecanismos también conocidos como **síncrono** y **asíncrono**.

- **Envío con bloqueo.** El proceso que envía se bloquea hasta que el proceso receptor o el buzón de correo reciben el mensaje.
- **Envío sin bloqueo.** El proceso transmisor envía el mensaje y continúa operando.
- **Recepción con bloqueo.** El receptor se bloquea hasta que hay un mensaje disponible.
- **Recepción sin bloqueo.** El receptor extrae un mensaje válido o un mensaje nulo.

Son posibles diferentes combinaciones de las operaciones `send()` y `receive()`. Cuando ambas operaciones se realizan con bloqueo, tenemos lo que se denomina un **rendezvous** entre el transmisor y el receptor. La solución al problema del productor-consumidor es trivial cuando se usan instrucciones `send()` y `receive()` con bloqueo. El productor simplemente invoca la llamada `send()` con bloqueo y espera hasta que el mensaje se entrega al receptor o al buzón de correo. Por otro lado, cuando el consumidor invoca la llamada `receive()`, se bloquea hasta que hay un mensaje disponible.

Observe que los conceptos de síncrono y asíncrono se usan con frecuencia en los algoritmos de E/S en los sistemas operativos, como veremos a lo largo del texto.

3.4.2.3 Almacenamiento en búfer

Sea la comunicación directa o indirecta, los mensajes intercambiados por los procesos que se están comunicando residen en una cola temporal. Básicamente, tales colas se pueden implementar de tres maneras:

- **Capacidad cero.** La cola tiene una longitud máxima de cero; por tanto, no puede haber ningún mensaje esperando en el enlace. En este caso, el transmisor debe bloquearse hasta que el receptor reciba el mensaje.
- **Capacidad limitada.** La cola tiene una longitud finita n; por tanto, puede haber en ella n mensajes como máximo. Si la cola no está llena cuando se envía un mensaje, el mensaje se introduce en la cola (se copia el mensaje o se almacena un puntero al mismo), y el transmisor puede continuar la ejecución sin esperar. Sin embargo, la capacidad del enlace es finita. Si el enlace está lleno, el transmisor debe bloquearse hasta que haya espacio disponible en la cola.
- **Capacidad ilimitada.** La longitud de la cola es potencialmente infinita; por tanto, puede haber cualquier cantidad de mensajes esperando en ella. El transmisor nunca se bloquea.

En ocasiones, se dice que el caso de capacidad cero es un sistema de mensajes sin almacenamiento en búfer; los otros casos se conocen como sistemas con almacenamiento en búfer automático.

3.5 Ejemplos de sistemas IPC

En esta sección vamos a estudiar tres sistemas IPC diferentes. En primer lugar, analizaremos la API de POSIX para el modelo de memoria compartida, así como el modelo de paso de mensajes en el sistema operativo Mach. Concluiremos con Windows XP, que usa de una forma interesante el modelo de memoria compartida como mecanismo para proporcionar ciertos mecanismos de paso de mensajes.

3.5.1 Un ejemplo: memoria compartida en POSIX

Para los sistemas POSIX hay disponibles varios mecanismos IPC, incluyendo los de memoria compartida y de paso de mensajes. Veamos primero la API de POSIX para memoria compartida.

En primer lugar, un proceso tiene que crear un segmento de memoria compartida usando la llamada al sistema `shmget()`. `shmget()` se deriva de Shared Memory GET (obtención de datos a través de memoria compartida). El siguiente ejemplo ilustra el uso de `shmget()`.

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

El primer parámetro especifica la clave (o identificador) del segmento de memoria compartida. Si se define como IPC_PRIVATE, se crea un nuevo segmento de memoria compartida. El segundo parámetro especifica el tamaño (en bytes) del segmento. Por último, el tercer parámetro identifica el modo, que indica cómo se va a usar el segmento de memoria compartida: para leer, para escribir o para ambas operaciones. Al establecer el modo como S_IRUSR | S_IWUSR, estamos indicando que el propietario puede leer o escribir en el segmento de memoria compartida. Una llamada a `shmget()` que se ejecute con éxito devolverá un identificador entero para el segmento. Otros procesos que deseen utilizar esa región de la memoria compartida deberán especificar este identificador.

Los procesos que deseen acceder a un segmento de memoria compartida deben asociarlo a su espacio de direcciones usando la llamada al sistema `shmat()` [Shared Memory ATtach]. La llamada a `shmat()` espera también tres parámetros. El primero es el identificador entero del segmento de memoria compartida al que se va a conectar, el segundo es la ubicación de un puntero en memoria, que indica dónde se asociará la memoria compartida; si pasamos el valor NULL, el sistema operativo selecciona la ubicación en nombre del usuario. El tercer parámetro especifica un indicador que permite que la región de memoria compartida se conecte en modo de sólo lectura o sólo escritura. Pasando un parámetro de valor 0, permitimos tanto lecturas como escrituras en la memoria compartida.

El tercer parámetro especifica un indicador de modo. Si se define, el indicador de modo permite a la región de memoria compartida conectarse en modo de sólo lectura; si se define como 0, permite tanto lecturas como escrituras en dicha región. Para asociar una región de memoria compartida usando `shmat()`, podemos hacer como sigue:

```
shared_memory = (char *) shmat(id, NULL, 0);
```

Si se ejecuta correctamente, `shmat()` devuelve un puntero a la posición inicial de memoria a la que se ha asociado la región de memoria compartida.

Una vez que la región de memoria compartida se ha asociado al espacio de direcciones de un proceso, éste puede acceder a la memoria compartida como en un acceso de memoria normal, usando el puntero devuelto por `shmat()`. En este ejemplo, `shmat()` devuelve un puntero a una cadena de caracteres. Por tanto, podríamos escribir en la región de memoria compartida como sigue:

```
sprintf(shared_memory, "Escribir en memoria compartida");
```

Los otros procesos que compartan este segmento podrán ver las actualizaciones hechas en el segmento de memoria compartida.

Habitualmente, un proceso que usa un segmento de memoria compartida existente asocia primero la región de memoria compartida a su espacio de direcciones y luego accede (y posiblemente actualiza) dicha región. Cuando un proceso ya no necesita acceder al segmento de memoria compartida, desconecta el segmento de su espacio de direcciones. Para desconectar una región de memoria compartida, el proceso puede pasar el puntero de la región de memoria compartida a la llamada al sistema `shmdt()`, de la forma siguiente:

```
shmdt(shared_memory);
```

Por último, un segmento de memoria compartida puede eliminarse del sistema mediante la llamada al sistema `shmctl()`, a la cual se pasa el identificador del segmento compartido junto con el indicador IPC_RMID.

El programa mostrado en la Figura 3.16 ilustra la API de memoria compartida de POSIX explicada anteriormente. Este programa crea un segmento de memoria compartida de 4.096 bytes. Una vez que la región de memoria compartida se ha conectado, el proceso escribe el mensaje `¡Hola!` en la memoria compartida. Después presenta a la salida el contenido de la memoria actualizada, y desconecta y elimina la región de memoria compartida. Al final del capítulo se proporcionan más ejercicios que usan la API de memoria compartida de POSIX.

3.5.2 Un ejemplo: Mach

Como ejemplo de sistema operativo basado en mensajes, vamos a considerar a continuación el sistema operativo Mach, desarrollado en la Universidad Carnegie Mellon. En el Capítulo 2, hemos presentado Mach como parte del sistema operativo Mac OS X. El *kernel* de Match permite la creación y destrucción de múltiples tareas, que son similares a los procesos, pero tienen múltiples hebras de control. La mayor parte de las comunicaciones en Mach, incluyendo la mayoría de las llamadas al sistema y toda la comunicación inter-tareas, se realiza mediante *mensajes*. Los mensajes se envían y se reciben mediante buzones de correo, que en Mach se denominan *puertos*.

Incluso las llamadas al sistema se hacen mediante mensajes. Cuando se crea una tarea, también se crean dos buzones de correo especiales: el buzón de correo del *kernel* (Kernel) y el de notificaciones (Notify). El *kernel* utiliza el buzón Kernel para comunicarse con la tarea y envía las notificaciones de sucesos al puerto Notify. Sólo son necesarias tres llamadas al sistema para la transferencia de mensajes. La llamada `msg_send()` envía un mensaje a un buzón de correo. Un mensaje se recibe mediante `msg_receive()`. Finalmente, las llamadas a procedimientos remotos (RPC) se ejecutan mediante `msg_rpc()`, que envía un mensaje y espera a recibir como contestación exactamente un mensaje. De esta forma, las llamadas RPC modelan una llamada típica a procedimiento, pero pueden trabajar entre sistemas distintos (de ahí el calificativo de *remoto*).

La llamada al sistema `port_allocate()` crea un buzón de correo nuevo y asigna espacio para su cola de mensajes. El tamaño máximo de la cola de mensajes es, de manera predeterminada, de ocho mensajes. La tarea que crea el buzón es la propietaria de dicho buzón. El propietario también puede recibir mensajes del buzón de correo. Sólo una tarea cada vez puede poseer o recibir de un buzón de correo, aunque estos derechos pueden enviarse a otras tareas si se desea.

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* el identificador para el segmento de memoria compartida */
int segment_id;=
/* un puntero al segmento de memoria compartida */
char* shared_memory;
/* el tamaño (en bytes) del segmento de memoria compartida */
const int size = 4096;
   /* asignar un segmento de memoria compartida */
   segment_id = shmget (IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

   /* asociar el segmento de memoria compartida */
   shared_memory = (char *) shmat(segment_id, NULL, 0);

   /* escribir un mensaje en el segmento de memoria compartida */
   sprintf(shared_memory, "¡Hola!");

   /* enviar a la salida la cadena de caracteres de la memoria
   /* compartida */
   printf("*%s\n", shared_memory);

   /* desconectar el segmento de memoria compartida */
   shmdt (shared_memory);

   /* eliminar el segmento de memoria compartida */
   shmctl(segment_id, IPC_RMID, NULL);

   return 0;
}
```

Figura 3.16 Programa C que ilustra la API de memoria compartida de POSIX.

Inicialmente, el buzón de correo tiene una cola de mensajes vacía. A medida que llegan mensajes al buzón, éstos se copian en el mismo. Todos los mensajes tienen la misma prioridad. Mach garantiza que los múltiples mensajes de un mismo emisor se coloquen en la cola utilizando un algoritmo FIFO (first-in, first-out; primero en entrar, primero en salir), aunque el orden no se garantiza de forma absoluta. Por ejemplo, los mensajes procedentes de dos emisores distintos pueden ponerse en cola en cualquier orden.

Los mensajes en sí constan de una cabecera de longitud fija, seguida de unos datos de longitud variable. La cabecera indica la longitud del mensaje e incluye dos nombres de buzón de correo. Uno de ellos es el del buzón de correo al que se está enviando el mensaje. Habitualmente, la hebra emisora espera una respuesta, por lo que a la hebra receptora se le pasa el nombre del buzón del emisor; la hebra emisora puede usar ese buzón como una "dirección de retorno".

La parte variable de un mensaje es una lista de elementos de datos con tipo. Cada entrada de la lista tiene un tipo, un tamaño y un valor. El tipo de los objetos especificados en el mensaje es importante, ya que pueden enviarse en los mensajes objetos definidos por el sistema operativo (como, por ejemplo, derechos de propiedad o de acceso de recepción, estados de tareas y segmentos de memoria).

Las operaciones de envío y recepción son flexibles. Por ejemplo, cuando se envía un mensaje a un buzón de correo, éste puede estar lleno. Si no está lleno, el mensaje se copia en el buzón y la hebra emisora continúa. Si el buzón está lleno, la hebra emisora tiene cuatro opciones:

1. Esperar indefinidamente hasta que haya espacio en el buzón.

2. Esperar como máximo *n* milisegundos.
3. No esperar nada y volver inmediatamente.
4. Almacenar el mensaje temporalmente en caché. Puede proporcionarse al sistema operativo un mensaje para que lo guarde, incluso aunque el buzón al que se estaba enviando esté lleno. Cuando el mensaje pueda introducirse en el buzón, el sistema enviará un mensaje de vuelta al emisor; en un instante determinado, para una determinada hebra emisora, sólo puede haber un mensaje pendiente de este tipo dirigido a un buzón lleno,

La última opción está pensada para las tareas de servidor, como por ejemplo un controlador de impresora. Después de terminar una solicitud, tales tareas pueden necesitar enviar una única respuesta a la tarea que solicitó el servicio, pero también deben continuar con otras solicitudes de servicio, incluso aunque el buzón de respuesta de un cliente esté lleno.

La operación de recepción debe especificar el buzón o el conjunto de buzones desde el se van a recibir los mensajes. Un **conjunto de buzones de correo** es una colección de buzones declarados por la tarea, que pueden agruparse y tratarse como un único buzón de correo, en lo que a la tarea respecta. Las hebras de una tarea pueden recibir sólo de un buzón de correo o de un conjunto de buzones para el que la tarea haya recibido autorización de acceso. Una llamada al sistema `port_status()` devuelve el número de mensajes que hay en un determinado buzón. La operación de recepción puede intentar recibir de (1) cualquier buzón del conjunto de buzones o (2) un buzón de correo específico (nominado). Si no hay ningún mensaje esperando a ser recibido, la hebra de recepción puede esperar como máximo *n* milisegundos o no esperar nada.

El sistema Mach fue especialmente diseñado para sistemas distribuidos, los cuales se estudian en los Capítulos 16 a 18, pero Mach también es adecuado para sistemas de un solo procesador, como demuestra su inclusión en el sistema Mac OS X. El principal problema con los sistemas de mensajes ha sido generalmente el pobre rendimiento, debido a la doble copia de mensajes: el mensaje se copia primero del emisor al buzón de correo y luego desde el buzón al receptor. El sistema de mensajes de Mach intenta evitar las operaciones de doble copia usando técnicas de gestión de memoria virtual (Capítulo 9). En esencia, Mach asigna el espacio de direcciones que contiene el mensaje del emisor al espacio de direcciones del receptor; el propio mensaje nunca se copia realmente. Esta técnica de gestión de mensajes proporciona un mayor rendimiento, pero sólo funciona para mensajes intercambiados dentro del sistema. El sistema operativo Mach se estudia en un capítulo adicional disponible en el sitio web del libro.

3.5.3 Un ejemplo: Windows XP

El sistema operativo Windows XP es un ejemplo de un diseño moderno que emplea la modularidad para incrementar la funcionalidad y disminuir el tiempo necesario para implementar nuevas características. Windows XP proporciona soporte para varios entornos operativos, o *subsistemas*, con los que los programas de aplicación se comunican usando un mecanismo de paso de mensajes. Los programas de aplicación se pueden considerar clientes del servidor de subsistemas de Windows XP.

La facilidad de paso de mensajes en Windows XP se denomina **llamada a procedimiento local** (LPC, local procedure call). En Windows XP, la llamada LPC establece la comunicación entre dos procesos de la misma máquina. Es similar al mecanismo estándar RPC, cuyo uso está muy extendido, pero está optimizado para Windows XP y es específico del mismo. Como Mach, Windows XP usa un objeto puerto para establecer y mantener una conexión entre dos procesos. Cada cliente que llama a un subsistema necesita un canal de comunicación, que se proporciona mediante un objeto puerto y que nunca se hereda. Windows XP usa dos tipos de puertos: puertos de conexión y puertos de comunicación. Realmente son iguales, pero reciben nombres diferentes según cómo se utilicen. Los puertos de conexión se denominan *objetos* y son visibles para todos los procesos; proporcionan a las aplicaciones una forma de establecer los canales de comunicación (Capítulo 22). La comunicación funciona del modo siguiente:

- El cliente abre un descriptor del objeto puerto de conexión del subsistema.

- El cliente envía una solicitud de conexión.
- El servidor crea dos puertos de comunicación privados y devuelve el descriptor de uno de ellos al cliente.
- El cliente y el servidor usan el descriptor del puerto correspondiente para enviar mensajes o realizar retrollamadas y esperar las respuestas.

Windows XP usa dos tipos de técnicas de paso de mensajes a través del puerto que el cliente especifique al establecer el canal. La más sencilla, que se usa para mensajes pequeños, usa la cola de mensajes del puerto como almacenamiento intermedio y copia el mensaje de un proceso a otro. Con este método, se pueden enviar mensajes de hasta 256 bytes.

Si un cliente necesita enviar un mensaje más grande, pasa el mensaje a través de un **objeto sección**, que configura una región de memoria compartida. El cliente tiene que decidir, cuando configura el canal, si va a tener que enviar o no un mensaje largo. Si el cliente determina que va a enviar mensajes largos, pide que se cree un objeto sección. Del mismo modo, si el servidor decide que la respuesta va a ser larga, crea un objeto sección. Para que el objeto sección pueda utilizarse, se envía un mensaje corto que contenga un puntero e información sobre el tamaño del objeto sección. Este método es más complicado que el primero, pero evita la copia de datos. En ambos casos, puede emplearse un mecanismo de retrollamada si el cliente o el servidor no pueden responder inmediatamente a una solicitud. El mecanismo de retrollamada les permite hacer un tratamiento asíncrono de los mensajes. La estructura de las llamadas a procedimientos locales en Windows XP se muestra en la Figura 3.17.

Es importante observar que la facilidad LPC de Windows XP no forma parte de la API de Win32 y, por tanto, no es visible para el programador de aplicaciones. En su lugar, las aplicaciones que usan la API de Win32 invocan las llamadas a procedimiento remoto estándar. Cuando la llamada RPC se invoca sobre un proceso que resida en el mismo sistema, dicha llamada se gestiona indirectamente a través de una llamada a procedimiento local. Las llamadas a procedimiento local también se usan en algunas otras funciones que forman parte de la API de Win32.

3.6 Comunicación en los sistemas cliente-servidor

En la Sección 3.4 hemos descrito cómo pueden comunicarse los procesos utilizando las técnicas de memoria compartida y de paso de mensajes. Estas técnicas también pueden emplearse en los sistemas cliente-servidor (Sección 1.12.2) para establecer comunicaciones. En esta sección, exploraremos otras tres estrategias de comunicación en los sistemas cliente-servidor: *sockets*, llamadas a procedimientos remotos (RPC) e invocación de métodos remotos de Java (RMI, remote method invocation).

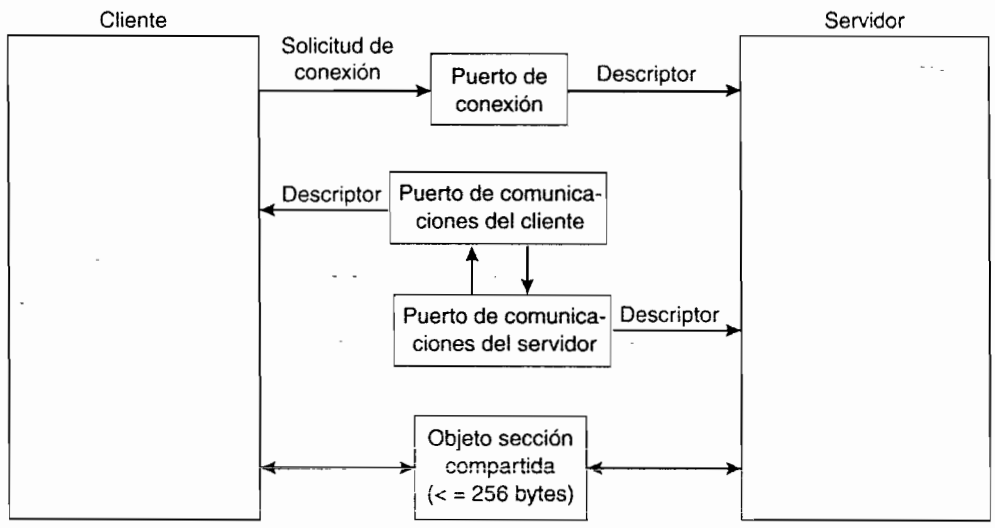

Figura 3.17 Llamadas a procedimiento locales en Windows XP.