En esta clase de problemas es necesario disponer de algún mecanismo de comunicación que permita a los procesos productor y consumidor intercambiar información. Ambos procesos además, deben sincronizar su acceso al mecanismo de comunicación para que la interacción entre ellos no sea problemática: cuando el mecanismo de comunicación se llene, el proceso productor se deberá quedar bloqueado hasta que haya hueco para seguir insertando elementos. A su vez, el proceso consumidor deberá quedarse bloqueado cuando el mecanismo de comunicación este vacío, ya que en este caso no podrá continuar su ejecución al no disponer de información a consumir. Por lo tanto, este tipo de problema requiere servicios para que los procesos puedan comunicarse y servicios para que se sincronicen a la hora de acceder al mecanismo de comunicación.

5.1.1.3. El problema de los lectores-escritores

En este problema existe un determinado objeto (ver Figura 5.2), que puede ser un archivo, un registro dentro de un archivo, etc., que va a ser utilizado y compartido por una serie de procesos concurrentes. Algunos de estos procesos sólo van a acceder al objeto sin modificarlo, mientras que otros van a acceder al objeto para modificar su contenido. Esta actualización implica leerlo, modificar su contenido y escribirlo. A los primeros procesos se les denomina *lectores*, y a los segundos se les denomina *escritores*. En este tipo de problemas existen una serie de restricciones que han de seguirse:

- Sólo se permite que un escritor tenga acceso al objeto al mismo tiempo. Mientras el escritor esté accediendo al objeto, ningún otro proceso lector ni escritor podrá acceder a él.
- Se permite, sin embargo, que múltiples lectores tengan acceso al objeto, ya que ellos nunca van a modificar el contenido del mismo.

En este tipo de problemas es necesario disponer de servicios de sincronización que permitan a los procesos lectores y escritores sincronizarse adecuadamente en el acceso al objeto.

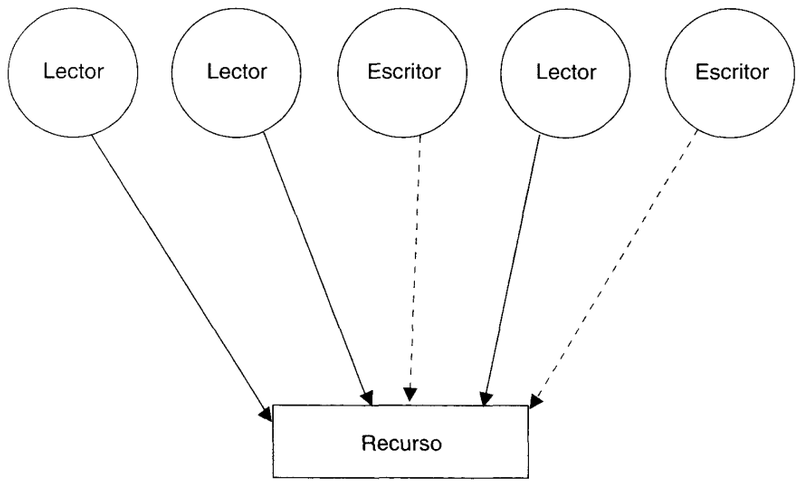

Figura 5.2. *Procesos lectores y escritores.*

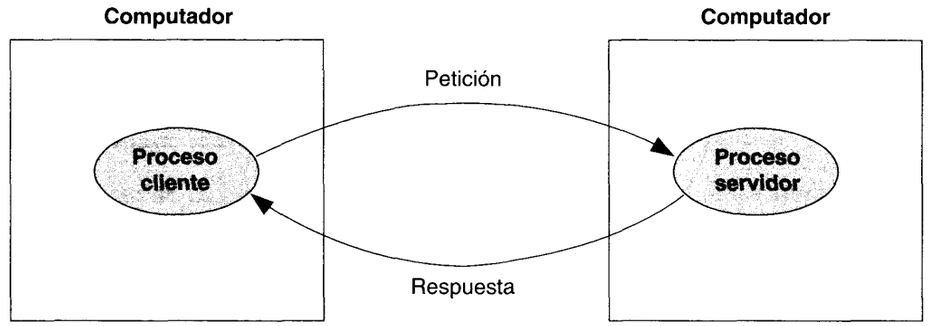

Figura 5.3. *Comunicación cliente-servidor.*

5.1.1.4. Comunicación cliente-servidor

En el modelo cliente-servidor, los procesos llamados servidores ofrecen una serie de servicios a otros procesos que se denominan clientes (ver Figura 5.3). El proceso servidor puede residir en la misma máquina que el cliente o en una distinta, en cuyo caso la comunicación deberá realizarse a través de una red de interconexión. Muchas aplicaciones y servicios de red como el correo electrónico y la transferencia de archivos se basan en este modelo.

En este tipo de aplicaciones es necesario que el sistema operativo ofrezca servicios que permitan comunicarse a los procesos cliente y servidor. Cuando los procesos ejecutan en la misma máquina, se pueden emplear técnicas basadas en memoria compartida o archivos. Sin embargo, este modelo de comunicación suele emplearse en aplicaciones que ejecutan en computadores que no comparten memoria y, por lo tanto, se usan técnicas basadas en paso de mensajes.

5.1.2. Mecanismos y servicios de comunicación

En esta sección se presentan los mecanismos y servicios que ofrece POSIX para la comunicación y sincronización de procesos, y que se utilizarán en la realización de las prácticas que se proponen en este capítulo.

5.1.2.1. Tuberías (pipes)

Una tubería es un mecanismo de comunicación y sincronización. Desde el punto de vista de su utilización, es como un pseudoarchivo mantenido por el sistema operativo. Conceptualmente, cada proceso ve la tubería como un conducto con dos extremos, uno de los cuales se utiliza para escribir o insertar datos y el otro para extraer o leer datos de la tubería. La escritura se realiza mediante el servicio que se utiliza para escribir datos en un archivo. De igual forma, la lectura se lleva a cabo mediante el servicio que se emplea para leer de un archivo.

El flujo de datos en la comunicación empleando tuberías es unidireccional y FIFO, esto quiere decir que los datos se extraen de la tubería (mediante la operación de lectura) en el mismo orden en el que se insertaron (mediante la operación de escritura). La Figura 5.4 representa dos procesos que se comunican de forma unidireccional utilizando una tubería.

Un pipe en POSIX no tiene nombre y, por lo tanto, sólo puede ser utilizado entre los procesos que lo hereden a través de la llamada `fork()`. A continuación se describen los servicios que permiten crear y acceder a los datos de un pipe.

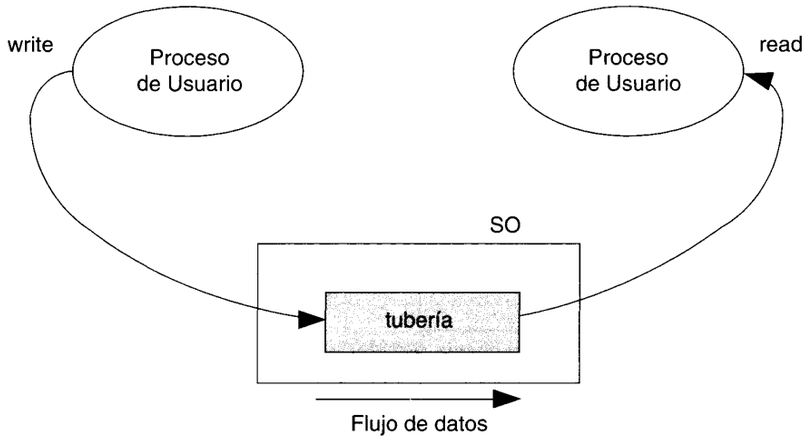

Figura 5.4. *Comunicación unidireccional utilizando una tubería.*

Creación de un pipe

El servicio que permite crear un pipe es el siguiente:

```
int pipe(int fildes[2]);
```

Esta llamada devuelve dos descriptores de archivos (véase la Figura 5.5) que se utilizan como identificadores:

- fildes[0], descriptor de archivo que se emplea para leer del pipe.
- fildes[1], descriptor de archivo que se utiliza para escribir en el pipe.

La llamada pipe devuelve 0 si fue bien y –1 en caso de error.

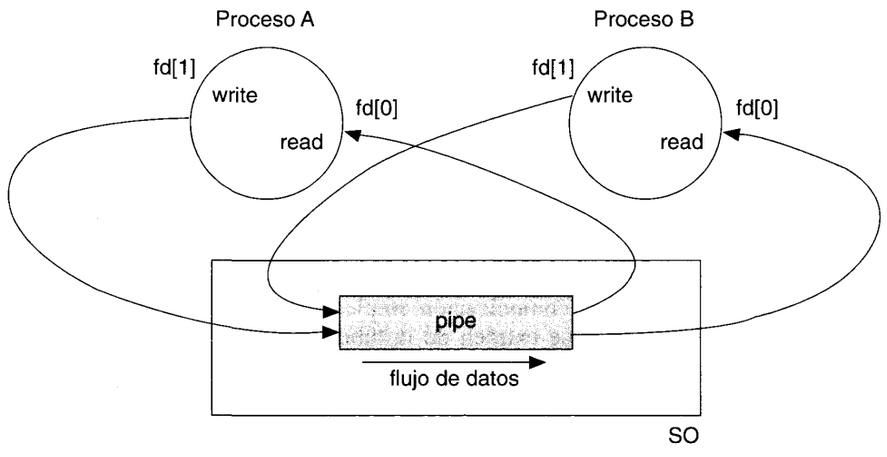

Figura 5.5. *Tuberías POSIX entre dos procesos.*

Cierre de un pipe

El cierre de cada uno de los descriptores que devuelve la llamada pipe se consigue mediante el servicio close que también se emplea para cerrar cualquier archivo. Su prototipo es:

```
int close(int fd);
```

El argumento de `close` indica el descriptor de archivo que se desea cerrar. La llamada devuelve 0 si se ejecutó con éxito. En caso de error devuelve –1.

Escritura en un pipe

El servicio para escribir datos en un pipe en POSIX es el siguiente:

```
int write(int fd, char *buffer, int n);
```

Este servicio también se emplea para escribir datos en un archivo. El primer argumento representa el descriptor de archivo que se emplea para escribir en un pipe. El segundo argumento especifica el buffer de usuario donde se encuentran los datos que se van a escribir al pipe. El último argumento indica el número de bytes a escribir. Los datos se escriben en el pipe en orden FIFO. La semántica de esta llamada es la siguiente:

- Si la tubería se encuentra llena o se llena durante la escritura, la operación bloquea al proceso escritor hasta que se pueda completar.
- Si no hay ningún proceso con la tubería abierta para lectura, la operación devuelve el correspondiente error. Este error se genera mediante el envío al proceso que intenta escribir de la señal SIGPIPE.
- Una operación de escritura sobre una tubería se realiza de forma **atómica**, es decir, si dos procesos intentan escribir de forma simultánea en una tubería, sólo uno de ellos lo hará, el otro se bloqueará hasta que finalice la primera escritura.

Lectura de un pipe

Para leer datos de un pipe se utiliza el siguiente servicio, también empleado para leer datos de un archivo.

```
int read(int fd, char *buffer, int n);
```

El primer argumento indica el descriptor de lectura del pipe. El segundo argumento especifica el buffer de usuario donde se van a situar los datos leídos del pipe. El último argumento indica el número de bytes que se desean leer del pipe. La llamada devuelve el número de bytes leídos. En caso de error, la llamada devuelve –1. Las operaciones de lectura siguen la siguiente semántica:

- Si la tubería está vacía, la llamada bloquea al proceso en la operación de lectura hasta que algún proceso escriba datos en la misma.
- Si la tubería almacena M bytes y se quieren leer n bytes, entonces:
  - Si $M \geq n$, la llamada devuelve n bytes y elimina de la tubería los datos solicitados.
  - Si $M < n$, la llamada devuelve M bytes y elimina los datos disponibles en la tubería.

- Si no hay escritores y la tubería está vacía, la operación devuelve fin de archivo (la llamada read devuelve cero). En este caso la operación no bloquea al proceso.
- Al igual que las escrituras, las operaciones de lectura sobre una tubería son atómicas. En general la atomicidad en las operaciones de lectura y escritura sobre una tubería se asegura siempre que el número de datos involucrados en las anteriores operaciones sea menor que el tamaño de la misma.

5.1.2.2. Semáforos

Un semáforo es un mecanismo de sincronización que se utiliza generalmente en sistemas con memoria compartida, bien sea un monoprocesador o un multiprocesador. Su uso en un multicomputador depende del sistema operativo en particular. Un semáforo es un objeto con un valor entero, al que se le puede asignar un valor inicial no negativo y al que sólo se puede acceder utilizando dos operaciones atómicas: `wait` y `signal` (también llamadas `down` o `up` respectivamente). Las definiciones de estas dos operaciones son las siguientes:

```
wait(s){
      s = s - 1;
      if (s < 0)
            Bloquear al proceso;
}

signal(s){
      s = s + 1;
      if (s <= 0)
            Desbloquear a un proceso bloqueado en la operación wait;
}
```

El número de procesos, que en un instante determinado se encuentran bloqueados en una operación `wait`, viene dado por el valor absoluto del semáforo si es negativo. Cuando un proceso ejecuta la operación `signal`, el valor del semáforo se incrementa. En el caso de que haya algún proceso bloqueado en una operación `wait` anterior, se desbloqueará a un solo proceso.

Las operaciones `wait` y `signal` son dos operaciones genéricas que deben particularizarse en cada sistema operativo. A continuación se presentan los servicios que ofrece el estándar POSIX para trabajar con semáforos.

En POSIX un semáforo se identifica mediante una variable del tipo `sem_t`. El estándar POSIX define dos tipos de semáforos:

- **Semáforos sin nombre.** Permiten sincronizar a los procesos ligeros que ejecutan dentro de un mismo proceso, o a los procesos que lo heredan a través de la llamada fork.
- **Semáforos con nombre.** En este caso el semáforo lleva asociado un nombre que sigue la convención de nombrado que se emplea para archivos. Con este tipo de semáforos se pueden sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada fork.

Creación de un semáforo sin nombre

Todos los semáforos en POSIX deben iniciarse antes de su uso. La función `sem_init` permite iniciar un semáforo sin nombre. El prototipo de este servicio es el siguiente:

```
int sem_init(sem_t *sem, int shared, int val);
```


Con este servicio se crea y se asigna un valor inicial a un semáforo sin nombre. El primer argumento identifica la variable de tipo semáforo que se quiere utilizar. El segundo argumento indica si el semáforo se puede utilizar para sincronizar procesos ligeros o cualquier otro tipo de proceso. Si `shared` es 0, el semáforo sólo puede utilizarse entre los procesos ligeros creados dentro del proceso que inicia el semáforo. Si `shared` es distinto de 0, entonces se puede utilizar para sincronizar procesos que lo hereden por medio de la llamada `fork`. El tercer argumento representa el valor que se asigna inicialmente al semáforo.

Destrucción de un semáforo sin nombre

Con este servicio se destruye un semáforo sin nombre previamente creado con la llamada `sem_init`. Su prototipo es el siguiente:

```
int sem_destroy(sem_t *sem)
```

Creación y apertura de un semáforo con nombre

El servicio `sem_open` permite crear o abrir un semáforo con nombre. La función que se utiliza para invocar este servicio admite dos modalidades según se utilice para crear el semáforo o simplemente abrir uno existente. Estas modalidades son las siguientes:

```
sem_t *sem_open(char *name, int flag, mode_t mode, int val);
sem_t *sem_open(char *name, int flag);
```

Un semáforo con nombre posee un nombre, un dueño y derechos de acceso similares a los de un archivo. El nombre de un semáforo es una cadena de caracteres que sigue la convención de nombrado de un archivo. La función `sem_open` establece una conexión entre un semáforo con nombre y una variable de tipo semáforo.

El valor del segundo argumento determina si la función `sem_open` accede a un semáforo previamente creado o si crea un nuevo. Un valor 0 en `flag` indica que se quiere utilizar un semáforo que ya ha sido creado, en este caso no es necesario los dos últimos parámetros de la función `sem_open`. Si `flag` tiene un valor `O_CREAT`, requiere los dos últimos argumentos de la función. El tercer parámetro especifica los permisos del semáforo que se va a crear, de la misma forma que ocurre en la llamada `open` para archivos. El cuarto parámetro especifica el valor inicial del semáforo.

POSIX no requiere que los semáforos con nombre se correspondan con entradas de directorio en el sistema de archivos, aunque sí pueden aparecer.

Cierre de un semáforo con nombre

Cierra un semáforo con nombre, rompiendo la asociación que tenía un proceso con un semáforo. El prototipo de la función es:

```
int sem_close(sem_t *sem);
```

Borrado de un semáforo con nombre

Elimina del sistema un semáforo con nombre. Esta llamada pospone la destrucción del semáforo hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función `sem_close`. El prototipo de este servicio es:

```
int sem_unlink(char *name);
```

Operación wait

La operación wait en POSIX se consigue con el siguiente servicio:

```
int sem_wait(sem_t *sem);
```

Operación signal

Este servicio se corresponde con la operación signal sobre un semáforo. El prototipo de este servicio es:

```
int sem_post(sem_t *sem);
```

Todas las funciones que se han descrito devuelven un valor 0 si la función se ha ejecutado con éxito ó –1 en caso de error.

5.1.2.3. Mutex y variables condicionales

Los mutex y las variables condicionales son mecanismos especialmente concebidos para la sincronización de procesos ligeros. Un **mutex** es el mecanismo de sincronización de procesos ligeros más sencillo y eficiente. Los mutex se emplean para obtener acceso exclusivo a recursos compartidos y para asegurar la exclusión mutua sobre secciones críticas.

Sobre un mutex se pueden realizar dos operaciones atómicas básicas:

- **lock:** intenta bloquear el mutex. Si el mutex ya está bloqueado por otro proceso, el proceso que realiza la operación se bloquea. En caso contrario se bloquea el mutex sin bloquear al proceso.
- **unlock:** desbloquea el mutex. Si existen procesos bloqueados en él, se desbloqueará a uno de ellos que será el nuevo proceso que adquiera el mutex. La operación unlock sobre un mutex debe ejecutarla el proceso ligero que adquirió con anterioridad el mutex mediante la operación lock. Esto es diferente a lo que ocurre con las operaciones wait y signal sobre un semáforo.

El siguiente segmento de pseudocódigo utiliza un mutex para proteger el acceso a una sección crítica.

```
lock(m);  /* solicita la entrada en la sección crítica */
< sección crítica >
unlock(m); /* salida de la sección crítica */
```

En la Figura 5.6 se representa de forma gráfica una situación en la que dos procesos ligeros intentan acceder de forma simultánea a ejecutar código de una sección crítica utilizando un mutex para protegerla.

Dado que las operaciones `lock` y `unlock` son atómicas, sólo un proceso conseguirá bloquear el mutex y podrá continuar su ejecución dentro de la sección crítica. El segundo proceso se bloqueará hasta que el primero libere el mutex mediante la operación `unlock`.

Una **variable condicional** es una variable de sincronización asociada a un mutex que se utiliza para bloquear a un proceso hasta que ocurra algún suceso. Las variables condicionales tienen dos operaciones atómicas para esperar y señalizar:

- **c_wait:** bloquea al proceso que ejecuta la llamada y le expulsa del mutex dentro del cual se ejecuta y al que está asociada la variable condicional, permitiendo que algún otro proceso adquiera el mutex. El bloqueo del proceso y la liberación del mutex se realiza de forma atómica.
- **c_signal:** desbloquea a uno o varios procesos suspendidos en la variable condicional. El proceso que se despierta compite de nuevo por el mutex.

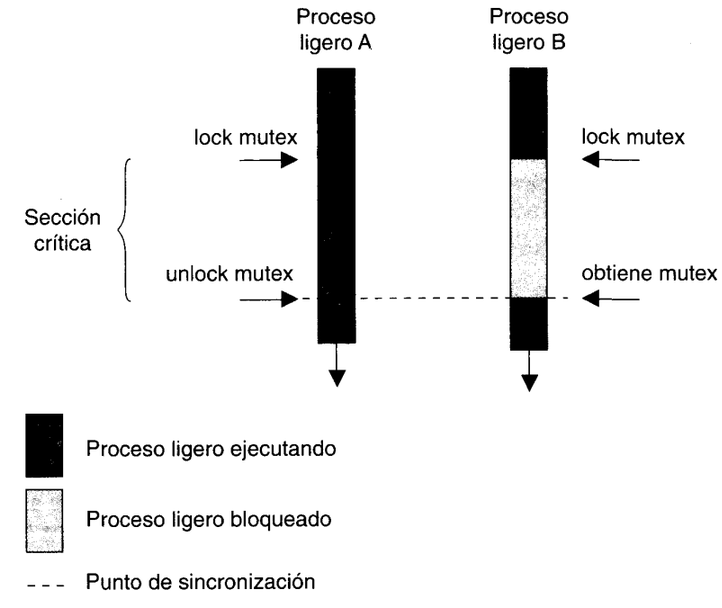

Figura 5.6. *Ejemplo de mutex en una sección crítica.*

A continuación se va a describir una situación típica en la que se utilizan los mutex y las variables condicionales de forma conjunta. Supóngase que una serie de procesos compiten por el acceso a una sección crítica. En este caso es necesario un mutex para proteger la ejecución de dicha sección crítica. Una vez dentro de la sección crítica puede ocurrir que un proceso no pueda continuar su ejecución dentro de la misma, debido a que no se cumple una determinada condición, por ejemplo, se quieren insertar elementos en un buffer común y éste se encuentra lleno. En esta situación el proceso debe bloquearse puesto que no puede continuar su ejecución. Además debe liberar el mutex para permitir que otro proceso entre en la sección crítica y pueda modificar la situación que bloqueó al proceso, en este caso eliminar un elemento del buffer común para hacer hueco.

Para conseguir este funcionamiento es necesario utilizar una o más variables compartidas que se utilizarán como predicado lógico y que el proceso consultará para decidir su bloqueo o no. El fragmento de código que se debe emplear en este caso es el siguiente:

```
lock(m);
/* código de la sección crítica */
while (condición == FALSE)
      c_wait(c, m);
/* resto de la sección crítica */
unlock(m);
```

En el fragmento anterior `m` es el mutex que se utiliza para proteger el acceso a la sección crítica y `c` la variable condicional que se emplea para bloquear el proceso y abandonar la sección crítica. Cuando el proceso que está ejecutando dentro de la sección evalúa la condición y esta es falsa, se bloquea mediante la operación `c_wait` y libera el mutex permitiendo que otro proceso entre en ella.

El proceso bloqueado permanecerá en esta situación hasta que algún otro proceso modifique alguna de las variables compartidas que le permitan continuar. El fragmento de código que debe ejecutar este otro proceso debe seguir el modelo siguiente:

```
lock(m);
/* código de la sección crítica */
/* se modifica la condición y esta se hace TRUE */
condición = TRUE;
c_signal(c);
unlock(m);
```

En este caso el proceso que hace cierta la condición ejecuta la operación `c_signal` sobre la variable condicional despertando a un proceso bloqueado en dicha variable. Cuando el proceso ligero que espera en una variable condicional se desbloquea, vuelve a competir por el mutex. Una vez adquirido de nuevo el mutex debe comprobar si la situación que le despertó y que le permitía continuar su ejecución sigue cumpliéndose, de ahí la necesidad de emplear una estructura de control de tipo `while`. Es necesario volver a evaluar la condición ya que entre el momento en el que la condición se hizo cierta y el instante en el que comienza a ejecutar de nuevo el proceso bloqueado en la variable condicional, puede haber ejecutado otro proceso que, a su vez, puede haber hecho falsa la condición. En la Figura 5.7 se representa de forma gráfica el uso de mutex y variables condicionales entre dos procesos tal y como se ha descrito anteriormente.

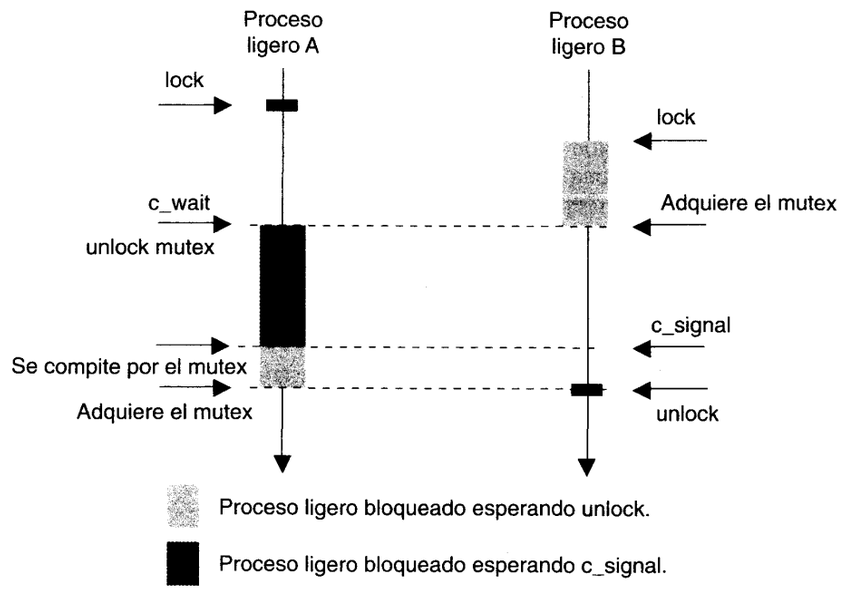

Figura 5.7. *Empleo de mutex y variables condicionales.*

A continuación se describen los servicios POSIX que permiten utilizar mutex y variables condicionales. Para utilizar un mutex un programa debe declarar una variable de tipo `pthread_mutex_t` (definido en el archivo de cabecera `pthread.h`) e iniciarla antes de utilizarla.

Iniciar un mutex

Esta función permite iniciar una variable de tipo mutex. Su prototipo es el siguiente:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
pthread_mutexattr_t *attr);
```

El segundo argumento especifica los atributos con los que se crea el mutex inicialmente, en caso de que este segundo argumento sea `NULL`, se tomarán los atributos por defecto.

Destruir un mutex

Permite destruir un objeto de tipo mutex. El prototipo de la función que permite invocar este servicio es:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Operación `lock`

Este servicio se corresponde con la operación `lock` descrita en la sección anterior. Esta función intenta obtener el mutex. Si el mutex ya se encuentra adquirido por otro proceso el proceso ligero que ejecuta la llamada se bloquea. Su prototipo es:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Operación `unlock`

Este servicio se corresponde con la operación `unlock` y permite al proceso ligero que la ejecuta liberar el mutex. El prototipo es:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Iniciar una variable condicional

Para emplear en un programa una variable condicional es necesario declarar una variable de tipo `pthread_cond_t` e iniciarla antes de usarla mediante el servicio `pthread_cond_init` cuyo prototipo se muestra a continuación:

```
int pthread_cond_init(pthread_cond_t *cond,
pthread_condattr_t *attr);
```

Esta función inicia una variable de tipo condicional. El segundo argumento especifica los atributos con los que se crea inicialmente la variable condicional. Si el segundo argumento es `NULL`, la variable condicional toma los atributos por defecto.

Destruir una variable condicional

Permite destruir una variable de tipo condicional. Su prototipo es:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Operación `c_wait` sobre una variable condicional

Este servicio se corresponde con la operación `c_wait` sobre una variable condicional. Su prototipo es:

```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex_t_ *mutex);
```

Esta función suspende al proceso ligero hasta que otro proceso ejecute una operación `c_signal` sobre la variable condicional pasada como primer argumento. De forma atómica se libera el mutex pasado como segundo argumento. Cuando el proceso se despierte volverá a competir por el mutex.

Operación `c_signal` sobre una variable condicional

Este servicio se corresponde con la operación `c_signal` sobre una variable condicional. Su prototipo es:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Se desbloquea a un proceso suspendido en la variable condicional pasada como argumento a esta función. Esta función no tiene efecto si no hay ningún proceso ligero esperando sobre la variable condicional. Para desbloquear a todos los procesos ligeros suspendidos en una variable condicional se emplea el servicio:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

5.1.2.4. Paso de mensajes

Todos los mecanismos vistos hasta el momento necesitan que los procesos que quieren intervenir en la comunicación o quieren sincronizarse ejecuten en la misma máquina. Cuando se quiere comunicar y sincronizar procesos que ejecutan en máquinas distintas es necesario recurrir al *paso de mensajes*. En este tipo de comunicación los procesos intercambian *mensajes* entre ellos. Es obvio que este esquema también puede emplearse para comunicar y sincronizar procesos que ejecutan en la misma máquina, en este caso los mensajes son locales a la máquina donde ejecutan los procesos.

Utilizando paso de mensajes como mecanismo de comunicación entre procesos, no es necesario recurrir a variables compartidas, únicamente debe existir un *enlace de comunicación* entre ellos. Los procesos se comunican mediante dos operaciones básicas:

- **`send`**`(destino, mensaje)`, envía un mensaje al proceso destino.
- **`receive`**`(origen, mensaje)`, recibe un mensaje del proceso origen.

De acuerdo con estas dos operaciones, las tuberías se pueden considerar en cierta medida como un mecanismo de comunicación basado en paso de mensajes. Los procesos pueden enviar un mensaje a otro proceso por medio de una operación de escritura y puede recibir mensajes de otros mediante una operación de lectura. En este caso el enlace que se utiliza para comunicar a los procesos es la propia tubería.

Existen múltiples implementaciones de sistemas con paso de mensajes. A continuación se describen algunos aspectos de diseño relativos a este tipo de sistemas.

Tamaño del mensaje

Los mensajes que envía un proceso a otro pueden ser de tamaño fijo o tamaño variable. En caso de mensajes de longitud fija la implementación del sistema de paso de mensajes es más sencilla, sin embargo, dificulta la tarea del programador ya que puede obligar a éste a descomponer los mensajes grandes en mensajes de longitud fija más pequeños.

Flujo de datos

De acuerdo al flujo de datos la comunicación puede ser **unidireccional** o **bidireccional**. Un enlace es unidireccional cuando cada proceso conectado a él únicamente puede enviar o recibir mensajes, pero no ambas cosas. Si cada proceso puede enviar o recibir mensajes entonces el paso de mensajes es bidireccional.

Nombrado

Los procesos que utilizan mensajes para comunicarse o sincronizarse deben tener alguna forma de referirse unos a otros. En este sentido la comunicación puede ser directa o indirecta.

La comunicación es **directa** cuando cada proceso que desea enviar o recibir un mensaje de otro debe nombrar de forma explícita al receptor o emisor del mensaje. En este esquema de comunicación, las operaciones básicas `send` y `receive` se definen de la siguiente manera:

- **`send`**`(P, mensaje)`, envía un mensaje al proceso P.
- **`receive`**`(Q, mensaje)`, espera la recepción de un mensaje por parte del proceso Q.

Existen modalidades de paso de mensajes con comunicación directa que permiten especificar al receptor la posibilidad de recibir un mensaje de cualquier proceso. En este caso la operación receive se define de la siguiente forma:

```
receive(ANY, mensaje);
```

La comunicación es **indirecta** cuando los mensajes no se envían directamente del emisor al receptor, sino a unas estructuras de datos que se denominan *colas de mensajes* o *puertos*. Una cola de mensajes es una estructura a la que los procesos pueden enviar mensajes y de la que se pueden extraer mensajes. Cuando dos procesos quieren comunicarse entre ellos, el emisor sitúa el mensaje en la cola y el receptor lo extrae de ella. Sobre una cola de mensajes puede haber múltiples emisores y receptores.

Un puerto es una estructura similar a una cola de mensajes, sin embargo, un puerto se encuentra asociado a un proceso y por tanto únicamente puede recibir de un puerto un proceso.

En este caso, cuando dos procesos quieren comunicarse entre sí, el receptor crea un puerto y el emisor envía mensajes al puerto del receptor.

Utilizando comunicación indirecta las operaciones send y receive toman la siguiente forma:

- **send**(Q, mensaje), envía un mensaje a la cola o al puerto Q.
- **receive**(Q, mensaje), recibe un mensaje de la cola o del puerto Q.

Cualquiera que sea el método utilizado, el paso de mensajes siempre se realiza en exclusión mutua. Si dos procesos ejecutan de forma simultánea una operación send, los mensajes no se entrelazan, primero se envía uno y, a continuación, el otro. De igual forma, si dos procesos desean recibir un mensaje de una cola, sólo se entregará el mensaje a uno de ellos.

Sincronización

La comunicación entre dos procesos es síncrona cuando los dos procesos han de ejecutar los servicios de comunicación al mismo tiempo, es decir, el emisor debe estar ejecutando la operación send y el receptor ha de estar ejecutando la operación `receive`. La comunicación es asíncrona en caso contrario.

En general, son tres las combinaciones más habituales que implementan los distintos tipos de paso de mensajes.

- **Envío y recepción bloqueante.** En este caso tanto el emisor como el receptor se bloquean hasta que tenga lugar la entrega del mensaje. Esta es una técnica de paso de mensajes totalmente síncrona que se conoce como *cita*.
- **Envío no bloqueante y recepción bloqueante.** Esta es la combinación generalmente más utilizada. El emisor no se bloquea hasta que tenga lugar la recepción y, por lo tanto, puede continuar su ejecución. El proceso que espera el mensaje, sin embargo, se bloquea hasta que le llega.
- **Envío y recepción no bloqueante.** Se corresponde con una comunicación totalmente asíncrona en la que nadie debe esperar. En este tipo de comunicación es necesario disponer de servicios que permitan al receptor saber si se ha recibido un mensaje.

Almacenamiento

Este aspecto hace referencia a la capacidad del enlace de comunicaciones. El enlace y, por tanto, el paso de mensajes pueden:

- **No tener capacidad** (sin almacenamiento) para almacenar mensajes. En este caso el mecanismo utilizado como enlace de comunicación no puede almacenar ningún mensaje y por lo tanto la comunicación entre los procesos emisor y receptor debe ser síncrona, es decir, el emisor sólo puede continuar cuando el receptor haya recogido el mensaje.
- **Tener capacidad** (con almacenamiento) para almacenar mensajes, en este caso la cola de mensajes o el puerto al que se envían los mensajes pueden tener un cierto tamaño para almacenar mensajes a la espera de su recepción. Si la cola no está llena al enviar un mensaje, se guarda en ella y el emisor puede continuar su ejecución sin necesidad de esperar. Sin embargo, si la cola ya está llena, el emisor deberá bloquearse hasta que haya espacio disponible en la cola para insertar el mensaje.

5.2. EJERCICIOS RESUELTOS

Ejercicio 5.1

¿Qué es falso en relación a las tuberías (pipes)?

A.– Si la tubería está vacía, el lector se queda bloqueado hasta que algún escritor escriba en la misma.
B.– Las operaciones de lectura pueden tener tamaños distintos a las operaciones de escritura.
C.– Dos procesos que quieren comunicarse ejecutan ambos la llamada `pipe`.
D.– El escritor puede escribir en la tubería aunque el lector no haya ejecutado una lectura del mismo.

Solución

La respuesta falsa es la C. Para que dos procesos puedan comunicarse a través de una tubería, es necesario que ambos la compartan. La única forma de compartir la tubería es que uno de ellos la cree y el otro la herede a través de la llamada al sistema `fork()`. Si los dos ejecutarán la llamada `pipe()`, se estarían creando dos tuberías distintas y la comunicación entre los dos procesos sería imposible.

Ejercicio 5.2

¿Qué es cierto acerca de los mecanismos de sincronización de procesos?

A.– Cualquier mecanismo es válido sobre cualquier tipo de plataforma.
B.– El paso de mensajes no se puede utilizar para comunicar procesos que ejecutan en un computador con una sola CPU.
C.– La espera activa es el mecanismo más ineficiente en el uso de la CPU.
D.– Con semáforos nunca se puede dar un interbloqueo.

Solución

La respuesta A es falsa, ya que no todos los mecanismos son válidos para todas las plataformas. Así, por ejemplo, no se puede utilizar memoria compartida en un multicomputador formado por varias CPU cada una con su memoria independiente. La respuesta B también es falsa puesto que el mecanismo de paso de mensajes es válido tanto para comunicar procesos que ejecutan en la misma máquina como para comunicar procesos que ejecutan en máquinas distintas conectadas a través de una red de interconexión. La respuesta D es falsa, ya que cuando se emplean semáforos se puede llegar a situaciones de interbloqueo. Por ejemplo, considere el siguiente escenario:

Proceso P1	**Proceso P2**
.	.
.	.
`wait(S1);`	`wait(S2);`
`signal(S2);`	`signal(S1);`
.	.