Comunicación y sincronización

José Ramón Herrero Zaragoza
Enric Morancho Llena
Dolors Royo Vallés

PID_00214803

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este módulo didáctico estudiamos las posibilidades que ofrece el sistema operativo para que dos o más procesos o flujos de ejecución puedan cooperar. Para hacerlo, el SO facilitará mecanismos de comunicación y de sincronización. Estos mecanismos suelen recibir el nombre de IPC (*inter-process communication*).

Estudiaremos dos escenarios: el caso de que los procesos que cooperan compartan recursos (una parte del espacio lógico o dispositivos) y el caso de que no compartan (o no quieran compartir) el espacio lógico. En cada escenario, veremos el repertorio típico de llamadas al sistema que el SO debe ofrecer, así como las problemáticas que pueden aparecer. En especial, introduciremos la problemática de los *deadlocks* (abrazos mortales o interbloqueos), que pueden aparecer especialmente entre procesos cooperativos que comparten memoria o dispositivos.

Objetivos

En los contenidos didácticos de este módulo encontraréis las herramientas básicas para alcanzar los objetivos siguientes:

1. Ser conscientes de la necesidad de sincronizar y comunicar los procesos que cooperan de modo concurrente para cumplir una tarea común.

2. Saber que la compartición de recursos lógicos del sistema (los dispositivos, el código, las variables compartidas, etc.) da lugar a la necesidad de sincronizar los procesos que los comparten para garantizar que el acceso de éstos a los recursos se efectúa correctamente (en exclusión mutua). Los procesos también comparten los recursos del sistema computador a nivel físico, pero esta compartición es gestionada por el sistema operativo y es transparente a los procesos. En este módulo, siempre nos referiremos a la compartición de recursos lógicos.

3. Conocer las ventajas y desventajas de los diferentes mecanismos que ofrece un sistema operativo para garantizar el acceso en exclusión mutua a los recursos lógicos compartidos.

4. Saber que todas las herramientas que sirven para asegurar el acceso exclusivo a los recursos del sistema son difíciles de utilizar y que se debe tener mucho cuidado para no producir situaciones inconsistentes o bloqueadoras cuando se trabaja con ellas.

5. Conocer las características generales de un mecanismo de comunicación basado en el paso de mensajes.

6. Saber utilizar las herramientas proporcionadas por algún sistema operativo concreto para comunicar y sincronizar procesos.

1. Comunicación y sincronización

En sistemas concurrentes, los procesos cooperan en el cálculo y la realización de tareas variadas y comparten recursos lógicos[1]. En estos sistemas, los procesos han de sincronizar su acceso a los objetos compartidos y han de poder intercambiar información entre ellos.

(1)Los dispositivos, el código y las variables son algunos de los recursos que los procesos comparten.

En general, se considera que hay dos maneras de establecer una cooperación entre los procesos:

1) La **sincronización**: este procedimiento permite el acceso concurrente de los procesos a objetos del sistema que requiere un acceso secuencial y asegura la integridad y la consistencia del sistema.

La sincronización entre procesos en sistemas concurrentes es imprescindible para que el sistema se mantenga coherente y, además, es necesaria a la hora de acceder a los recursos compartidos tanto en las estructuras de datos como en los dispositivos. Como el sistema operativo no sabe cuáles son las intenciones de los procesos, todos aquellos procesos que quieran compartir algún objeto dentro del sistema deben ponerse de acuerdo entre ellos para que el acceso que se ha de llevar a cabo sea coherente y no tenga ningún tipo de error. Para poder establecer los protocolos de acceso, el sistema y algunos lenguajes de programación ofrecen herramientas de sincronización entre los procesos.

Ejemplos

Los siguientes son ejemplos claros de objetos del sistema a los que acceden de manera concurrente los procesos:

- Las variables compartidas por todos los procesos dentro del sistema.
- Los dispositivos de naturaleza no compartida, como la impresora.

La utilización de protocolos de sincronización adecuados a los procesos que utilizan recursos comunes puede ser complicada y puede tener consecuencias bastante negativas para el sistema.

En este módulo didáctico presentamos con detalle los principales problemas que plantea la sincronización y describimos los semáforos como mecanismo de sincronización que la mayoría de los sistemas operativos suele ofrecer.

2) La **comunicación**: en procesos concurrentes, normalmente hay que intercambiar resultados parciales, se debe enviar información del estado de los procesos o intercambiar información en general. Los diferentes sistemas operativos han propuesto varios mecanismos que permiten comunicar ciertas cantidades de información entre los procesos.

Algunos de estos mecanismos son los siguientes:

- La **memoria compartida**, que permite comunicar procesos mediante el uso de variables compartidas.

- El **paso de mensajes**, un mecanismo relativamente sencillo que integra tareas de sincronización y comunicación entre procesos en sistemas centralizados y sistemas distribuidos.

- Las **señales**, interrupciones de software que pueden recibir los procesos para indicar que se ha producido un determinado éxito; por ejemplo, que ha expirado un temporizador, que se ha producido una bajada de tensión, que se ha generado una salida de rango en operaciones aritméticas (*overflow*), etc.

A continuación, veremos cómo realizar estas operaciones en dos escenarios: cuando los procesos comparten algún recurso lógico (por ejemplo, un rango de direcciones del espacio lógico) y cuando los procesos no comparten el espacio lógico.

2. Primer escenario: compartición de recursos

En este escenario, consideraremos diferentes procesos (o flujos de ejecución) que comparten algún recurso lógico. Por ejemplo, un rango de direcciones lógicas de memoria (en el que se almacena alguna estructura de datos compartida) o un dispositivo de entrada/salida. La utilización de variables compartidas es una manera bastante habitual de comunicación que los procesos utilizan dentro del sistema. En este apartado veremos los problemas que surgen cuando distintos procesos comparten un objeto lógico del sistema. Introduciremos los semáforos, una herramienta que nos puede ayudar a solucionar este problema, y veremos la interfaz de semáforos ofrecida por un sistema operativo.

2.1. La sincronización de procesos

2.1.1. ¿Por qué es necesaria la sincronización?

Veamos por qué es necesaria la sincronización de los procesos con un caso concreto. Un ejemplo claro de variable compartida[2] es el caso de una variable contador, a la que denominaremos `trabajo_pendiente`, que se modificada por diferentes procesos (usuario y gestor); tenemos que:

(2)Suponemos que los procesos pueden compartir memoria.

a) Los diferentes procesos usuario que se están ejecutando de modo concurrente en el sistema la actualizan (la incrementan) cada vez que se envía un trabajo[3] a la impresora. Así, esta variable compartida indica cuántos trabajos pendientes de impresión hay en el sistema.

(3)Los trabajos que los procesos usuario envían a la impresora son ficheros.

b) El proceso gestor de la impresora envía los trabajos a imprimir y actualiza el valor del contador decrementándolo cada vez que finaliza la impresión de un trabajo.

Si la actualización de esta variable contador se efectúa sin ningún tipo de restricción, se pueden generar inconsistencias, es decir, resultados erróneos. En el caso de nuestro ejemplo, estos resultados erróneos podrían repercutir en el proceso de impresión, de manera que se podría dejar de imprimir algún trabajo o intentar imprimir algún trabajo que no existe.

Veamos cómo se pueden generar estas inconsistencias analizando nuestro ejemplo. Como ya hemos indicado antes, un **proceso usuario** a la hora de enviar un trabajo a imprimir incrementa la variable `trabajo_pendiente` (figura 1).

El **proceso gestor de la impresora** cada vez que se acaba de imprimir un trabajo decrementa la variable `trabajo_pendiente` (figura 2).

```
...
trabajo_pendiente = trabajo_pendiente + 1
...
```

Figura 1. Fragmento de código del proceso de usuario

```
...
trabajo_pendiente = trabajo_pendiente - 1
...
```

Figura 2. Fragmento de código del proceso gestor

Así pues, la variable compartida `trabajo_pendiente` indica cuántos trabajos están pendientes de ser enviados a la impresora.

El principal problema que se plantea con el código que acabamos de presentar, y en general con el código programado en cualquier lenguaje de alto nivel, es que durante la compilación del programa una instrucción de alto nivel se traduce en una o más instrucciones de lenguaje máquina.

Por ejemplo, en una máquina RISC (*reduced instruction set computer*) el código generado una vez se ha compilado el código de nuestro ejemplo podría ser el que indicamos a continuación: en el caso del **código del proceso usuario** (figura 3) y en el caso del **código del proceso gestor de la impresora** (figura 4).

```
LOAD R0,trabajo_pendiente  ;Copiar el contenido de trabajo_pendiente en el registro R0
INC R0                     ;Incrementar R0
STORE trabajo_pendiente,R0 ;Almacenar R0 en trabajo_pendiente
```

Figura 3. Código ensamblador correspondiente al fragmento de código del proceso usuario

```
LOAD R1,trabajo_pendiente  ;Copiar el contenido de trabajo_pendiente en el registro R1
DEC R1                     ;Decrementar R1
STORE trabajo_pendiente,R1 ;Almacenar R1 en trabajo_pendiente
```

Figura 4. Código ensamblador correspondiente al fragmento de código del proceso gestor

Los detalles de la codificación no son importantes, lo que nos interesa advertir aquí es que durante la ejecución de la instrucción de alto nivel para incrementar/decrementar la variable `trabajo_pendiente` se realizan copias temporales de su contenido en registros del sistema[4]. Estas copias locales, como las que se llevan a cabo en los registros `R0` y `R1`, pueden ser inconsistentes con el valor de la variable de la memoria en un determinado instante de la ejecución del proceso. Esta inconsistencia es la que puede llevar a las dos situaciones erróneas mencionadas:

(4)Los registros del sistema son registros de hardware y no han de ser necesariamente diferentes para cada copia temporal.

1) No impresión de un trabajo

Consideremos, por ejemplo, que la variable `trabajo_pendiente` tiene un valor inicial igual a 3 cuando el gestor de la impresora ejecuta las instrucciones en lenguaje máquina que se han especificado anteriormente de la manera siguiente (figura 5):

a) En primer lugar, hace una copia del valor de la variable en el registro `R1`. En este momento, el valor de `R1` coincide con el valor de la variable `trabajo_pendiente`, 3.

b) A continuación, decrementa el valor del registro, que pasa a valer 2. Ahora el valor de la copia es diferente del valor de la variable `trabajo_pendiente` en la memoria, 3, y, por lo tanto, hay una inconsistencia.

Líneas de código		Variable contador	Registros	
Proceso en espera	Proceso activo	trabajo_pendiente	R0	R1
...	...	3	...	...
LOAD R0, trabajo_pendiente	**LOAD R1, trabajo_pendiente**	3	...	3
INC R0	**DEC R1**	3	...	2
STORE trabajo_pendiente, R0	STORE trabajo_pendiente, R1			
...	...			

Figura 5. El proceso gestor empieza a decrementar la variable `trabajo_pendiente`.

Si en este punto de la ejecución del código máquina pasa alguna cosa en el sistema[5] que provoca que el proceso gestor deje de ejecutarse, el sistema queda inconsistente.

[5] Un cambio de contexto porque expira el *quantum* del proceso gestor, una interrupción de un dispositivo, etc.

En este estado de inconsistencia, lo peor que puede suceder es que un proceso usuario intente enviar un trabajo a imprimir. Entonces, cuando el proceso quiere incrementar el valor de la variable `trabajo_pendiente`, sucede lo siguiente (figura 6):

a) Se hace una copia de la variable en el registro `R0`, y `R0` vale 3, porque la variable `trabajo_pendiente` continúa teniendo el valor de 3, ya que el gestor no lo había actualizado.

b) Se incrementa el valor del registro `R0`, que pasa a valer 4.

c) Se actualiza la variable `trabajo_pendiente`, que pasa a valer 4.

Líneas de código		Variable contador	Registros	
Proceso en espera	Proceso activo	trabajo_pendiente	R0	R1
...	...	3	...	2
STORE trabajo_pendiente, R1	LOAD R0, trabajo_pendiente	3	3	2
	INC R0	3	4	2
	STORE trabajo_pendiente, R0	4	4	2
	...			

Figura 6. El proceso usuario incrementa la variable trabajo_pendiente.

Al cabo de un cierto tiempo, cuando el proceso gestor se vuelve a ejecutar (figura 7), como se había interrumpido justo cuando debía ejecutar la instrucción `STORE`, lo primero que ejecuta es `STORE trabajo_pendiente, R1`. El valor de R1 en el proceso gestor era 2, por lo tanto el valor de la variable `trabajo_pendiente` en la memoria se actualiza y pasa a ser 2.

Líneas de código		Variable contador	Registros	
Proceso en espera	Proceso activo	trabajo_pendiente	R0	R1
...	...	4	4	2
...	STORE trabajo_pendiente, R1	2	4	2
	...			

Figura 7. El proceso gestor finaliza el decremento de la variable trabajo_pendiente.

Después de esta secuencia de operaciones, el valor de `trabajo_pendiente` no es correcto. Habíamos partido de un valor de 3, el proceso gestor lo ha decrementado una vez y un proceso usuario lo ha incrementado otra vez. Por lo tanto, el valor final de la variable debería ser 3 y, en cambio, lo que hemos conseguido es asignarle un valor igual a 2.

A efectos prácticos, en el sistema hay un trabajo que no se imprimirá.

2) Intento de impresión de un trabajo no existente

El caso simétrico al anterior es aquél en el que el usuario inicia la operación de actualización de la variable y no la finaliza porque es interrumpido (figura 8 y figura 9). Este proceso implica que el valor final de la variable considerada, `trabajo_pendiente`, sea erróneo e igual a 4.

Líneas de código		Variable contador	Registros	
Proceso en espera	Proceso activo	trabajo_pendiente	R0	R1
...	...	3	...	...
LOAD R1, trabajo_pendiente	**LOAD R0, trabajo_pendiente**	3	3	...
DEC R1	**INC R0**	3	4	...
STORE trabajo_pendiente, R1	STORE trabajo_pendiente, R0			
...	...			

Líneas de código		Variable contador	Registros	
Proceso en espera	Proceso activo	trabajo_pendiente	R0	R1
...	...	3	4	...
STORE trabajo_pendiente, R0	**LOAD R1, trabajo_pendiente**	3	4	3
...	**DEC R1**	3	4	2
	STORE trabajo_pendiente, R1	2	4	2
	...			

Figura 8. El proceso usuario es interrumpido mientras incrementa la variable trabajo_pendiente y el proceso gestor decrementa la misma variable.

Líneas de código		Variable contador	Registros	
Proceso en espera	Proceso activo	trabajo_pendiente	R0	R1
...	...	2	4	2
...	**STORE trabajo_pendiente, R0**	4	4	2
	...			

Figura 9. El proceso usuario finaliza el incremento de la variable trabajo_pendiente.

En este caso, el gestor de la impresora intentaría enviar a imprimir un trabajo que no existe.

Todos los procesos se habrían ejecutado correctamente si las operaciones de incremento y decremento que el programador escribe en el lenguaje de alto nivel se hubieran ejecutado de manera indivisible o atómica. La indivisibilidad a la hora de ejecutar estas operaciones nos asegura a efectos prácticos que cuando un proceso incrementa/decrementa la variable no deja el procesador hasta que no ha finalizado la actualización o, dicho de otra manera, una vez iniciada una operación de actualización de una variable compartida, ningún proceso puede iniciar otra operación de actualización de aquella variable hasta que no ha finalizado la primera.

Con este ejemplo se demuestra que la ejecución concurrente no sincronizada de procesos que utilizan variables compartidas puede dar lugar a errores irrecuperables. Este tipo de situaciones recibe el nombre de *race conditions* (condición de carrera) porque pueden generar un resultado erróneo en función de cómo hayan sido planificados los procesos y hay que erradicarlas de los programas.

2.1.2. La sección crítica

Ante la posibilidad de generar resultados no coherentes e incorrectos durante la ejecución concurrente de procesos con variables compartidas, es absolutamente necesario encontrar alguna solución. Antes de hacer alguna propuesta, veamos cuál es el problema real en el funcionamiento descrito en el ejemplo

anterior. Como hemos visto, la aparición de copias temporales y la posibilidad de que un proceso acceda al contenido de una variable compartida en cualquier instante antes de asegurarse de que todas las peticiones de modificación previas han finalizado son la causa principal de todos los males.

La actualización de una variable compartida puede ser considerada una **sección crítica**, es decir, una secuencia bien delimitada (con inicio y fin) de instrucciones que modifican una o más variables compartidas.

Cuando un proceso entra en una sección crítica, debe completar todas las instrucciones de dentro de la sección antes de que cualquier otro proceso pueda acceder a la región crítica de la misma variable compartida. De esta manera, se garantiza que las variables compartidas que se modifican dentro de la sección tan sólo son modificadas por un único proceso a la vez. Esta manera de proceder se denomina **acceso a la sección crítica en exclusión mutua** y garantiza que hasta que un proceso no finaliza la modificación de la variable compartida ningún otro proceso pueda empezar a modificarla[6].

(6)Es decir, el acceso se efectúa estrictamente de manera secuencial.

La solución al problema del acceso en exclusión mutua a variables y dispositivos compartidos en el sistema debe cumplir los requisitos siguientes:

a) Asegurar la exclusión mutua entre los procesos a la hora de acceder al recurso compartido.
b) No efectuar ningún tipo de suposición de la velocidad de los procesos ni del orden en el que se ejecutarán.
c) Garantizar que si un proceso finaliza el acceso a la zona de exclusión mutua, no afectará al resto de procesos que estén interesados en acceder a la sección crítica.
d) Permitir que todos los procesos que están esperando para entrar en la sección crítica puedan hacerlo en un tiempo finito (evitar la inanición, *starvation*).

La manera más sencilla de asegurar la exclusión mutua es no permitir la concurrencia de procesos. Pero esta solución es inaceptable. Lo que nos interesa es determinar protocolos y herramientas de acceso a las secciones críticas que cumplan los requisitos que hemos enumerado.

En la mayoría de las estrategias propuestas para garantizar la exclusión mutua, los procesos siguen el protocolo siguiente (figura 10):

```
1. - Petición de acceso a la sección
2. - Acceso a la sección crítica
3. - Liberar la sección crítica
```

Figura 10. Protocolo de entrada y salida de una región por crítica

Antes de acceder a la sección crítica, todos los procesos formulan una petición de acceso; entonces:

1) Si la sección crítica está ocupada por otro proceso, esperan a que se liberen bien bloqueándose[7], bien llevando a cabo una espera activa[8]. En esta asignatura consideraremos que los procesos en espera se bloquean y no realizan una espera activa porque, en general, ello supone una mejor utilización de los recursos del sistema operativo.

(7)Durmiéndose, poniéndose en alguna cola de espera.

(8)Consultando continuamente el estado del recurso hasta que se produzca un cambio de estado.

2) Cuando la sección crítica esté libre, entrará uno de los procesos en espera y cerrará el acceso a cualquier otro proceso que esté interesado en entrar. Una vez dentro, utilizará el recurso compartido y cuando lo haya utilizado, liberará la exclusión mutua para que otros procesos puedan utilizar el recurso.

En el apartado siguiente describimos los semáforos, una estrategia eficiente y fácil de utilizar que permite la exclusión mutua y que en general se utiliza para implementar otras herramientas de software de nivel más alto.

2.2. Semáforos

Los semáforos son una herramienta de sincronización que permite garantizar la exclusión mutua de una sección crítica entre un número arbitrario de procesos de manera limpia y sencilla.

2.2.1. Definición de semáforo

Un semáforo es una variable (asumimos que es de tipo *semaphore*) que tiene asociado un contador. El semáforo se manipula utilizando las tres operaciones siguientes:

- `sem_init(semaphore s, unsigned int valor)`: inicializa el contador del semáforo asignando el valor indicado. El valor inicial del contador debe ser mayor o igual que cero.
- `Sem_wait(semaphore s)`: de manera atómica, espera a que el contador sea mayor que cero; cuando lo sea, decrementará el valor del contador (figura 11).
- `Sem_signal(semaphore s)`: de manera atómica, incrementa el valor del contador asociado al semáforo (figura 12).

```
sem_wait (semaphore s) {
   while (s <= 0) /* atómicamente */
      espera();
   s = s - 1;
}
```

Figura 11. Pseudo-código de `sem_wait`

```
sem_signal (sempahore s) {
   s = s + 1; /* atómicamente */
}
```

Figura 12. Pseudo-código de `sem_signal`

Observaciones:

- Existen dos tipos de semáforos en función del rango de valores que pueda alcanzar el contador asociado al semáforo: **binarios** (el contador sólo puede valer 0 o 1) y ***n*-arios** (el contador puede tomar cualquier valor mayor o igual que cero). Los semáforos binarios son los que implementan de modo más natural el acceso en exclusión mutua a una región crítica.

- Cabe destacar que las operaciones realizan sus acciones de manera atómica. Supongamos que el contador asociado a un semáforo vale 0 y dos procesos están esperando haciendo un `sem_wait` a que el contador tome un valor mayor que cero. Si ahora algún proceso ejecuta un `sem_signal` sobre este semáforo e incrementa el contador del semáforo, únicamente uno de los procesos que ejecuta el `sem_wait` verá que el contador vale 1 y lo decrementará; el otro proceso deberá continuar esperando hasta que se realice otro `sem_signal`.

- Si varios procesos están esperando haciendo un `sem_wait` a que el contador del semáforo tome un valor mayor que cero y algún proceso ejecuta un `sem_signal` sobre este semáforo, se plantea un problema de elección. ¿Cuál de los procesos en espera ha de ver este incremento del semáforo? Inicialmente, consideraremos que esta elección será aleatoria, pero más adelante volveremos sobre este tema.

- Las operaciones básicas no devuelven cuál es el valor actual del contador asociado al semáforo. Algunas implementaciones de semáforos ofrecen una operación para obtener este valor pero en este documento no las consideraremos.

- Los nombres de las operaciones `sem_wait` y `sem_signal` no son estándar. En la literatura o en implementaciones concretas podéis encontrar que la operación `sem_wait` se denomina `P, down, signal, acquire` o `pend`; la operación `sem_signal` se puede denominar `V, up, wait, release` o `post`. Es importante no confundir las llamadas al sistema UNIX `wait` y `signal` con las operaciones sobre semáforos `wait` y `signal`.

2.2.2. Utilización de los semáforos

A continuación, presentamos tres posibles usos de los semáforos: exclusión mutua, contador de recursos y sincronización.

Acceso a la sección crítica en exclusión mutua

Para asegurar el acceso a la sección crítica en exclusión mutua, hay que utilizar los semáforos binarios o *n*-arios de la manera siguiente:

1) Se inicializa el semáforo. El valor inicial del semáforo ha de ser 1 para indicar que no hay ningún proceso dentro de la sección crítica y dejar paso al primer proceso que pida permiso para entrar mediante la operación `sem_wait`.

2) Se ejecuta la operación `sem_wait`. El proceso que quiere acceder a una zona compartida en exclusión mutua ejecuta la función sem_wait sobre el semáforo asociado a la sección.

3) Se ejecuta la operación `sem_signal`. El proceso que ha finalizado la ejecución de la zona crítica libera la exclusión y permite el paso a otros procesos.

Como ejemplo, veamos cuál sería el código de los procesos usuario y del proceso gestor de la impresora en caso de que utilicen semáforos para actualizar la variable compartida `trabajo_pendiente`.

En un principio, el sistema inicializa la variable semáforo exclusion, utilizando el código de la figura 13. Consideremos el **código de un proceso usuario** (figura 14) y el **código del proceso gestor de la impresora** (figura 15).

```
semaphore exclusion;
...
sem_init(exclusion, 1);
```

Figura 13. Definición de la variable de tipo semáforo e inicialización

```
...
/*Generar fichero*/
...
sem_wait(exclusion);
trabajo_pendiente = trabajo_pendiente + 1;
sem_signal(exclusion);
...
```

Figura 14. Código de los procesos usuario

```
...
while (cierto){
   /* Espera activa */
```

```
    while (trabajo_pendiente == 0);
    sem_wait(exclusion);
    trabajo_pendiente = trabajo_pendiente - 1;
    sem_signal(exclusion);
    /*Enviar trabajo a la impresora*/
    ...
}
```

Figura 15. Código del proceso gestor

Para ilustrar el comportamiento y el funcionamiento de los semáforos, mostramos a continuación un posible escenario de ejecución de tres procesos usuario (`P1`, `P2`, `P3`) y el proceso gestor (`G1`). Las diferentes columnas indican qué acciones llevan a cabo cada uno de los procesos en cada unidad de tiempo, qué procesos están esperando el semáforo, qué proceso está dentro de la sección crítica y cuál es el valor del semáforo en cada instante concreto (figura 16).

Supondremos que el semáforo, `exclusion`, se ha inicializado en 1 y que los procesos usuario y el proceso gestor están dentro de un bucle infinito. Los procesos usuario generan continuamente trabajos que se envían a la impresora. El proceso gestor imprime continuamente los trabajos que han generado los procesos usuario.

En la tabla se ve que en el tiempo `T0` se inicializa el semáforo y todavía no se ha creado ningún proceso. No hay ningún trabajo pendiente de ser imprimido. En `T1`, el semáforo exclusion ha decrementado su valor hasta 0. Esto indica que un proceso ha recibido el permiso para entrar en la sección crítica. En este caso, ha sido el proceso usuario `P1`. El resto de los procesos, el gestor y los usuarios están intentando entrar en la sección, pero el mecanismo del semáforo asegura que ningún otro proceso pueda entrar hasta que `P1` lo libere mediante la ejecución de la operación `sem_signal(exclusion)`.

Tiempo	Estado de los procesos				Semáforo 1-Libre 0-Ocupado	Dentro: fuera (espera)
	P1	P2	P3	G1		
T0	...	...	...	...	1	... : ...
T1	sem_wait	sem_wait	sem_wait	sem_wait	0	... : P1, P2, P3, G1
T2	Sección crítica	Esperando	Esperando	Esperando	0	P1 : P2, P3, G1
T3	sem_signal	Esperando	Esperando	Esperando	1	... : P2, P3, G1
T4	Otras operaciones	Esperando	Sección crítica	Esperando	0	P3 : P2, G1
T5	sem_wait	Esperando	Sección crítica	Esperando	0	P3 : P2, G1, P1
T6	Esperando	Esperando	sem_signal	Esperando	1	... : P2, G1, P1
T7	Sección crítica	Esperando	Otras operaciones	Esperando	0	P1 : P2, G1
T8	sem_signal	Esperando	Otras operaciones	Esperando	1	... : P2, G1

Figura 16. Evolución del estado de los tres procesos usuario, del proceso gestor y del semáforo

Supongamos que los tres procesos usuario generan los ficheros correspondientes y, tal como se ha indicado en el código del proceso usuario, a continuación examinan el valor del semáforo `exclusion` con la intención de decre-

(9)Con esta utilización de los semáforos, el orden de acceso de los procesos a la sección crítica es totalmente aleatorio.

mentarle el valor y entrar en la sección crítica (modificar la variable). Uno de los tres tiene éxito, aunque no podemos asegurar cuál acabará entrando[9] en la sección crítica. En el ejemplo, la operación `sem_wait` que ejecuta `P1`, después de haber leído el valor del semáforo a 1, debe apropiarse la variable semáforo y evitar que el resto de los procesos concurrentes (`P2` y `P3`) puedan acceder a ella antes de que sea modificada (decrementada a 0). Éste es el motivo por el cual el código de la operación `sem_wait` debe ser indivisible. Más adelante veremos cómo podemos conseguirlo.

Una vez dentro de la sección crítica, el proceso `P1` actualiza la variable. A continuación, libera la sección crítica ejecutando la operación `sem_signal(exclusion)`. En este instante, `P2`, `P3` y el proceso gestor de la impresora tienen la misma probabilidad de entrar en la sección crítica. En el ejemplo, es el proceso `P3` el que lo consigue. Se vuelven a repetir las mismas operaciones y cuando `P3` libera la sección entra de nuevo `P1`, mientras que los procesos `P2` y el proceso gestor de la impresora continúan esperándose. Esta situación se conoce como *inanición* y no es deseable; más adelante, al hablar de la implementación de los semáforos, veremos cómo se puede evitar.

Observación: la **granularidad de un semáforo** indica el número de zonas críticas a las que está asociado. Distinguimos dos grados de granularidad:

- **Granularidad baja**: implica tener semáforos diferentes para zonas críticas diferentes. Esto aumenta el grado de concurrencia entre los diferentes procesos concurrentes. En general, interesa tener este grado de granularidad.

- **Granularidad alta**: es cuando se utiliza un mismo semáforo para asegurar la exclusión mutua de diferentes zonas críticas (independientes entre ellas).

Contador de recursos disponibles

Muchas aplicaciones deben controlar la cantidad de recursos disponibles de un determinado tipo pero no necesitan saber exactamente cuántos recursos disponibles hay, sólo necesitan saber si hay alguno; si no hay ninguno, han de esperar a que haya alguno libre.

Es posible utilizar semáforos *n*-arios en este entorno. El contador asociado al semáforo reflejará el número de recursos disponibles.

1) Se inicializa el semáforo. El valor inicial del semáforo debe ser el número inicial de recursos disponibles (será un número entero mayor o igual que cero).

2) Se ejecuta la operación `sem_wait`. Cuando un proceso necesite un recurso, invocará esta operación. Si hay algún recurso disponible, se decrementará el contador asociado al semáforo; de lo contrario, se esperará hasta que haya algún recurso disponible.

3) Se ejecuta la operación `sem_signal`. El proceso que ha finalizado en la utilización de un recurso incrementa el contador y permite que otros procesos lo utilicen.

El ejemplo anterior de los procesos usuario y el proceso gestor de la impresora se podría haber resuelto utilizando un semáforo *n*-ario y aprovechando el contador del semáforo para acumular el número de trabajos pendientes (figura 17). El proceso usuario incrementará este contador haciendo `sem_signal` (figura 18) y el proceso gestor esperará la aparición de trabajos efectuando `sem_wait` (figura 19). El contador asociado al semáforo indicará el número de trabajos pendientes.

```
semaphore trabajos_pendientes;
...
sem_init(trabajos_pendientes, 0);
```

Figura 17. Definición de la variable de tipo semáforo e inicialización

```
...
while (cierto){
   /*Generar fichero*/
   ...
   sem_signal(trabajos_pendientes);
}
```

Figura 18. Código de los procesos usuario

```
...
while (cierto){
   sem_wait(trabajos_pendientes);
   /*Enviar trabajo a la impresora*/
   ...
}
```

Figura 19. Código del proceso gestor de la impresora

En el apartado 2 del anexo, tenéis un ejemplo más completo de la utilización de este tipo de semáforos. Fijaos que en la figura 19 el gestor se bloquea si no hay trabajos pendientes. En cambio, el código de la figura 15 hacía una espera activa.

Sincronización entre procesos

Un semáforo binario o *n*-ario puede implementar una sincronización entre dos procesos en la que un proceso espera una autorización por parte del otro con el fin de poder continuar su ejecución.

1) Se inicializa el semáforo. El valor inicial del semáforo debe ser 0.

2) Se ejecuta la operación `sem_wait`. El proceso que espera la autorización por parte del otro proceso debe invocar esta operación.

3) Se ejecuta la operación `sem_signal`. El proceso que debe conceder la autorización ha de invocar esta operación.

El proceso que espera la autorización (figura 21) ejecutará el código posterior al `sem_wait` únicamente cuando el proceso que autoriza (figura 22) haya ejecutado el `sem_signal`.

```
semaphore sync;
...
sem_init(sync, 0);
```

Figura 20. Definición de la variable de tipo semáforo e inicialización

```
...
sem_wait(sync);
...
```

Figura 21. Código del proceso que espera autorización

```
...
sem_signal(sync);
...
```

Figura 22. Código del proceso que autoriza

2.2.3. Consideraciones sobre la implementación de los semáforos

Como observación, presentamos algunas consideraciones sobre la implementación de los semáforos:

- La operación `sem_wait` debe realizar una espera hasta que el contador asociado al semáforo alcance un determinado valor. Esta espera se puede implementar básicamente de dos maneras: espera activa y bloqueo. La espera activa es la opción más sencilla pero provoca un cierto desperdicio del procesador. En función de la cantidad de procesos en ejecución, este desperdicio puede ser inaceptable.

- Cuando varios procesos están esperando haciendo un `sem_wait` sobre un mismo semáforo y algún proceso invoca `sem_signal`, la implementación de semáforos debe elegir uno de los procesos en espera. Se pueden considerar varios criterios como el aleatorio, FIFO (*first-in, first-out*), prioridades, etc. La implementación debería elegir un criterio que no provoque inanición (*starvation*, como en la figura 16), es decir, evitar que un proceso espere indefinidamente mientras otros procesos completan el `sem_wait` sobre el mismo semáforo. El criterio FIFO evita la inanición y es el que suelen utilizar las soluciones basadas en bloqueos.

- Algunos sistemas operativos ofrecen los semáforos dentro de su repertorio de llamadas al sistema. Estas implementaciones suelen estar basadas en bloqueos. Los procesos en espera pasan al estado *blocked* y dejan de competir por consumir tiempo de procesador hasta que el sistema operativo detecta que se ha hecho un `sem_signal` sobre el semáforo y los pasa al estado *ready*.

- En el apartado 1 del anexo describimos el soporte que debe ofrecer el hardware para poder implementar la exclusión mutua.

2.3. Semáforos en Linux

Linux ofrece una implementación de semáforos POSIX dentro de su repertorio de llamadas al sistema. En este subapartado presentamos una descripción básica de la interfaz de llamadas al sistema y de cómo utilizarlos dentro de los programas. Aunque la implementación ofrece semáforos con nombre (accesibles a través del sistema de ficheros) y semáforos sin nombre (accesibles por medio de la memoria compartida), en este subapartado nos centraremos en los semáforos sin nombre y que serán utilizados por flujos de ejecución que pertenezcan a un mismo proceso (por lo tanto, comparten el espacio lógico de memoria).

POSIX ofrece el tipo de datos `sem_t` para declarar variables de tipo semáforo. La interfaz de llamadas al sistema es la siguiente (observad que el parámetro de tipo semáforo se pasa por referencia):

- `int sem_init(sem_t * sem, int pshared, unsigned int value)`: inicializa el semáforo al valor indicado. En el parámetro pshared especificaremos el valor 0.

- `int sem_wait(sem_t *sem)`: análoga a la explicada en el subapartado anterior.

- `int sem_post(sem_t *sem)`: es la equivalente a la operación sem_signal explicada en el subapartado anterior.

Las tres llamadas devuelven 0 en caso de poderse realizar correctamente y -1 en caso de error. Un posible error está relacionado con la recepción de señales software (apartado 3.2).

Como ejemplo, la figura 23 muestra un programa[10] que emula el funcionamiento de un motor de explosión de cuatro tiempos. Para hacerlo, crea cuatro flujos de ejecución que se sincronizan utilizando semáforos para garantizar que la ordenación de los tiempos sea la correcta (admisión, compresión, explosión y escape). Notad que uno de los semáforos ha sido inicializado en 1 para permitir que el flujo Admisión pueda "arrancar" el motor.

(10)Para generar el ejecutable correspondiente hay que utilizar la biblioteca de *pthreads* (parámetro de compilación `-pthread`).

```
#include <semaphore.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

sem_t sync_tiempo[4];

char *name[4]={"Admisión", "Compresión", "Explosión", "Escape"};

/* Código común para los cuatro tiempos.
   En función del parámetro "par", cada uno imprimirá un mensaje
   y determinará qué semáforos utiliza
*/

void *tiempo(void *par)
{
  int t = (int) par;

  while (1) {
    if (sem_wait(&sync_tiempo[t]) < 0) error();

    printf("%d %s\n", t, name[t]);
    sleep(rand() % 4); /* Retraso aleatorio */

    if (sem_post(&sync_tiempo[(t+1)%4]) < 0) error();
  }
}

int main()
{
  int i;
  pthread_t th[4];

  /* Sincronismo Escape -> Admisión. Inicializado en 1 */
  if (sem_init(&sync_tiempo[0], 0, 1) < 0) error();
```

```
    /* Sincronismo Admisión -> Compresión */
    if (sem_init(&sync_tiempo[1], 0, 0) < 0) error();
    /* Sincronismo Compresión -> Explosión */
    if (sem_init(&sync_tiempo[2], 0, 0) < 0) error();
    /* Sincronismo Explosión -> Escape */
    if (sem_init(&sync_tiempo[3], 0, 0) < 0) error();

    /* Creación de threads */
    for (i=0; i<4; i++)
      if (pthread_create(&th[i], NULL, tiempo, (void*)i) < 0) error();

    /* Espera la muerte de los threads */
    for (i=0; i<4; i++)
      if (pthread_join(th[i], NULL) < 0) error();

    return(0);
  }
```

Figura 23. Programa de ejemplo que utiliza semáforos POSIX en Linux.

3. Segundo escenario: memoria no compartida

En este escenario, los procesos no comparten el espacio lógico (porque el SO no lo permite o porque los procesos no lo desean). Para que estos procesos puedan comunicarse, el SO debe ofrecer un mecanismo de paso de mensajes. Presentaremos las características abstractas del mecanismo y a continuación describiremos las características concretas de los mecanismos de paso de mensajes disponibles en Unix. También presentaremos las señales de software, que permiten acercar al usuario un mecanismo similar al de las interrupciones hardware. Estas señales permitirán que los procesos puedan recibir notificaciones de acontecimientos asíncronos por parte del SO, o de otros procesos, y tratarlos convenientemente.

3.1. Paso de mensajes

Hemos visto que repartiendo la realización de una actividad entre un conjunto de procesos que pueden ejecutarse de manera concurrente es posible mejorar considerablemente el rendimiento del sistema. El hecho de llevar a cabo las actividades conjuntamente plantea la necesidad de sincronizar y de comunicar los datos, ya sean datos parciales, resultados finales, etc. Como a menudo las dos acciones son necesarias, se han desarrollado mecanismos que permiten la sincronización y, al mismo tiempo, el intercambio de información.

El **paso de mensajes** es un mecanismo bastante sencillo e intuitivo que permite la sincronización y la comunicación entre procesos que se están ejecutando en sistemas centralizados o en sistemas distribuidos.

En general, un **mensaje** es un conjunto de informaciones que pueden ser intercambiadas entre un emisor y un receptor. Un mensaje puede contener cualquier tipo de información. El formato de un mensaje es flexible pero, por norma general, incluye campos para indicar el tipo, la longitud, los identificadores del emisor y del receptor y un campo de datos. De hecho, toda esta información y la que se pueda añadir se clasifica en los dos grupos siguientes:

- La **cabecera**, con un formato predefinido en un sistema, contiene la información necesaria para que el mensaje pueda llegar a su destino correctamente. Suele tener una longitud fija.

- El **cuerpo**, constituido por el mensaje propiamente dicho. Suele ser de longitud variable.

Los sistemas operativos actuales

Son muchos los sistemas operativos actuales que proporcionan algún mecanismo que permite la comunicación de los procesos mediante el paso de mensajes. Por ejemplo, UNIX ofrece una implementación específica de este mecanismo.

3.1.1. Características generales

Un sistema que soporta mensajes generalmente ofrece dos tipos de operaciones, que son enviar_mensaje (*send*) y recibir_mensaje (*receive*). La implementación específica del paso de mensajes en un sistema afecta directamente a los parámetros que se han de pasar a estas operaciones. Así pues, pasamos a describir con más detalle algunas cuestiones que se deben tener en cuenta a la hora de definir un mecanismo de paso de mensajes:

- Los mensajes directos y los mensajes indirectos.
- La comunicación síncrona y asíncrona.
- La longitud de los mensajes.

Los mensajes directos y los mensajes indirectos

Si el tipo de mensaje es **directo**, el proceso emisor debe indicar cuál es el proceso receptor y, al revés, el proceso receptor debe indicar cuál es el proceso de quien quiere recibir el mensaje.

Por ejemplo, si tenemos dos procesos A y B y A quiere enviar un mensaje a B, se podría ejecutar en los dos procesos el código siguiente (figura 24):

```
Proceso A
... send(B, mensaje)

Proceso B
... receive(A, mensaje)
```

Figura 24. Paso de mensajes directo

donde A y B han de ser los identificadores de los procesos dentro del sistema y la variable mensaje contiene la información que se están intercambiando. Este tipo de comunicación se denomina **comunicación simétrica**, ya que cada emisor debe conocer todos los posibles receptores, y al revés.

En el caso de que el receptor no conozca a priori quién es el emisor que envía el mensaje, se habla de **comunicación asimétrica**.

Cuando el tipo de mensaje es **indirecto**, el proceso emisor no debe indicar cuál es el proceso receptor y el proceso receptor no debe indicar cuál es el proceso emisor. Los mensajes se envían y se reciben mediante ciertas estructuras que hacen de intermediarias. Estas estructuras se denominan **buzones (*mailboxes*)** a causa de su funcionamiento.

Por ejemplo, si el proceso A quiere enviar un mensaje al proceso B mediante un buzón, los procesos podrían ejecutar las instrucciones siguientes (figura 25):

```
Proceso A
... send(buzón1, mensaje)

Proceso B
... receive(buzón1, mensaje)
```

Figura 25. Paso de mensajes indirecto

El proceso A deja el mensaje en el buzón designado, `buzón1`, mediante la operación `send(buzón1, mensaje)` y el proceso `B` cuando lo quiere leer lo saca del buzón mediante la operación `receive(buzón1, mensaje)`. Cuando se dispone de un mecanismo de este tipo, también son necesarias otras operaciones para el mantenimiento del buzón, como las de crear y destruir un buzón.

Consideraremos que intentar leer (*receive*) de un buzón vacío bloquea la ejecución del proceso e intentar escribir (*send*) en un buzón lleno también lo hace. La comunicación mediante buzones permite la comunicación de uno a uno, de uno a muchos, de muchos a uno y de muchos a muchos (seguramente con ciertas restricciones).

La comunicación síncrona y asíncrona

La **comunicación síncrona** se basa en el intercambio síncrono de mensajes. Este intercambio implica que el emisor y el receptor deben acabar la transferencia de información en el mismo momento, se deben sincronizar[11].

(11) La telefonía clásica sería un ejemplo convencional de este tipo de comunicación.

En sistemas síncronos, la operación `send` es bloqueadora; por lo tanto, si un emisor quiere enviar un mensaje y el receptor todavía no ha efectuado la operación complementaria, `receive`, el emisor queda bloqueado. Por lo tanto, sólo puede haber un único mensaje pendiente para cada pareja de emisor-receptor.

Las ventajas de la comunicación síncrona son la fácil implementación y un bajo coste de recursos. Además, este modo de comunicación permite establecer un punto de sincronismo entre el emisor y el receptor: el emisor puede estar seguro de que el mensaje ha sido recibido.

La **comunicación asíncrona** no bloquea el proceso y el sistema operativo se encarga de guardar el mensaje temporalmente en la memoria hasta que se lleva a cabo la operación receive complementaria. Con esta manera de proceder, el emisor, una vez ha enviado el mensaje, puede continuar su ejecución independientemente de lo que hagan los receptores[12].

(12)El correo postal sería un ejemplo convencional de este tipo de comunicación.

Con la comunicación asíncrona, aumenta el grado de concurrencia del sistema. Como inconveniente, cabe mencionar la gestión de los vectores de memoria intermedia para almacenar los mensajes pendientes, que es el aspecto más complicado de este método de comunicación.

La longitud de los mensajes

Por último, hemos de considerar la longitud de los mensajes, que puede ser fija o variable. Debemos tener en cuenta, a la hora de decidir la longitud de los mensajes, que hay que llegar a un compromiso entre la flexibilidad de los mensajes de longitud variable y la sencillez y la facilidad de gestión de los mensajes de longitud fija.

3.1.2. Ejemplo: la exclusión mutua mediante mensajes

Un método muy sencillo de implementar el acceso en exclusión mutua a una región crítica es mediante mensajes indirectos. En el ejemplo que se muestra a continuación, se crea un buzón que denominamos buzón. Para pedir acceso a la sección crítica, el proceso lee del buzón con la operación receive y entonces:

- Si el buzón está vacío, el proceso se queda bloqueado esperando que alguien le escriba el mensaje.
- Si el buzón está lleno, el proceso lee el mensaje, deja el buzón vacío y accede a la sección.

Nos interesa que el primer proceso que pida acceso a la sección crítica pueda pasar. Para conseguirlo, el buzón se inicializa introduciendo un mensaje de cualquier longitud. En el código que se presenta en la figura 26, el buzón desarrolla un papel equivalente al de un semáforo binario.

```
receive(buzón, mensaje);  /* entrada exclusión mutua */
/* sección crítica */
...
```

```
send(buzón, mensaje);     /* salida exclusión mutua */
```

Figura 26. Implementación de la exclusión mutua utilizando mensajes indirectos

También sería posible implementar semáforos *n*-arios con buzones, así se permitiría que en el buzón pudiera haber hasta *n* mensajes.

3.1.3. Paso de mensajes en Unix (colas de *bytes*)

Unix ofrece diferentes mecanismos de paso de mensajes de tipo indirecto, asíncrono y de tamaño variable. En esta sección, consideramos tres mecanismos: las *pipes*, las *named pipes* y los *sockets*.

Conceptualmente, los tres mecanismos son muy similares. Ofrecen una cola de *bytes* que se gestionará siguiendo la política FIFO (*first in, first out*). Algunos procesos (productores) añadirán información en un extremo de la cola y otros procesos (consumidores) podrán extraerla del otro extremo. Por lo tanto, serán mecanismos propicios para implementar modelos de comunicación productor-consumidor.

Los tres mecanismos difieren en los requisitos que imponen a los procesos que pretenden utilizarlos y en la serie de llamadas al sistema que deben invocar para utilizarlos.

- **Las *pipes*** permiten comunicar dos procesos que tengan algún tipo de parentesco (padre e hijo, dos hermanos, etc.).
- **Las *named pipes*** permiten comunicar dos procesos del sistema que tengan acceso al directorio del sistema de ficheros donde se haya creado un fichero de tipo *named pipe*.
- **Los *sockets*** permiten comunicar procesos que se ejecutan en máquinas diferentes.

La tabla 1 muestra las llamadas al sistema Unix que hay que invocar para crear, abrir, leer y escribir sobre estos mecanismos. Una vez que el mecanismo está disponible para el proceso (el proceso de usuario dispone de un *file descriptor* para operar sobre el mecanismo de comunicación), los tres mecanismos se acceden con las mismas llamadas al sistema read y write y desde el punto de vista del usuario programador son idénticos.

		Pipes	***Named pipes***	***Sockets***
Llamadas al sistema	Crear	pipe	mknod	socket
	Abrir		open	bind, listen, accept (servidores), connect (clientes)
	Leer	read		

Tabla 1. Llamadas al sistema UNIX que permiten acceder a las *pipes*, *named pipes* y *sockets*.

		Pipes	*Named pipes*	*Sockets*
	Escribir	write		
	Cerrar	close		

Tabla 1. Llamadas al sistema UNIX que permiten acceder a las *pipes*, *named pipes* y *sockets*.

Aunque las lecturas y escrituras sobre estos mecanismos se realizan con las mismas llamadas al sistema que permiten acceder a ficheros ordinarios (`read` y `write`), hay tres casos particulares que se tratan de manera especial en el caso de trabajar sobre colas de *bytes*:

- **Lectura sobre cola vacía**. Cuando un proceso consumidor intenta leer de una cola que en este momento se encuentra vacía, la cola puede estar vacía debido a que el proceso consumidor "es más rápido" que el proceso productor o que el proceso consumidor ya ha consumido toda la información que el productor debía producir. Normalmente, al proceso consumidor le interesará tratar de modo diferente los dos casos; en el primer caso, lo más razonable sería esperar a que el productor produzca alguna cosa; en el segundo caso, lo más razonable sería recibir la notificación de final de fichero. Para que el SO pueda saber en qué caso nos encontramos, utilizará un dato que conoce con exactitud: el número de *file descriptors* de escritura abiertos actualmente sobre esta cola. Si este número es mayor que 0, considerará que estamos en el primer caso; si el número es 0, considerará que estamos en el segundo caso. Por lo tanto, en el primer caso la llamada read se bloqueará hasta que algún productor escriba alguna cosa en la cola; en el segundo caso, la llamada `read` volverá inmediatamente indicando que se han leído 0 caracteres.

Los *file descriptors* de escritura

El hecho de que el SO utilice el número de *file descriptors* de escritura abiertos provoca que dejar abierto innecesariamente algún *file descriptor* de escritura sobre una cola de *bytes* pueda provocar comportamientos anómalos en nuestros programas. Es aconsejable cerrar los *file descriptors* tan pronto como dejen de ser necesarios.

- **Escritura sobre cola llena**. Internamente, el SO asigna una medida a estas colas. Si se produce información sobre una cola a un ritmo superior al que la información es consumida, llegará un momento en el que la cola se llenará. Si en este momento se intenta añadir más información, el comportamiento por defecto es que la llamada `write` se bloquee[13] hasta que haya bastante espacio disponible para escribir todos los datos.

[13] No consideramos el caso de que una llamada write intente escribir un número de *bytes* superior a la medida interna de la cola de *bytes*.

- **Escritura sobre cola sin proceso consumidor**. Unix considera que esta situación es anómala y probablemente provocada por un error en alguno de los procesos involucrados. Por lo tanto, avisa al proceso productor utilizando un mecanismo parecido a las excepciones vistas en el módulo 2, el mecanismo de las señales software (los veremos en el apartado 3.2). Si el proceso consumidor no está preparado para tratar esta señal, el SO le hará abortar.

Otras diferencias entre *pipes* y ficheros ordinarios son que no es posible utilizar la llamada lseek para reposicionar el puntero de lectura escritura sobre una pipe (la gestión de los datos es estrictamente FIFO) y que las *pipes* son volátiles. Si detenemos el sistema operativo (*shutdown*), se pierde toda la información que no haya sido leída de una *pipe*.

Ejemplo: *pipes*

Una *pipe* es un dispositivo que no se puede abrir de manera explícita mediante la llamada al sistema *open*; se crea en el momento en el que se invoca la llamada en el sistema *pipe* y se destruye cuando el último proceso que lo tiene abierto lo cierra. La llamada pipe crea el dispositivo y le asocia dos *file descriptors*, uno de lectura y otro de escritura. Si ahora el proceso crea procesos hijos (mediante llamada al sistema fork), éstos heredarán los *file descriptors* de su proceso padre y podrán acceder a esta misma *pipe*.

La figura 27 muestra un ejemplo de utilización de las *pipes* (asumimos que ninguna llamada al sistema devolverá error).

```
int descFichero[2], n, estado;
char buf[512];

estado = pipe(descFichero); /* Creación de la pipe */
switch(fork()){
    case 0:
        /* El proceso hijo lee del canal 0 y escribe en la pipe */
        close(descFichero[0]);
        while ((n = read(0, buf, sizeof(buf))) > 0) {
            write(descFichero[1], buf, n);
        }
        close(descFichero[1]); /* Permite que el padre detecte fin de fichero */
        break;

    default:
        /* El proceso padre lee de la pipe y escribe en el canal 1 */
        close(descFichero[1]); /* Permite detectar el final de transmisión cuando el
                                  hijo acabe */

        while ((n = read(descFichero[0], buf, sizeof(buf))) > 0) {
            write(1, buf, n);
        }
        close(descFichero[0]);
}
...
```

Figura 27. Fragmento de programa que utiliza una *pipe* para comunicar un proceso hijo con un proceso padre.

El proceso pide crear una *pipe*. El SO le devolverá dos *file descriptors* (uno de lectura y otro de escritura) para acceder a la *pipe*, como podéis ver en la figura 28, donde se muestra esquemáticamente el estado del proceso (espacio lógico, PCB[14], tabla de *file descriptors)* y del SO (tabla de punteros de lectura/escritura).

(14)*Process control block*, estructura de datos gestionada por el sistema operativo en la que éste almacena la información necesaria para gestionar cada proceso.

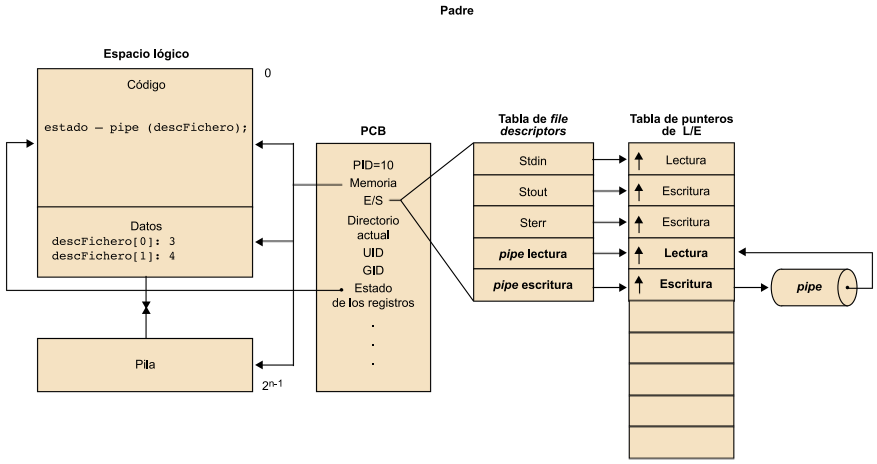

Figura 28. Estado del proceso justo después de haber creado la *pipe*

Para que esta *pipe* pueda comunicar dos procesos, hay que invocar la llamada al sistema fork. De esta manera, el proceso hijo hereda del padre los *file descriptors* entre los que encontramos los que se han creado con la llamada. pipe: descFichero[0] es el de lectura, y descFichero[1], el de escritura.

Después de la creación del hijo, para establecer un canal de comunicación unidireccional entre el hijo y el padre, los dos procesos cierran el canal del sentido de la comunicación que no utilizarán.

Finalmente, el proceso hijo lee datos de su entrada estándar (canal 0, *stdin*) y las escribe en la *pipe*. El proceso padre lee los datos de la *pipe* y los escribe en su salida estándar (canal 1, *stdout*).

La figura 29 muestra el estado de los dos procesos mientras éstos están ejecutando el código correspondiente a las sentencias while.

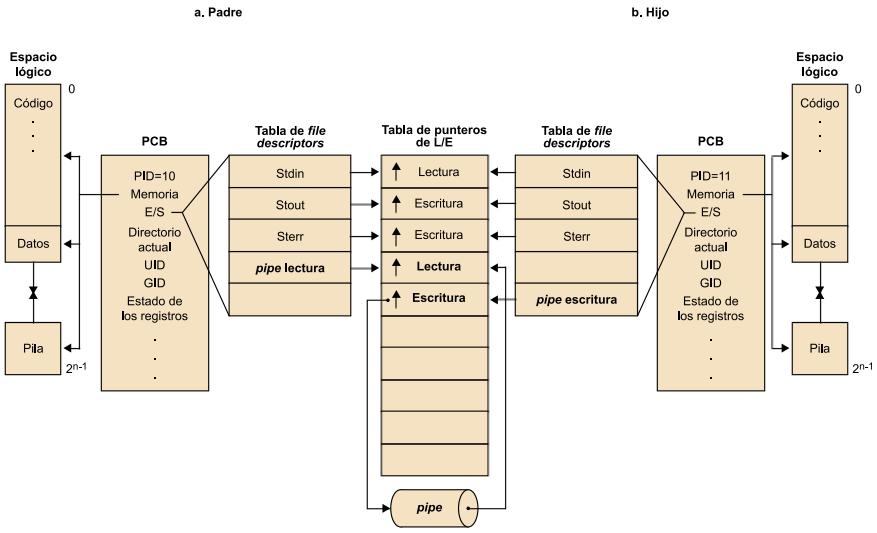

Figura 29. Estado del proceso padre y del proceso hijo mientras los procesos se comunican.

Finalmente, la figura 30 muestra un ejemplo completo de un programa que utiliza *pipes*. Se trata de un programa que emula las llamadas al sistema que debe ejecutar el intérprete de comandos cuando el usuario introduce la línea de comandos `ps aux | grep getty` (el metacarácter | de los intérpretes de comandos Unix permite comunicar la salida estándar de un proceso con la entrada estándar de otro siguiendo el paradigma de comunicación productor-consumidor). El programa crea una *pipe* y dos procesos hijos, uno ejecutará el comando `ps` y el otro el comando `grep`; la salida estándar del primer hijo y la entrada estándar del segundo hijo estarán comunicadas gracias a la *pipe* y a las redirecciones efectuadas (llamada al sistema `dup2`). Notad que los ejecutables `ps` y `grep` son totalmente ajenos al hecho de estar escribiendo/leyendo sobre una *pipe*; `ps` escribe sobre su salida estándar (que en este caso está redireccionada al canal de escritura sobre la *pipe*) y `grep` lee de su entrada estándar (que en este caso está redireccionada al canal de lectura sobre la *pipe*).

```
#include <unistd.h>
#include <sys/wait.h>

int
main (int argc, char *argv[])
{
  int p[2], st;

  if (pipe (p) < 0)
    error ();

  switch (fork ())
    {
    case -1:
      error ();

    case 0:
```

```
        if (dup2 (p[0], 0) < 0)
          error ();
        if (close (p[0]) < 0)
          error ();
        if (close (p[1]) < 0)
          error ();
        execlp ("grep", "grep", "getty", NULL);
        error ();

      default:
        switch (fork ())
         {
        case -1:
         error ();
        case 0:
         if (dup2 (p[1], 1) < 0)
          error ();
         if (close (p[0]) < 0)
          error ();
         if (close (p[1]) < 0)
          error ();
         execlp ("ps", "ps", "aux", NULL);
         error ();
        }

      }
    if (close (p[0]) < 0)
      error ();
    if (close (p[1]) < 0)
      error ();
    if (wait (&st) < 0)
      error ();
    if (wait (&st) < 0)
      error ();
    return (0);
  }
```

Figura 30. Programa que emula las llamadas al sistema que debe ejecutar el intérprete de comandos cuando el usuario introduce el comando `ps aux | grep getty`.

Ved también

Podéis consultar la documentación adicional de la asignatura y el manual de llamadas al sistema para entender todos los parámetros de las llamadas al sistema utilizados.

3.2. Las señales de software (señales software, *signals*)

3.2.1. Descripción

Como hemos visto, un SO define una nueva máquina con una semántica más elaborada que la de la máquina física sobre la que se ejecuta. A pesar de este aumento del nivel semántico, el SO conserva muchas de las funciones y los

mecanismos de los que dispone el hardware. Uno de estos mecanismos que reproduce es el de las interrupciones. Los procesos pueden necesitar ser informados de acontecimientos que suceden de manera imprevista o asíncrona en cualquier instante de la ejecución de un programa con el fin de poder tratarlos y poder continuar la ejecución de éste.

Por ejemplo, un proceso puede necesitar saber si se ha producido un error en el acceso a la memoria con el fin de pedir que se aumente el área asignada a una cierta variable. O puede necesitar saber si un cierto terminal sobre el que trabaja ha sido apagado para acabar la aplicación correctamente, etc.

Las **señales de software** son la herramienta que proporciona el sistema operativo para trasladar el mecanismo de las interrupciones al ámbito del proceso. Igual que el mecanismo de hardware de las interrupciones, las señales dan soporte a un amplio conjunto de situaciones diferentes que los procesos han de conocer y atender con una cierta urgencia.

La **procedencia de las señales de software** nos permite clasificarlas de la manera siguiente:

1) Los **dispositivos de entrada/salida**, que necesitan informar del proceso de situaciones que pueden dar lugar a un error si el proceso no tiene conocimiento de ellas. Son situaciones como, por ejemplo, la desconexión de un terminal, la pulsación de una tecla de control por parte del usuario, etc.

2) Las **excepciones**, que son provocadas por el proceso de manera involuntaria durante la ejecución de una instrucción de lenguaje máquina o a causa de la saturación de un elemento de hardware (*overflow*), de un acceso a la memoria inválido o de una instrucción anómala[15].

(15)Son instrucciones de lenguaje máquina anómalas la división entre cero, la raíz cuadrada de un número negativo, etc.

3) Las **llamadas explícitas al sistema**, que se pueden efectuar para enviar señales a un proceso determinado. Para poder hacerlo, el proceso que envía la señal ha de tener permiso. En general, sólo se permite enviar señales entre procesos del mismo dominio de protección (el administrador puede enviar señales a quien quiera). Se trata de señales como la orden para eliminar un proceso o señales para sincronizar la ejecución de procesos que lo necesiten. Por ejemplo, un proceso que tiene como función inicializar una estructura de datos determinada podría utilizar una señal para informar de los procesos usuarios de esta estructura que ya ha sido inicializada.

4) El **reloj del sistema**, que es utilizado por los procesos cuando necesitan llevar a cabo ciertas acciones a intervalos concretos de tiempo. Por ejemplo, un proceso que esté esperando una cierta entrada desde un módem puede pedir ser avisado dentro de un cierto lapso de tiempo (*timeout*) con el fin de detectar si la línea se ha cortado o no.

5) Finalmente, un proceso puede recibir una señal como el **efecto lateral de una llamada al sistema**. Por ejemplo, un proceso padre puede recibir una señal que le indica que uno de sus procesos hijo ha sido destruido.

El tratamiento que da un proceso a una señal que le llega puede ser de uno de los tres tipos siguientes:

1) Un tratamiento definido por el mismo usuario. El proceso indica mediante una llamada al SO qué procedimiento se debe ejecutar cuando llegue una cierta señal.

2) No dar ningún tratamiento (ignorar el acontecimiento). El proceso puede decidir no hacer caso de la señal y, por lo tanto, cuando llegue ésta continuará su ejecución normalmente como si no hubiera pasado nada.

3) Un tratamiento dado por el SO. Algunas circunstancias que provocan señales son debidas o pueden llevar a un mal funcionamiento del proceso si no se utilizan las medidas necesarias. En estos casos, si el proceso no ha definido un tratamiento, el sistema debe proporcionar uno. Normalmente este tratamiento por defecto lleva asociada la destrucción del proceso al que iba destinada la señal. Por ejemplo, si un proceso efectúa un acceso incorrecto a la memoria y el proceso no lo soluciona, el SO se ve en la obligación de destruir el proceso e informar de ello a su proceso padre.

Así pues, un proceso será destruido por el SO si recibe una señal no esperada a causa de un error en su ejecución, a causa de un acontecimiento imprevisto en uno de los dispositivos a los que accede o por la actuación deliberada de otro proceso. En este caso, el SO es el encargado de enviar el estado de finalización al proceso padre con el fin de indicarle el motivo de la destrucción.

Finalmente, hemos de destacar que la programación de las señales es una información propia de cada proceso que configura el entorno que se va a ejecutar y, como tal, debe formar parte del PCB y se debe tener en cuenta en el momento de la creación de un nuevo proceso.

3.2.2. Linux: señales POSIX

El sistema operativo Unix ofrece a los procesos el mecanismo de las señales. Desgraciadamente, existen varias interfaces de señales (SystemV, BSD, POSIX, etc.) que presentan diferencias de funcionalidad muy significativas. Como la interfaz de señales más completa y que permite hacer programas más robustos es la interfaz POSIX (la utilizada por Linux), en este subapartado presentamos una breve descripción de ella. La interfaz POSIX también recibe el nombre de *reliable signals*, mientras que la SystemV también recibe el nombre de *traditional signals*.

Terminología

A continuación, describimos algunos términos utilizados a lo largo de esta sección:

- **Generación de la señal**: momento en el que se produce una señal.
- **Tratamiento de la señal**: rutina de atención para una determinada señal (puede ser el tratamiento por defecto proporcionado por el SO o una rutina proporcionada por el usuario).
- **Depósito de la señal**: momento en el que se empieza a ejecutar la rutina de tratamiento de una señal.
- **Señal pendiente**: señal que ha sido generada pero que todavía no ha sido depositada.
- **Programación de una señal**: hecho de asociar un tratamiento a una señal.
- **Captura de una señal**: ejecución de una rutina proporcionada por el usuario para atender una determinada señal.
- **Señal bloqueada**: si un proceso bloquea una señal, el SO no depositará temporalmente las señales de este tipo que se generen sobre el proceso. Cuando el proceso desbloquee esta señal, el SO depositará las señales pendientes. Sería equiparable a inhibir temporalmente las interrupciones hardware.

  El término *bloqueado* y las señales pendientes

  El término *bloqueado* tiene diferentes significados en función del contexto porque se puede aplicar a procesos y a señales. Recordemos que un proceso bloqueado es aquel que temporalmente no puede competir para utilizar el procesador porque está esperando algún acontecimiento.

  El número de señales pendientes de una determinada señal está limitado. En algunas versiones POSIX, sólo puede haber una señal pendiente de cada tipo, mientras que otras versiones POSIX permiten acumular varias señales pendientes de un mismo tipo (*queued signals*). En este documento asumiremos que no será posible acumular señales pendientes.

- **Señal ignorada**: si un proceso ignora una señal, el SO descartará las señales de este tipo que se generen sobre el proceso y no las depositará.
- **Conjunto de señales bloqueadas**: atributo de todo proceso que indica qué señales tiene bloqueadas el proceso. Se almacena en el PCB del proceso y se manipula utilizando una llamada al sistema.

La figura 31 muestra un diagrama de tiempo en el que aparecen estos términos. Se representa la ejecución de un proceso que genera, bloquea, desbloquea e ignora señales.

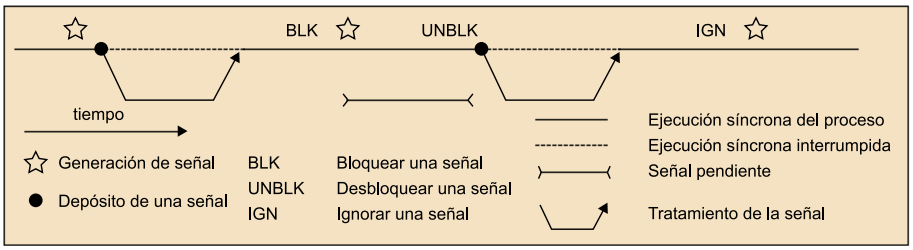

Figura 31. Terminología utilizada para las señales POSIX

Lista de señales

POSIX define una serie de señales con un significado concreto; otras implementaciones de señales presentan variaciones con respecto al tratamiento y al número de señales posibles. En la tabla 2 mostramos algunas de las señales definidas por POSIX con el tratamiento por defecto que asigna el SO. El significado de estos tratamientos es:

- **EXIT**: destrucción del proceso.
- **EXIT+CORE**: destrucción del proceso y generación de un fichero *core* que contiene el estado del proceso en el momento del depósito de la señal.
- **IGNORE**: ignorar la señal.
- **STOP**: detener la ejecución del proceso.
- **CONT**: reanudar la ejecución de un proceso detenido.

Ved también

En el módulo "La gestión de la memoria" hablamos de estos ficheros core cuando un proceso intenta acceder a una dirección de memoria inválida.

Nombre señal	Significado de la señal	Tratamiento por defecto
SIGHUP	Se ha producido un corte de la línea de comunicación, generalmente con un terminal o un módem.	EXIT
SIGINT	Se ha pulsado la secuencia de teclas de control asociada a esta señal (típicamente Ctrl-C).	EXIT
SIGABRT	Aborta la ejecución del proceso.	EXIT + CORE
SIGFPE	Se ha producido una excepción de coma flotante.	EXIT + CORE
SIGKILL	Destruir el proceso. La señal no se puede ignorar, bloquear o programar.	EXIT
SIGSEGV	Se ha producido un acceso indebido a la memoria.	EXIT + CORE
SIGPIPE	Se ha escrito en una *pipe* sin ningún lector.	EXIT
SIGALRM	Se ha producido una expiración del temporizador.	EXIT
SIGTERM	Se solicita que el proceso acabe ("*Prepare to die*").	EXIT
SIGUSR1	Disponible para usos propios de las aplicaciones.	EXIT
SIGUSR2	Disponible para usos propios de las aplicaciones.	EXIT
SIGCHLD	Ha muerto algún proceso hijo.	IGNORE
SIGSTOP	Se ha pulsado la secuencia de teclas de control asociada a esta señal (típicamente Ctrl-Z). La señal no se puede ignorar, bloquear o programar.	STOP
SIGCONT	Reanuda la ejecución de un proceso que haya sido detenido con SIGSTOP.	CONT

Tabla 2. Lista de algunas señales POSIX: nombre de la señal, significado asociado y acción por defecto llevada a cabo por el SO

Cada señal (SIGHUP, SIGKILL, etc.) tiene asociado un código numérico entero; por ejemplo, el SIGKILL tiene asociado el valor 9[16]. Para facilitar la legibilidad del código, en los fragmentos de código de ejemplo utilizaremos los nombres simbólicos de las señales y no su código numérico.

(16)Los habituados a trabajar con algún intérprete de comandos Unix probablemente habrán utilizado el comando `kill -9 pid` para matar un proceso. Este comando genera la señal número 9 (SIGKILL) sobre el proceso con identificador pid.

Conjuntos de señales (*signal sets*)

Diferentes llamadas al sistema relacionadas con la interfaz de señales POSIX están parametrizadas con una variable que representa un conjunto de señales. Por ejemplo, nos puede interesar bloquear las señales SIGUSR1, SIGUSR2 y SIGTERM. Para hacerlo, necesitamos definir una variable de tipo "conjunto de señales" que contenga estos tres tipos de señales. POSIX nos ofrece un tipo de datos (sigset_t) y una serie de funciones para manipular estas variables. La figura 32 enumera estas funciones y describe su funcionalidad; también muestra un par de ejemplos: crear un conjunto con las señales SIGUSR1 y SIGUSR2 y crear un conjunto con todas las señales salvo la SIGTERM.

```
#include <signal.h>

int sigemptyset(sigset_t *set); /* Inicializa "set" en el conjunto vacío */
int sigfillset(sigset_t *set);  /* Inicializa "set" con todos los signals */
int sigaddset(sigset_t *set, int signum);   /* Añade "signum" a "set" */
int sigdelset(sigset_t *set, int signum);   /* Elimina "signum" de "set" */
int sigismember(const sigset_t  *set, int signum);  /* Indica si "signum" pertenece a "set" */
/* Ejemplos de uso */
sigset_t u12 /* Contendrá los signals SIGUSR1 y SIGUSR2 */
         noterm; /* Contendrá todos los signals salvo el SIGTERM */

sigemptyset(&u12); sigaddset(&u12, SIGUSR1); sigaddset(&u12, SIGUSR2);
sigfillset(&noterm); sigdelset(&noterm, SIGTERM);
```

Figura 32. Funciones que permiten manipular variables de tipo sigset_t y dos ejemplos de utilización.

Programación de una señal

La llamada al sistema que permite definir la rutina de atención a una señal es la llamada sigaction (figura 33). Su primer parámetro es el identificador de señal del que queremos modificar la rutina de tratamiento. El segundo parámetro es un puntero a una estructura (struct sigaction) que indica cuál será el nuevo tratamiento:

- En el campo sa_handler, indicaremos qué rutina queremos que atienda esta señal (en caso de que queramos ignorar la señal, hay que indicar el valor SIG_IGN, y en caso de querer recuperar el tratamiento por defecto, hay que indicar el valor SIG_DFL). Cuando esta rutina se ejecute, recibirá como parámetro el código numérico correspondiente a la señal que ha

provocado su ejecución (es posible asociar la misma rutina a señales diferentes).

- En el campo `sa_mask`, de tipo "conjunto de señales", indicaremos qué señales queremos que estén bloqueadas mientras se ejecute la rutina de atención.

- En el campo `sa_flags`, podremos modificar el comportamiento por defecto de las señales POSIX. El valor 0 hace que el tratamiento de esta señal siga el estándar POSIX.

```
#include <signal.h>

struct sigaction {
    void (*sa_handler)(int);
    sigset_t sa_mask;
    int      sa_flags;
};

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Figura 33. Interfaz de la llamada al sistema sigaction

La llamada devuelve el valor 0 si se ha podido reprogramar la señal y -1 en caso de error. Un error típico es intentar reprogramar una señal que el SO no permite reprogramar (por ejemplo, el `SIGKILL`). Además, en el tercer parámetro nos devuelve el puntero a una estructura `struct sigaction` mediante la que podemos recuperar cuál era hasta el momento la programación de esta señal.

En el estándar POSIX, la programación de la señal es válida hasta el momento en el que se vuelva a programar esta misma señal (en el estándar SystemV, sólo era válida hasta el primer depósito de una señal de este tipo).

En el caso de invocar la llamada al sistema `fork`, el proceso hijo hereda la programación de las señales del proceso padre. En el caso de invocar alguna llamada al sistema exec, todas las señales recuperan la programación por defecto salvo aquellos que hayan sido programadas como `SIG_IGN`.

Generación de señales

POSIX proporciona diferentes llamadas al sistema para generar señales. Comentaremos dos: `kill`[17] y `alarm` (figura 34).

[17] El nombre `kill` (matar) viene dado porque las primeras implementaciones de señales se utilizaban únicamente para provocar la muerte de procesos.

```
int kill(pid_t pid, int sig);
unsigned int alarm(unsigned int seconds);
```

Figura 34. Llamadas al sistema que permiten generar señales.

La llamada `kill` genera una señal "inmediatamente" sobre un determinado proceso. Tiene como parámetro el identificador de proceso sobre el que queremos generar la señal y el tipo de señal que se va a generar. La llamada devolverá error si intentamos generar una señal sobre un proceso que pertenece a otro usuario o sobre un proceso inexistente.

La llamada `alarm` permite programar el temporizador asociado al proceso. Tiene como parámetro el número de segundos[18] (de tiempo real) que deben transcurrir antes de que expire el temporizador. Cuando el temporizador expire, el SO generará una señal `SIGALRM` sobre el proceso. Para generar un nuevo `SIGALRM`, habrá que programar nuevamente el temporizador. En el caso de invocar la llamada `alarm` con el parámetro 0, el SO ignorará la última programación del temporizador.

[18] POSIX ofrece otros temporizadores con una resolución más fina, pero no son objeto de este documento.

Conjunto de señales bloqueadas

Todo proceso tiene un atributo en su PCB que indica qué señales tiene bloqueadas en este momento. La gestión de este atributo se debe realizar utilizando la llamada al sistema `sigprocmask`. El primer parámetro de la llamada (`how`) indica el tipo de modificación que queremos hacer (`SIG_BLOCK`, `SIG_UNBLOCK` o `SIG_SETMASK`); la llamada permite modificar el valor de este atributo añadiendo (`SIG_BLOCK`), eliminando (`SIG_UNBLOCK`) o asignando directamente (`SIG_SETMASK`) al conjunto de señales indicado en el segundo parámetro (`set`). En el tercer parámetro, la llamada nos puede devolver el valor del atributo antes de realizar este cambio.

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

/* el atributo se modifica en función de valor de "how" (SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK):

    SIG_BLOCK:       atributo = atributo UNIÓN set;
    SIG_UNBLOCK:     atributo = atributo INTERSECCIÓN COMPLEMENTARIO(set)
    SIG_SETMASK:     atributo = set;
*/
```

Figura 35. Llamada al sistema `sigprocmask`

El valor de este atributo se hereda al invocar la llamada `fork` y se mantiene al invocar alguna de las llamadas `exec`. Si el valor de este atributo se modifica dentro de la rutina de atención a alguna señal (o utilizando el campo `sa_mask` de la estructura `struct sigaction` de la llamada al sistema `sigaction`), al acabar la ejecución de la rutina de atención, el SO restaurará el valor previo de este atributo. Es decir, la ejecución de una rutina de tratamiento de una señal no puede modificar permanentemente el valor de este atributo.

La figura 36 muestra un fragmento de un código de ejemplo que utiliza esta llamada. Se pretende bloquear temporalmente la señal `SIGTERM` mientras el proceso está escribiendo datos en un fichero (se entiende que hay que hacerlo sin modificar al resto de señales bloqueadas por el proceso). Se quiere evitar que se pueda depositar esta señal mientras el proceso escribe datos en el fichero porque esto provocaría la muerte del proceso pudiendo dejar el fichero en un estado incoherente. El programa bloquea temporalmente la señal, con lo que, si la señal se genera, se depositará únicamente cuando todos los datos hayan sido escritos en el fichero. Se presentan tres opciones para hacerlo: la primera y la segunda no son del todo correctas porque no tienen en consideración qué señales pueden estar bloqueadas actualmente; la tercera opción tiene en cuenta todas las posibilidades.

```
sigset_t term, old;

sigemptyset(&term); sigaddset(&term, SIGTERM);

/* Opción 1: Errónea si ya tenemos algún signal bloqueado */
sigprocmask(SIG_SETMASK, &term, NULL);
escritura_fichero();
sigprocmask(SIG_UNBLOCK, &term, NULL);

/* Opción 2: Errónea si ya tenemos el SIGTERM bloqueado */
sigprocmask(SIG_BLOCK, &term, NULL);
escritura_fichero();
sigprocmask(SIG_UNBLOCK, &term, NULL);

/* Opción 3: Correcta */
sigprocmask(SIG_BLOCK, &term, &old);
escritura_fichero();
sigprocmask(SIG_SETMASK, &old, NULL);
```

Figura 36. Ejemplos de utilización de la llamada `sigprocmask`

Otro posible uso del bloqueo de señales es garantizar el acceso en exclusión mutua a una estructura de datos a la que se acceda tanto desde el programa principal como desde una rutina de atención a una señal. Hay que bloquear esta señal mientras el programa principal esté modificando la estructura de datos.

Espera del depósito de una señal

Una de las operaciones más habituales que se realizan con señales es la sincronización, en la que un proceso debe esperar hasta que se deposite un determinado tipo de señal. Asumimos que el proceso está esperando una señal de tipo `SIGUSR1` y que la rutina de tratamiento a esta señal pone en TRUE la variable global `usr1`.

Una primera aproximación consistiría en realizar la espera efectuando una espera activa (figura 37), es decir, que el proceso esté continuamente consultando el valor de esta variable hasta que cambie de valor. Ello penaliza al resto de procesos en ejecución en el sistema y, en general, no sería una solución aceptable.

```
int usr1 = FALSE;

void ras_usr1(int signum)
{
  usr1 = TRUE;
}

int main()
{
  struct sigaction act;

  /* Programación de la rutina de atención en el SIGUSR1 */
  act.sa_handler = ras_usr1;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGUSR1, &act, NULL);

  while (usr1 == FALSE); /* Espera activa */
  ...
}
```

Figura 37. Espera del depósito de una señal utilizando una espera activa

Las versiones tradicionales de señales ofrecían la llamada al sistema `pause` pero, en general, la utilización de esta llamada provoca que los programas no sean *reliables*. La llamada `pause` hace esperar al proceso hasta que se deposite una señal sobre éste. Si utilizamos esta llamada (figura 38), el código resultante puede comportarse de manera errónea en caso de que el `SIGUSR1` se deposite una vez que se ha verificado que usr1 tiene el valor FALSE pero antes de invocar la llamada al sistema `pause`; se ejecutará la rutina de atención a la señal y el proceso esperará el depósito de una señal que ya ha sido depositada.

```
if (usr1 == FALSE) pause();
```

Figura 38. Espera del depósito de una señal utilizando la llamada al sistema `pause`

Para solucionar este problema, la interfaz de señales POSIX proporciona la llamada al sistema `sigsuspend` (figura 39). Esta llamada realiza dos operaciones de manera atómica[19]:

(19)Entre otras cosas, el SO garantiza que no se depositará ninguna señal sobre el proceso mientras se realizan estas dos operaciones.

- hacer que `mask` sea el nuevo valor del atributo del conjunto de señales bloqueadas por el proceso y
- hacer que el proceso se espere[20] hasta el depósito de una señal no bloqueada.

(20)En las llamadas `pause` y `sigsuspend`, el SO no implementa esta espera utilizando una espera activa. El SO provoca que el proceso pase al estado *blocked* hasta que se deposite una señal sobre el proceso.

Cuando se deposite la señal, se ejecutará el tratamiento asociado y, al acabar, se restaurará el valor previo del atributo y el proceso continuará su ejecución.

```
int sigsuspend(const sigset_t *mask);
```

Figura 39. Llamada al sistema `sigsuspend`

Mediante la utilización de esta llamada, podremos dar una nueva solución al problema anterior. La figura 40 muestra el código resultante asumiendo que queremos permitir tratar cualquier otra señal mientras esperamos el depósito del `SIGUSR1`.

```
sigset_t m, old;

sigemptyset(&m); sigaddset(&m, SIGUSR1);

sigprocmask(SIG_BLOCK, &m, &old);
sigemptyset(&m);

while (usr1 == FALSE)
  sigsuspend(&m); /* Esperando cualquier señal */

sigprocmask(SIG_SETMASK, &old, NULL);
```

Figura 40. Llamada al sistema `sigsuspend`

Si mientras ejecutamos el `SIGUSR1` queremos permitir únicamente el tratamiento de las señales que no estuviesen bloqueadas, habría que introducir algunas modificaciones en el código (figura 41).

```
sigset_t m, old;

sigprocmask(SIG_BLOCK, NULL, &m); /* Obtenemos signals bloqueadas */
sigaddset(&m, SIGUSR1);
```

```
sigprocmask(SIG_BLOCK, &m, &old);
sigdelset(&m, SIGUSR1);

while (usr1 == FALSE)
  sigsuspend(&m);/* Esperando SIGUSR1 o cualquier señal no bloqueada */
sigprocmask(SIG_SETMASK, &old, NULL);
```

Figura 41. Espera del depósito de una señal utilizando la llamada al sistema sigsuspend

En los ejercicios de autoevaluación se proponen diferentes actividades relacionadas con el conjunto de señales bloqueadas que tiene el proceso y con la utilización de la llamada al sistema `sigsuspend`.

Depósito de señales sobre procesos bloqueados

Mientras un proceso está bloqueado (por ejemplo, leyendo del teclado o de una *pipe* vacía, esperando la muerte de un proceso hijo, haciendo un `sem_wait` sobre un semáforo), es posible que el SO deposite una señal sobre este proceso. Si la señal no está bloqueada, se ejecutará el tratamiento asociado. Ahora bien, si este tratamiento no provoca la muerte del proceso, ¿qué sucederá con la llamada al sistema invocada por el proceso?

- En algunos casos, el SO abortará la llamada al sistema (la llamada devolverá un -1 como resultado y en la variable `errno` –código de error– encontraremos el valor `EINTR`) y el proceso continuará su ejecución.

- En otros casos, el SO reiniciará la llamada al sistema con lo que el proceso se volverá a bloquear.

En función del tipo de llamada al sistema y de cómo haya sido definido el tratamiento de la señal (campo `sa_flags` de la estructura `struct sigaction` de la llamada al sistema `sigaction`) estaremos en un caso o en otro. Para encontrar más información al respecto, podéis consultar el manual del sistema operativo ejecutando el comando `man 7 signal` sobre algún sistema Linux.

En todo caso, aconsejamos llevar a cabo un control de errores exhaustivo en las llamadas al sistema con el fin de detectar llamadas al sistema que el SO no haya podido servir.

4. *Deadlocks* (abrazos mortales o interbloqueos)

En los sistemas operativos actuales, que contienen un número de dispositivos y un número de procesos considerablemente grande, la posibilidad de la ejecución concurrente/paralela de estos procesos y la posibilidad de acceder de manera concurrente a los dispositivos aumenta el rendimiento del sistema de manera significativa.

Todo este maremágnum de procesos y recursos a los que se accede a la vez de manera compartida o exclusiva puede generar fácilmente situaciones de interbloqueo (*deadlock*) que hacen que el sistema deje de hacer un trabajo útil.

Un **interbloqueo** (*deadlock*) es una situación en la que un grupo de procesos están indefinidamente bloqueados sin ninguna posibilidad de que continúe su ejecución. En términos generales, la causa de este bloqueo indefinido es que cada proceso del grupo ha adquirido un conjunto de recursos necesarios para operar y está esperando otros recursos que han sido asignados a otros procesos del grupo. Ello crea una situación en la que ningún proceso puede continuar su ejecución.

Los interbloqueos habitualmente aparecen en sistemas concurrentes como consecuencia de la asignación de recursos del sistema de manera incontrolada, es decir, sin mirar o establecer ningún protocolo que intente evitar la situación. Muchas veces, es el mismo programador de los procesos el que crea esta situación de manera involuntaria.

Supongamos que un sistema tiene un dispositivo impresora y un dispositivo disco a los que se debe acceder en exclusión mutua. Se crean dos semáforos binarios, disco e impresora, inicializados en 1 para indicar que los recursos están libres. En el sistema, creamos dos procesos que se ejecutan de manera concurrente: P0 (figura 42) y P1 (figura 43).

```
...
sem_wait(disco)
sem_wait(impresora)
/*Utilizar el disco y la impresora*/
...
sem_signal(impresora)
sem_signal(disco)
...
```

Figura 42. Código del proceso `P0`

```
...
sem_wait(impresora)
sem_wait(disco)
/*Utilizar el disco y la impresora*/
...
sem_signal(disco)
sem_signal(impresora)
...
```

Figura 43. Código del proceso `P1`

Podemos observar que la ejecución secuencial de estos dos procesos es totalmente correcta y no supone ningún error. En cambio, a la hora de ejecutarse de manera concurrente, y según cómo se entrelacen las operaciones de los dos procesos, se puede generar una situación de interbloqueo. Por ejemplo, si se produce la secuencia siguiente:

1) El proceso `P0` pide el recurso disco y se le concede, `sem_wait` (`disco`).

2) El proceso `P1` pide el recurso `impresora` y se le concede, `sem_wait` (`impresora`).

3) El proceso `P1` pide el recurso `disco` y se queda bloqueado, `sem_wait` (`disco`), ya que el disco ha sido asignado.

4) El proceso `P0` pide el recurso impresora y se queda bloqueado, `sem_wait` (`impresora`), ya que este recurso ha sido asignado al proceso `P1`.

A partir de este momento, el proceso `P1` espera que se libere el recurso `disco`, pero éste está asignado al proceso `P0`, que no lo puede liberar porque está bloqueado esperando el recurso `impresora`. Éste tampoco será liberado, ya que el proceso que lo tiene asignado, el proceso `P1`, también está bloqueado. En definitiva, se ha creado un bucle de espera del que no hay salida.

Para que se produzca un interbloqueo, se deben dar las cuatro condiciones siguientes simultáneamente:

- **Exclusión mutua**: se debe acceder a los recursos en exclusión mutua.

- **Retención y espera**: cada proceso tiene asignados ciertos recursos en exclusión mutua y espera que se liberen otros.

- **No expropiación**: los recursos que un proceso tiene asignados tan sólo se liberan a petición explícita del propio proceso, el sistema operativo no los puede tomar.

- **Espera circular**: los procesos bloqueados forman una cadena circular en la que cada proceso está bloqueado esperando un recurso que ya ha sido asignado a otro proceso de la cadena.

Se han propuesto varios mecanismos para tratar de solucionar este problema. Las diferentes propuestas se pueden clasificar en tres categorías según su objetivo:

1) Prevenir el interbloqueo: estas propuestas pretenden asegurar que una de las cuatro condiciones necesarias para el interbloqueo no se producirá nunca. Lo más sencillo es evitar la espera circular ordenando de manera lineal los recursos del sistema, a fin de que los procesos estén obligados a pedir los recursos que necesiten en un orden creciente dentro de la ordenación especificada. En el ejemplo anterior, se podría haber prevenido el interbloqueo si todos los procesos hubieran pedido los recursos en el mismo orden (por ejemplo, primero el disco y después la impresora).

2) Evitar el interbloqueo: estas propuestas se basan en el hecho de asignar sólo los recursos disponibles que no puedan generar ningún tipo de interbloqueo. Esta técnica necesita llevar un control detallado de qué recursos se han asignado, a qué procesos se han asignado y si estos procesos están esperando algún otro recurso.

3) Detectar y recuperar el interbloqueo: las propuestas que utilizan esta técnica dejan que aparezcan interbloqueos, ya que asignan libremente los recursos según los van pidiendo los procesos. De vez en cuando, se comprueba si se ha producido un interbloqueo mirando si hay algún ciclo en el grafo de recursos asignados. Así se determina qué procesos están bloqueados. Para romper el interbloqueo, el sistema operativo reinicia algunos de los procesos que estaban bloqueados.

Resumen

En este módulo didáctico hemos estudiado con detenimiento el problema de la sincronización de diferentes procesos y hemos ofrecido una idea de cómo se solucionan los problemas de comunicación de procesos y los interbloqueos (*deadlocks*).

Hemos visto que en la ejecución concurrente de procesos la **sincronización** es necesaria cuando éstos acceden a recursos compartidos (principalmente, dispositivos y variables compartidas). Sin una sincronización adecuada, la ejecución concurrente, al entrelazar la ejecución de varios procesos, puede producir errores de temporización y puede dejar el sistema inconsistente. Las causas de estas irregularidades dependen del recurso compartido. Así, tenemos las causas siguientes:

- En el caso de variables compartidas, la principal causa de este problema es la existencia de copias temporales por parte de los procesos concurrentes.

- En el caso de los dispositivos compartidos, una sincronización incorrecta puede provocar problemas de interbloqueo.

Para solucionar los problemas que surgen en el momento de sincronizar los procesos se han estudiado con detalle los **semáforos**, un mecanismo que permite la sincronización de manera sencilla. El principal problema que presentan es que el programador es el responsable de su buena utilización. Los programadores tienen que hacer un buen uso para evitar problemas de *deadlock* y de inanición.

También hemos visto el mecanismo de paso de mensajes, que, además de permitir la sincronización, también posibilita la comunicación de información entre procesos cooperativos y el mecanismo de las señales software. Finalmente, hemos descrito el problema del interbloqueo de procesos (*deadlock*).

Actividades

1. ¿Por qué interesa que las zonas de exclusión sean cuanto más pequeñas mejor? Proponed un ejemplo en el que se vean claras las posibles ventajas.

2. ¿Es necesario acceder en exclusión mutua a una variable compartida si tan sólo se quiere consultar su valor? ¿Por qué?

3. Demostrad que si las operaciones `sem_wait` y `sem_signal` no se ejecutan atómicamente se puede violar la condición de exclusión mutua.

4. Implementad un semáforo *n*-ario utilizando mensajes indirectos con buzones.

5. Considerad la comunicación entre procesos utilizando el esquema de los buzones. Considerad buzones infinitos (*send* no bloquea nunca).

a) Suponed que un proceso quiere esperar un mensaje del buzón-A y un mensaje del buzón-B (un mensaje de cada buzón). ¿Qué secuencia de *sends* y *receives* debería ejecutar?

b) ¿Qué secuencia de *sends* y *receives* se debería ejecutar si el mismo proceso quiere esperar un mensaje de un buzón o un mensaje del otro, o de buzón-A o de buzón-B?

6. Escribid la serie de llamadas al sistema que ha de invocar el intérprete de comandos para dar servicio a la línea de comandos `ps | grep sh | wc -1`.

7. Cuando se hace el *shut-down* de una máquina con el sistema operativo Linux, el SO genera la señal `SIGTERM` a todos los procesos que quedan en ejecución, espera unos segundos y, a continuación, genera la señal `SIGKILL` sobre todos los procesos que quedan en ejecución. ¿Por qué creéis que procede de este modo y no genera directamente la señal `SIGKILL`?

8. Trabajando sobre un terminal en Linux, pulsar las teclas Ctrl-C suele provocar la muerte del proceso en ejecución. ¿Podéis explicar los acontecimientos que se producen desde que el usuario pulsa Ctrl-C hasta que el proceso muere?

9. Suponed que un proceso Unix muere debido al depósito de una señal. ¿Su proceso padre puede saber qué señal ha provocado la muerte del hijo? Para responder podéis consultar la página del manual de la llamada al sistema `wait`.

10. Buscad información sobre qué características del estándar de señales POSIX se pueden cambiar utilizando el campo `sa_flags` de la estructura de datos `struct sigaction` de la llamada al sistema `sigaction`.

11. Escribid un programa que calcule cuál es la medida interna de las *pipes*. Podéis utilizar señales.

Ejercicios de autoevaluación

1. Os proponemos que creéis un generador de números aleatorios. Los números se irán obteniendo de un vector de memoria intermedia (*buffer*) circular de medida `maxbuff` en la que diferentes flujos irán insertando y extrayendo números siguiendo el esquema del productor/consumidor. El sistema se compone de los tres tipos de flujos siguientes:

- El flujo productor, que se encarga de crear números y los va introduciendo en un vector de memoria intermedia con una política de asignación FIFO compartida por todos los flujos. Como este vector tiene una medida finita, los flujos productores se bloquearán cuando esté lleno.
- El flujo aleatorizador, que se encarga de consumir los números. Lo hace tomando los cinco números más antiguos del vector de memoria intermedia (si no hay cinco, se espera a que estén), aplica la función `int processa(X1...,X5)` y obtiene otro número como resultado. Este número es introducido otra vez en el vector de memoria intermedia como si este proceso fuera un productor.
- El flujo consumidor, que se encarga de obtener los valores aleatorios y de darlos a quien se los pida. Para hacerlo, en el instante en el que necesita un número, mira el valor de un número cualquiera de los que hay en el vector de memoria intermedia sin extraerlo.

Pedimos el código esquemático de generador de números aleatorios poniendo énfasis en las partes de sincronización y exclusión mutua de estos tres tipos de flujos. La solución debe tener en cuenta que podría haber diferentes flujos de cada tipo.

2. Tenemos un vector de memoria intermedia circular de medida *N* que contiene elementos de un cierto tipo y que permite comunicar diferentes procesos entre ellos. Algunos de los procesos –los productores– escriben elementos mientras el vector de memoria intermedia no esté lleno, y los otros –los consumidores– leen elementos de este vector de memoria mientras no esté vacío.

Queremos introducir el concepto de prioridad entre los productores y los consumidores. Cada proceso productor y cada proceso consumidor tendrá una prioridad asociada entre `0` (la menos prioritaria) y `P - 1` (la más prioritaria), donde `P` es el número de procesos que hay en ejecución. Querremos también que un proceso consumidor no acceda al vector de memoria intermedia mientras haya algún otro proceso consumidor con más prioridad que también quiera acceder a él. Análogamente, un proceso productor no debe acceder al vector de memoria intermedia mientras haya algún otro proceso productor con más prioridad que también quiera acceder a él.

Implementad la solución de manera que si damos permiso a un proceso para acceder al vector de memoria intermedia, el resto de procesos del mismo tipo que quieran acceder a él deberán esperar que éste lo libere, independientemente de la prioridad de los procesos que pidan permiso para acceder a éste.

Debéis resolver este problema utilizando tantos semáforos como necesitéis y también memoria compartida entre los procesos (variables compartidas). Disponéis de una función que devuelve la prioridad del proceso que la ejecuta denominada *int prioridad*, que devuelve un valor entre `0` y `P - 1`.

3. Simulad una horchatería en la que trabajan dos personas: un horchatero y un camarero. El primero se encarga de producir las horchatas y pasarlas al camarero para que las pueda vender. Los clientes, que pueden ser muy numerosos, cuando llegan a la horchatería piden un número variable de horchatas al camarero. Éste, una vez las ha servido, avisa al cliente para que las tome. Mientras no hay clientes, el horchatero continúa preparando horchatas y el camarero espera clientes nuevos.

```
/*Definiciones e inicialización de variables compartidas*/
#define true 1
int n_horchatas = 0;
semaphore sem1, sem2, sem3, sem4, sem5, sem6;

void camarero() {
int tmp, i;
while (true) {
   sem_wait(sem2);
   sem_wait(sem5);
   tmp = n_horchatas;
   n_horchatas = 0;
   sem_signal(sem5);
   for (i=0; i<tmp; i++) {
         sem_wait(sem1);
         servir_horchata();
}
   sem_signal(sem3);
}
}
```

Figura 44

```
void cliente() {
   ir_a_la_horchatería();
   sem_wait(sem4);
   sem_wait(sem5);
   n_horchatas =
        cuantas_horchatas();
   sem_signal(sem5);
   sem_signal(sem2);
   sem_wait(sem3);
   sem_signal(sem4);
   tomar_horchatas();
}
```

Figura 45

```
void horchatero() {
int preparadas = 0;
```

```
while (true) {
   preparar_horchata();
   sem_signal(sem1);
   sem_wait(sem6);
   preparadas++;
   sem_signal(sem6);
}
}
```

Figura 46

Ayudas: antes de empezar, leed bien todas las preguntas, os puede ayudar sustituir los nombres de los semáforos por otros más significativos. Las rutinas *servir_horchata*, *ir_a_la_horchatería*, *cuantas_horchatas*, *tomar_horchatas y preparar_horchata* no modifican ninguna variable ni ningún semáforo compartido.

a) ¿Con qué valor inicializaríais cada uno de los semáforos?

b) ¿Para qué se supone que sirven el semáforos `sem5` y `sem6`? ¿Son realmente necesarios estos semáforos? ¿Por qué?

c) ¿Qué información se guarda de manera implícita en el contador del semáforo sem1?

d) ¿Para qué sirve el semáforo `sem4`?

4. En el programa de la figura 30, ¿qué efecto tendría eliminar las dos invocaciones a la llamada al sistema `close` realizadas por el proceso padre justo antes de invocar las llamadas al sistema `wait`?

5. Indicad qué señales hay bloqueadas en los puntos de ejecución A, B, C, D, E, F, G, H e I.

```
void sigusr1(int signum)
{sigset_t mascara;
   /* C */
   sigemptyset(&mascara);
   sigaddset(&mascara, SIGINT); sigaddset(&mascara, SIGusr1);
   sigprocmask(SIG_BLOCK, &mascara, NULL);
   /* D*/
}
void sigusr2(int signum)
{ /* B */
   kill(getpid(), SIGUSR1);
}
void sigalrm(int signum)
{ /* H*/
}
main()
{sigset_t mascara;
   struct sigaction new;
   new.sa_handler = sigusr1; new.sa_flags = 0;
   sigemptyset(&new.sa_mask);sigaddset(&new.sa_mask, SIGALRM);
   sigaction(SIGUSR1, &new, NULL);
   new.sa_handler = sigalrm; sigemptyset(&new.sa_mask);
   sigaction(SIGALRM, &new, NULL);
   new.sa_handler = sigusr2;
   sigemptyset(&new.sa_mask);sigaddset(&new.sa_mask, SIGPIPE);
   sigaction(SIGUSR2, &new, NULL);
   /* A */
   kill(getpid(), SIGUSR2);
   /* E */
   sigemptyset(&mascara); sigaddset(&mascara, SIGALRM);
   sigprocmask(SIG_BLOCK, &mascara, NULL);
   /* F */
   sigfillset(&mascara); sigdelset(&mascara, SIGALRM);
   alarm(2);
    /* G */
   sigsuspend(&mascara);
   /* I */
}
```

6. Escribid un fragmento de código que espere la llegada de una señal de tipo `SIGUSR1` o `SIGUSR2`, pero que atienda cualquier otro tipo de señal que no esté bloqueada.

7. Escribid un fragmento de código que espere la llegada de una señal de tipo `SIGUSR1` y otro de tipo `SIGUSR2` (en cualquier orden), pero que atienda cualquier otro tipo de señal que no esté bloqueada.

Solucionario

Ejercicios de autoevaluación

1. El código propuesto para el generador de números aleatorios es el siguiente:

```
void Productor()
{for (;;)

 {n = crearNombre ();
  sem_wait(full); /*Esperar a que pueda poner un elemento*/
  sem_wait(mutexProd); /*Exclusión mutua entre productores*/
  /*(y aleatorizadores)*/
  buffer[in] = n;
  in = (in + 1) % maxbuff;
  sem_signal(mutexProd); /*Fin de exclusión*/
  sem_signal(empty); /*Indica nuevo elemento en el vector*/
  /*de memoria intermedia*/
 }
}

int Consumidor () /*Toma un número cualquiera del vector*/
/*de memoria intermedia*/
{Return buffer[out]; }

void Aleatorizador () /*SOLUCIÓN 1.0*/
{/*Variables locales*/
 int i,aux;
 int n[5];
  for (;;){ /*FALLA*/
   for (i = 0; i < 5; i++)
   sem_wait(empty); /*Espera a que haya 5 o más elementos*/
   sem_wait(mutexAleat); /*Exclusión entre aleatorizadores*/
  for (i = 0; i < 5; i++){
   n[i] = buffer [out];
   out = (out + 1) % maxbuff;
   sem_signal(full); /*FALLA*/
  } /*Fin de la exclusión. Ha extraído 5*/
  sem_signal(mutexAleat); /*Elementos consecutivos*/
  aux = procesa (n[0], n[1], n[2], n[3], n[4]);
  sem_wait(full); /*Espera a que haya sitio*/
  /*Exclusión mutua entre productores (y aleatorizadores)*/
  sem_wait(mutexProd); buffer[in] = aux;in = (in + 1) % maxbuff;
  sem_signal(mutexProd); /*Fin de exclusión*/
  sem_signal(empty); /*Elemento nuevo disponible*/
 }
}
```

Cabe observar que el aleatorizador puede traer problemas. Por ejemplo, si sólo hay un aleatorizador, mientras está procesando los últimos cinco números que ha extraído, los productores pueden llenar el vector de memoria intermedia –se quedarán todos bloqueados–, el aleatorizador hará un `sem_wait(full)` –también se bloqueará– y nadie consumirá elementos.

Para arreglar esto, sólo hay que conseguir que el aleatorizador se guarde el sitio del último elemento que extrae: se puede sustituir la línea de código.

```
sem_signal(full);/*FALLA*/
```

para

```
if (i < 4) sem_signal(full);/*NO FALLA*/
```

y eliminar el

```
sem_wait(full);/*Espera a que haya sitio*/
```

Por otra parte, si hay más de un aleatorizador, éstos se pueden repartir los elementos del vector de memoria intermedia sin que ninguno de ellos tome cinco –haga cinco `sem_wait(empty)`–, el vector de memoria intermedia se puede llenar y se quedaría muy bloqueado porque ningún aleatorizador sacaría elementos.

Para corregir estos dos problemas, proponemos la solución siguiente para el aleatorizador:

```
void Aleatorizador () /*SOLUCIÓN 1.1*/
{int i,aux; /*Declaración de variables locales*/
 int n[5];
 for (;;)
 {sem_wait(mutexAleat); /*Exclusión mutua entre aleatorizadores.*/
  /*Evita que algún otro aleatorizador pueda*/
  /*quitarle elementos*/
  for (i = 0; i < 5; i++)
  {Sem_wait(empty); /*Espera un elemento*/
   n[i] = buffer[out];
   out = (out + 1) % maxbuff;
   if (i < 4) sem_signal(full); /*NO FALLA*/
  } /*Fin de la exclusión. Ha extraído 5*/
  sem_signal(mutexAleat); /*Elementos consecutivos*/
  aux = procesa (n[0], n[1], n[2], n[3], n[4]);
  /*En este punto, SEGURO QUE TENEMOS SITIO PARA UN ELEMENTO*/
  sem_wait(mutexProd); /*Exclusión mutua entre*/
  /*Productores (y Aleatorizadores)*/
  buffer[in] = aux;
  in = (in + 1) % maxbuff;
  sem_signal(mutexProd); /*Fin de exclusión*/
  sem_signal(empty); /*Elemento nuevo disponible*/
 }
}
```

2. Tendremos las variables y las estructuras de datos siguientes en la memoria compartida (además de las necesarias para implementar el vector de memoria intermedia circular):

```
/*Variables que indican si hay algún productor o algún*/
/*consumidor que actualmente accede al vector de memoria*/
/*intermedia. Inicialmente no hay ninguno.*/
int cap_prod_accediendo = 1;
int cap_cons_accediendo = 1;

/*Vectores que indicarán el número de productores y de consumidores*/
/*que están bloqueados esperando acceder al vector de memoria*/
/*intermedia. Todas las posiciones deben estar inicializadas en 0.*/
int prod_esperando[P];
int cons_esperando[P];

/*Vectores de semáforos en los que estarán bloqueados por prioridades*/
/*los consumidores y los productores. Los semáforos han de estar*/
/*inicializados en 0. Tendremos 2 semáforos para cada prioridad.*/
semaphore prod_semaforos[P];
semaphore cons_semaforos[P];

/*Semáforos de exclusión mutua productores/consumidores,*/
/*inicializados en 1.*/
semaphore prod_mutex, cons_mutex;

/*Semáforos de bloqueo en caso de vector de memoria*/
/*intermedia lleno o vacío*/
semaphore full; /*Inicializado en N*/
semaphore empty; /*Inicializado en 0*/

/*Productores*/
elem = producir();
sem_wait(prod_mutex);
if (cap_prod_accediendo)
{ cap_prod_accediendo = 0;
  sem_signal(prod_mutex);
}
else
{ prod_esperando[prioridad()]++;
  sem_signal(prod_mutex);
  sem_wait(prod_semaforos[prioridad()]);
}
sem_wait(full);
buffer[in] = elem; /*No es necesaria la exclusión porque sólo*/
in = (in + 1)% N; /*Puede acceder 1 productor*/
```

```
sem_signal(empty);
sem_wait(prod_mutex);
precio (i = P - 1; (i >= 0) && (prods_esperando[i] == 0);i--);
if (i < 0)
  cap_prod_accediendo = 1;
else
{ prod_esperando[i]-;
  sem_signal(prod_semaforos[i]);
}
sem_signal(prod_mutex);

/*Consumidores*/
sem_wait(cons_mutex);
if (cap_cons_accediendo)
{ cap_cons_accediendo = 0;
  sem_signal(cons_mutex);
}
else
{ cons_esperando[prioridad()]++;
  sem_signal(cons_mutex);
  sem_wait(cons_semaforos[prioridad()]);
}
sem_wait(empty);
elem = buffer[out];
out = (out + 1) % N;
sem_signal(full);
sem_wait(cons_mutex);
for (i = P - 1;(i >= 0) && (cons_esperando[i] == 0);i--);
if (i<0)
  cap_cons_accediendo = 1;
else
{ cons_esperando[i]-;
  sem_signal(cons_semaforos[i]);
}
sem_signal(cons_mutex);
consumir(elem);
```

3.a) Veamos el caso de cada semáforo:

- El `sem1` se debe inicializar en 0. Este semáforo sirve para sincronizar al horchatero y al camarero. Cada vez que el horchatero haya acabado de preparar una horchata, hará un `sem_signal` sobre `sem1` y el camarero la consumirá efectuando un `sem_wait`.
- El `sem2` se debe inicializar en 0. Este semáforo sincroniza al camarero con el cliente; cuando el cliente haya escrito su petición, hará un `sem_signal` sobre este semáforo para indicar al camarero que la sirva.
- El `sem3` se debe inicializar en 0. Este semáforo sincroniza al cliente con el camarero. Cuando todas las horchatas que ha pedido el cliente estén servidas, el camarero hará un `sem_signal` sobre `sem3` y el cliente podrá continuar ejecutándose.
- El `sem4` se debe inicializar en 1. Este semáforo regula un acceso en exclusión mutua (garantiza que no pueda haber diferentes clientes realizando peticiones al camarero simultáneamente).
- El `sem5` se debe inicializar en 1. Garantiza el acceso a la variable *cuantas_horchatas* en exclusión mutua.
- El `sem6` se debe inicializar en 1 porque también es un semáforo de exclusión mutua. Regula el acceso a la variable *preparadas*.

b) Los semáforos `sem5` y `sem6` sirven para implementar exclusiones mutuas.

El `sem5` no es necesario. Gracias al `sem2` y al `sem4` no se puede dar el caso de que en el mismo momento diferentes procesos accedan a *n*_horchatas.

El semáforo `sem6` tampoco es necesario, ya que controla una exclusión mutua sobre una variable (*preparadas*) que no es compartida.

c) El contador de `sem1` nos indica el número de horchatas preparadas y pendientes de ser servidas.

El horchatero lo incrementa cada vez que tiene una preparada y el camarero la decrementa cuando consume una.

d) El `sem4` sirve para tener exclusión mutua entre los clientes. Hasta que uno no está totalmente servido, los otros pueden indicar al camarero que se esperan.

4. Provocará que el proceso padre se bloquee indefinidamente en el segundo `wait` porque estará esperando la muerte de uno de sus hijos (el que ejecuta el comando *grep*), pero éste tampoco morirá porque estará bloqueado leyendo de una *pipe* vacía en la que existe un proceso escritor, el propio proceso padre. Por lo tanto, tenemos un *deadlock* en el que el padre está esperando que el hijo muera y el hijo está esperando que el padre cierre el canal de escritura sobre la *pipe*.

5. Las señales bloqueadas en cada punto son:

- A: ninguna
- B: usr2, pipe
- C: usr2, pipe, usr1, alrm
- D: usr2, pipe, usr1, alrm, int
- E: vacío
- F: alrm
- G: alrm
- H: todos
- I: alrm

6.

```
sigprocmask(SIG_BLOCK, NULL, &m);
sigaddset(&m, SIGUSR1); sigaddset(&m, SIGUSR2);
sigprocmask(SIG_BLOCK, &m, &old);
sigdelset(&m, SIGUSR1); sigdelset(&m, SIGUSR2);
while ((usr1rebut || usr2rebut) == FALSE)
  sigsuspend(&m);
sigprocmask(SIG_SETMASK, &old, NULL);
```

7.

```
sigprocmask(SIG_BLOCK, NULL, &m);
sigaddset(&m, SIGUSR1); sigaddset(&m, SIGUSR2);
sigprocmask(SIG_BLOCK, &m, &old);
sigdelset(&m, SIGUSR1); sigdelset(&m, SIGUSR2);
while ((usr1rebut && usr2rebut) == FALSE)
  sigsuspend(&m);
sigprocmask(SIG_SETMASK, &old, NULL);
```

Glosario

cambio de contexto *m* Manera de implementar la concurrencia de procesos en un sistema multiprogramado que implica dejar de ejecutar el proceso que estaba ocupando al procesador para pasar a ejecutar otro proceso.

deadlock *m* Bloqueo indefinido de dos o más procesos que esperan la liberación de algún recurso compartido del sistema que ya ha sido asignado.

espera activa *f* Consulta continuada por parte de un proceso del estado de un recurso, generalmente para determinar si está libre. Consume ciclos del procesador sin hacer ningún trabajo útil.

exclusión mutua *f* Acceso individualizado a un conjunto de recursos compartidos, de manera que si una operación se inicia no se puede iniciar otra hasta que no haya finalizado la primera.

ejecución concurrente *f* Ejecución entrelazada de diferentes procesos.

ejecución paralela *f* Ejecución simultánea de diferentes procesos. Sólo es posible en sistemas que dispongan de más de un procesador.

inhibir interrupciones *v* No permitir interrupciones de ningún tipo.

paso de mensajes *m* Herramienta que ofrecen algunos sistemas que permite sincronizar y comunicar procesos.

sección crítica *f* Conjunto de instrucciones en lenguaje máquina que se deben ejecutar en exclusión mutua porque modifican el estado de los recursos compartidos del sistema.

semáforos *m* Mecanismos de sincronización.

señal software *f* Herramienta que proporciona el sistema operativo con el objetivo de trasladar el mecanismo de las interrupciones al nivel de proceso. Igual que en el mecanismo de hardware de las interrupciones, las señales dan soporte a un amplio conjunto de situaciones diferentes que deben ser conocidas y atendidas con una cierta urgencia por parte de los procesos.

Bibliografía

Silberschatz, A.; Galvin. P. B.; Gagne G. (2008). *Operating Systems Concepts* (8.ª edición). John Wiley & Sons.

Tanembaum, A. (2009). *Modern Operating Systems*. Prentice-Hall.

Documentación disponible sobre llamadas en el sistema Unix en los recursos del aula.

Anexo

1. El soporte de hardware para la exclusión mutua

Hemos visto que la sincronización es necesaria para conseguir que el acceso a la sección crítica en exclusión mutua se lleve a cabo de manera correcta. También hemos visto que los semáforos son una herramienta bastante fácil de utilizar que soluciona el problema de la sincronización de manera elegante para cualquier número de procesos. Ahora bien, para implementar los semáforos es necesario asegurar la indivisibilidad en la ejecución del código de las operaciones `sem_wait` y `sem_signal`.

A continuación, presentamos soluciones de hardware para ejecutar el código de manera indivisible. Indirectamente ello nos permite la implementación de los semáforos.

1.1. Modo de ejecución privilegiado: la inhibición de las interrupciones

En la mayoría de los ordenadores hay instrucciones del lenguaje máquina que dan la posibilidad de inhibir (deshabilitar) y desinhibir (habilitar) las interrupciones. Con estas instrucciones, la implementación de la exclusión mutua es trivial.

Hemos visto que el problema de la exclusión mutua en general radica en la desasignación del procesador a un proceso en ejecución cuando éste está modificando una variable compartida, ya que entonces la actualización de esta variable queda a medio hacer. El procesador necesita el mecanismo de las interrupciones para implementar la multiplexación de procesos. Si un proceso desactiva las interrupciones, éste siempre tendrá asignado el procesador hasta que las vuelva a activar.

Con este funcionamiento, se puede garantizar la exclusión mutua de una sección crítica mediante el esquema siguiente (figura 47).

```
inhibir interrupciones    /* desactivar las interrupciones */
sección crítica
desinhibir interrupciones /* activar de nuevo las interrupciones */
```

Figura 47. Implementación de la entrada y salida de la exclusión mutua inhibiendo y desinhibiendo interrupciones

Estas instrucciones son muy peligrosas si se hace un uso indebido o malintencionado de ellas. Podrían llevar al sistema fácilmente al caos al no permitir la ejecución de ningún proceso. Por este motivo, interesa que estas instrucciones

no estén al alcance de los programadores de aplicaciones. Lo más habitual es que sean instrucciones privilegiadas del lenguaje máquina y puedan ser utilizadas únicamente por el sistema operativo.

En el caso de los semáforos, el sistema operativo puede utilizar estas instrucciones para asegurar la indivisibilidad en la ejecución de las llamadas al sistema `sem_wait` y `sem_signal`.

1.2. Modo de ejecución no privilegiado

A continuación se presentan dos instrucciones de lenguaje máquina no privilegiadas que pueden ser utilizadas para implementar el acceso en exclusión mutua a una región crítica. Por lo tanto, estas instrucciones pueden ser utilizadas directamente por los programas de usuario.

1.2.1 *Test and set*

La instrucción *test and set* está pensada para dar un soporte de hardware al problema del acceso a la zona crítica en exclusión mutua. Está diseñada expresamente para resolver conflictos de acceso a zonas críticas y asegurar que tan sólo un proceso pueda acceder a la vez a la sección crítica.

La idea básica es definir una variable de control (global para todos los procesos) asociada a la sección crítica[21] que determine si el acceso es posible o no. La variable se inicializa en `libre` e indica que el recurso compartido está disponible. Cada proceso que quiere acceder al recurso ejecuta la instrucción *test and set* (`TS`) y le pasa por parámetro la variable de control asociada al recurso.

[21] Recordad que la sección crítica es un trozo de código asociado a la utilización de un recurso compartido.

Esta operación se escribe `TS variable` y funciona comparando el valor de la variable con `ocupado`. Si está en `libre`, lo modifica a `ocupado`. Los bits de condición (*flags* de estado del procesador) se modifican para indicar el estado de la variable justo antes de ejecutar la operación `TS`.

Un semáforo binario que utilice espera activa podría implementar la operación utilizando TS de la manera siguiente (figura 48 y figura 49):

```
sem_wait: TS S
   BNF sem_wait
   return
```

Figura 48. Implementación de sem_wait utilizando *test_and_set*

```
sem_signal: mov S, libre
```

Figura 49. Implementación de sem_signal utilizando *test_and_set*

Supongamos que inicialmente `S` tiene el valor `libre`. Cuando el primer proceso ejecuta sem_wait sucede lo siguiente:

1) La instrucción `TS S` mira el estado de `S`, y como `S` está en `libre` lo pasa a `ocupado`. La instrucción `TS` modifica los *flags* para indicar que `S` estaba `libre`.

2) La instrucción `BNF` (*branch if not free*) salta a la etiqueta `sem_wait` si el valor de `S` estaba en `ocupado`. En nuestro caso, como está en `libre` no salta, el proceso vuelve de la llamada `sem_wait` y, por lo tanto, se le asigna el recurso compartido.

Si otros procesos quieren acceder al recurso, cuando ejecuten la operación `sem_wait` sucederá lo siguiente:

1) La instrucción `TS S` mira el estado de `S` y, como `S` está en `ocupado`, no se cambia la variable. La instrucción `TS` modifica los *flags* para indicar que `S` estaba `ocupado`.

2) La instrucción `BNF` salta a la etiqueta `sem_wait`, ya que el valor de `S` estaba en `ocupado`. De esta manera, todos los procesos se quedan ejecutando el bucle `sem_wait` a la espera de que alguien libere el recurso compartido. Para liberar el recurso compartido, lo único que se debe hacer es poner en `libre` el valor de `S`. El código de la operación `sem_signal` que lo puede hacer es el que mostramos en la figura 49.

Es importante señalar que el código de la instrucción `sem_wait` funciona porque la instrucción `TS` es indivisible o porque mientras el hardware ejecuta el microcódigo asociado a esta instrucción las interrupciones están inhibidas.

1.2.2. El intercambio (*swap*)

La instrucción *swap* intercambia el contenido de dos palabras de memoria atómicamente y se define tal como se indica en la figura 50.

```
swap (A,B){
   temp = A;
   EN = B;
   B = temp;
}
```

Figura 50. Definición de la instrucción de lenguaje máquina *swap*

Si el lenguaje máquina ofrece una instrucción de intercambio (*swap*), entonces la exclusión mutua se puede conseguir de la manera siguiente: definimos una variable global a la que denominamos `lock`, que puede tomar los valores

`cierto` o `falso`. La variable se inicializa en `falso`. Además, cada proceso tiene una variable local denominada clave, que puede tomar los valores `cierto` o `falso`.

Todo proceso que quiere acceder a una zona crítica en exclusión mutua debe ejecutar el código de la figura 51.

```
/* entrada sección crítica */
clave = cierto;
do {
   swap (lock, clave);
} until (clave == falso);
/* sección crítica */
...
/* salida sección crítica */
lock = falso;
```

Figura 51. Implementación de la entrada y la salida de la sección crítica utilizando la instrucción

2. Ejemplo: procesos productores y consumidores

En un sistema es habitual encontrar procesos con relaciones de productor-consumidor. En general, un **proceso productor** genera información que será consumida (tratada/manipulada) por el **proceso consumidor**.

El problema que se quiere resolver se podría formular en los términos siguientes: consideramos un grupo de procesos consumidores y productores que se están ejecutando de manera concurrente y que pueden producir y consumir a diferentes velocidades. Queremos proponer un protocolo de sincronización que permita ejecutar los procesos productores y consumidores de manera concurrente a sus respectivas velocidades y que los datos se consuman en el mismo orden en el que se han generado.

Todas las versiones propuestas para conseguirlo están basadas en semáforos. En concreto, se presentan las tres versiones que se indican a continuación:

1) Un productor y un consumidor con vector de memoria intermedia (*buffer*) ilimitado.

2) Diferentes productores y diferentes consumidores con vectores de memoria intermedia ilimitados.

3) Diferentes productores y diferentes consumidores con vectores de memoria intermedia limitados.

Para permitir a los procesos productores y consumidores trabajar de manera concurrente, supondremos que disponemos de vectores de memoria intermedia, espacios de memoria que los productores llenan con los datos generados y de los que los consumidores obtienen los datos que deben ser tratados.

2.1. Un productor y un consumidor con un vector de memoria intermedia ilimitada

En una primera versión, consideraremos que la medida del vector de memoria intermedia es ilimitada y que tenemos un único productor y un único consumidor. Una vez inicializado el sistema, el primer proceso que se debe poder ejecutar es el productor.

Cuando ya se han producido datos, el consumidor puede empezar a tratar la información. En el código que se propone a continuación, se utiliza un único semáforo, `producido`, inicializado en 0. Esto quiere decir que no es posible entrar en la sección crítica o, dicho de otra manera, si un proceso ejecuta la operación `sem_wait` sobre el semáforo producido, se bloqueará siempre que ningún otro proceso haya ejecutado la operación `sem_signal` sobre el semáforo producido antes.

En este primer caso, el **código propuesto para el proceso productor** es el de la figura 52 y el **código propuesto para el proceso consumidor** es el de la figura 53.

```
while (cierto){
   /* Producir */
   ...
   /* Dejar en el buffer */
   ...
   sem_signal(producido);
   /* Otras operaciones */
   ...
}
```

Figura 52. Código de los productores (versión 1)

```
while (cierto){
   sem_wait(producido);
   /* Leer del buffer */
   ...
   /* Consumir */
   ...
   /* Otras operaciones */
   ...
}
```

Figura 53. Código de los consumidores (versión 1)

El semáforo `producido` lleva la cuenta del número de datos producidos y todavía no consumidos. De hecho, tal como se ha especificado, el proceso productor no necesita pedir permiso para acceder a la variable compartida (vector de memoria intermedia) y lo único que debe hacer es señalar que ha dejado un elemento nuevo en el vector de memoria intermedia. El productor lo señaliza con la operación `sem_signal`.

El proceso consumidor, por otra parte, con el fin de que el funcionamiento sea correcto, se debe asegurar de que hay datos en el vector de memoria intermedia antes de intentar acceder a ella. La manera que tiene de saberlo es ejecutar la operación `sem_wait`.

Supongamos que el consumidor no tiene asignado el procesador y, por lo tanto, no se ejecuta durante un cierto tiempo. Mientras el productor va generando datos y metiéndolos en el vector de memoria intermedia para que el consumidor sepa cuántos datos debe procesar una vez vuelva a ejecutarse cada `sem_signal`, debe quedar reflejado en el semáforo `producido`. Es, pues, absolutamente necesario utilizar un semáforo *n*-ario[22] para tener constancia del número de señales y, por lo tanto, del número de datos pendientes de procesar que hay en el vector de memoria intermedia.

[22] El hecho de tener más de un dato hace inviable el uso de un semáforo binario.

El proceso consumidor, una vez tiene asignado el procesador, podrá entrar tantas veces en la sección crítica como datos haya en el vector de memoria intermedia. Es decir, la operación `sem_wait` no lo bloqueará hasta que el vector de memoria intermedia no se vacíe. Esta primera solución es muy sencilla, de hecho con ella no aparece ninguna sección crítica.

El vector de memoria intermedia puede considerarse un vector infinito con los dos punteros siguientes:

- Un puntero denominado *ent*, que indica cuál es la posición libre siguiente del vector de memoria intermedia.
- Otro puntero, denominado *sal*, que indica cuál es la primera posición del vector de memoria intermedia que tiene datos válidos pendientes de ser tratados.

Inicialmente los dos punteros son inicializados en 0. El productor tan sólo modifica el puntero `ent` al ejecutar la función *dejar_en_el_buffer*, y el consumidor modifica el puntero `sal` al ejecutar la función *leer_del_buffer*, de manera que no hay ningún tipo de conflicto.

2.2. Diferentes productores y diferentes consumidores con un vector de memoria intermedia ilimitada

En el caso de considerar la posibilidad de tener más de un productor y más de un consumidor, la cosa cambia un poco y no podemos utilizar el código que se ha mostrado antes. Todos los productores al ejecutar la función

dejar_en_el_buffer modifican una variable compartida: el puntero `ent`, y todos los consumidores al ejecutar la función *leer_del_buffer* modifican una variable compartida, el puntero `sal`.

A continuación, proponemos el código necesario para garantizar la ejecución concurrente en el caso de tener *N* procesos productores y *M* procesos consumidores. En este segundo caso, el **código propuesto para los procesos productores** es el de la figura 54 y el **código propuesto para los procesos consumidores** es el de la figura 55.

```
while (cierto){
   /* Producir */
   ...
   sem_wait(exclusion);
   /* Dejar en el buffer */
   ...
   sem_signal(exclusion);
   sem_signal(producido);
   /* Otras operaciones */
   ...
}
```

Figura 54. Código de los productores (versión 2)

```
while (cierto){
   sem_wait(producido);
   sem_wait(exclusion);
   /* Leer del buffer */
   ...
   sem_signal(exclusion);
   /* Consumir */
   ...
   /* Otras operaciones */
   ...
}
```

Figura 55. Código de los consumidores (versión 2)

El problema que plantea la gestión de tantos procesos se soluciona con un nuevo semáforo denominado `exclusion`[23] e inicializado en 1, para que el primer proceso que intente acceder a la sección crítica tenga la posibilidad de hacerlo. Como en el caso de antes, un consumidor no intentará entrar en la zona crítica hasta que algún productor no haya generado datos. En esta solución, también sería posible utilizar dos semáforos diferentes para asegurar la exclusión mutua en el acceso de los procesos al vector de memoria intermedia, uno para los consumidores y otro para los productores. Con dos semáforos, se conseguiría una mayor concurrencia entre productores y consumidores.

[23] El semáforo *exclusión* es binario. El semáforo *producido* es *n*-ario.

2.3. Diferentes productores y diferentes consumidores con un vector de memoria intermedia limitado

En la tercera versión, consideramos que tenemos un número de productores y consumidores ilimitado y, además, que la medida del vector de memoria intermedia es limitada, tiene una capacidad máxima. En este caso, el vector de memoria intermedia es un recurso compartido por todos los procesos y lo definimos como un vector, `buffer[0..capacidad - 1]`, de capacidad finita e igual a `capacidad`, más dos punteros, `ent` y `sal`, inicializados en 0. Los punteros indican cuál es la primera posición libre del vector de memoria intermedia (`ent`) y cuál es la primera posición de este vector que contiene información útil (`sal`). En este tercer caso, el **código propuesto para los procesos productores** es el de la figura 56 y el **código propuesto para los procesos consumidores** es el de la figura 57.

```
while (cierto){
   sem_wait(puedeproducir);
   /* Producir */
   pdata = producir();
   sem_wait(p_exclusion);
   /* Dejar en el buffer */
   buffer[ent] = pdata;
   ent = (ent + 1) mod capacidad;
   sem_signal(p_exclusion);
   sem_signal(puedeconsumir);
   /*Otras operaciones*/
   ...
}
```

Figura 56. Código de los productores (versión 3)

```
while(cierto){
   sem_wait(puedeconsumir);
   sem_wait(c_exclusion);
   /*Leer del buffer */
   cdata = buffer[sal];
   sal = (sal + 1) mod capacidad;
   sem_signal(c_exclusion);
   sem_signal(puedeproducir);
   /*Consumir*/
   ...
   /*Otras operaciones*/
   ...
}
```

Figura 57. Código de los consumidores (versión 3)

En la solución propuesta, se han definido cuatro semáforos:

- `c_exclusion` y `p_exclusion`, que son semáforos binarios inicializados en 1. Garantizan la exclusión mutua entre los productores que acceden a la variable `ent` y entre los consumidores que acceden a la variable `sal`.

- `puedeproducir`, que es un semáforo *n*-ario y se inicializa en la capacidad del vector de memoria intermedia, de manera que estamos permitiendo llenar el vector de memoria intermedia. Si ésta se llena, el siguiente productor que intente meter alguna cosa más se quedará bloqueado hasta que algún consumidor consuma datos, deje espacio libre en el vector de memoria intermedia y ejecute `sem_signal(puedeproducir)`.

- `puedeconsumir`, que es un semáforo *n*-ario inicializado en 0, ya que en un principio el vector de memoria intermedia está vacío y no hay nada que consumir. Los procesos productores cada vez que dejan alguna cosa en el vector de memoria intermedia lo indican con `sem_signal(puedeconsumir)`. El valor del semáforo `puedeconsumir` será como máximo el valor de capacidad, el número de señales (*signals*) en este semáforo que se pueden ejecutar antes de que el vector de memoria intermedia se llene.