2º de Grado en Ingeniería Informática
Escuela Politécnica Superior
Universidad de Córdoba
Departamento de Informática y Análisis Numérico

Sistemas operativos, apuntes de la asignatura.

Profesor: Juan Carlos Fernández Caballero
email: jfcaballero@uco.es

TEMA 3 – COMUNICACIÓN ENTRE PROCESOS

1.Índice de contenidos

1 Comunicación entre procesos, problemas de concurrencia

Dos procesos se dice que son **concurrentes** si hay una existencia simultánea de varios procesos en ejecución, aunque dicha existencia no implique una ejecución paralela.

En base al término concurrente se habla de paralelismo, por lo que de manera general se puede decir que el termino concurrencia da lugar a **dos tipos de paralelismo** respecto a la ejecución de procesos**:**

- **Pseudoparalelismo o concurrencia**: Los procesos se intercalan (multiplexan) en el tiempo en un solo procesador (con un solo núcleo).

- **Paralelismo o concurrencia real**: Los procesos se ejecutan de manera paralela en el mismo instante de tiempo, cada uno en un procesador diferente o dentro de un mismo procesador en diferentes núcleos.

Aunque puede parecer que la **intercalación** o "multiprogramación" y la **superposición** o "multiprocesamiento" de la ejecución de procesos presentan formas de ejecución distintas, se verá que ambas pueden contemplase como ejemplos de procesos "concurrentes" que presentan los mismos problemas cuando quieren hacer uso de recursos compartidos (compartiendo y compitiendo por el recurso).

A continuación se exponen una serie de **ejemplos** donde hay **problemas de concurrencia,** y que son **aplicables** tanto a **procesos** que comparten recursos como a **hilos**.

1.1 Ejemplo 1

Spooling en informática se refiere al proceso mediante el cual la computadora introduce trabajos en un *buffer,* un área especial en memoria o incluso en un disco, de manera que se pueda acceder a estos trabajos cuando el *buffer* esté listo.

Consideremos un ***spooler* o cola de impresión**. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un directorio de *spooler* especial. Otro proceso, el demonio de impresión, comprueba en forma periódica si hay archivos que deban imprimirse y si los hay, los imprime y luego elimina sus nombres del directorio.

Imagine también que nuestro directorio de *spooler* tiene una cantidad muy grande de ranuras, numeradas como 0, 1, 2, ..., cada una de ellas capaz de contener el nombre de un archivo. Imagine también que hay dos variables:

- ***sal***, que apunta al siguiente archivo a imprimir y se usa por parte del servicio o demonio de

impresión.

- ***ent***, que es una **variable compartida** que apunta a la siguiente ranura libre en el directorio, y que usan todos los procesos que deseen poner un archivo en el directorio de *spooler*.

Suponga que en cierto momento, las ranuras de la 0 a la 3 están vacías (ya se han impreso los archivos) y las ranuras de la 4 a la 6 están llenas (con los nombres de los archivos en la cola de impresión). De manera más o menos simultánea, los procesos A y B deciden que desean poner en cola un archivo para imprimirlo. Esta situación se muestra en la Figura 2-21.

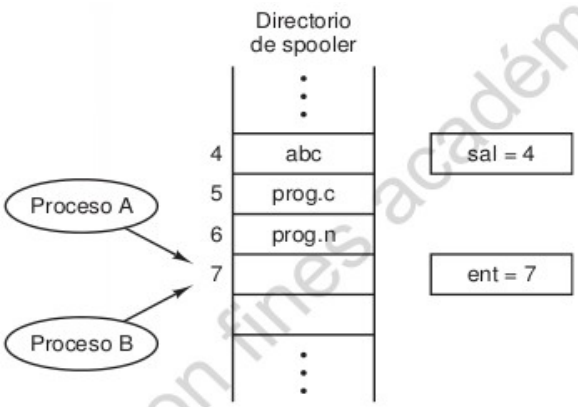


Figura 2-21. Dos procesos desean acceder a la memoria compartida al mismo tiempo.

En las jurisdicciones en las que se aplica la ley de Murphy (si algo tienen que salir mal, saldrá...) podría ocurrir lo siguiente:

- El proceso A lee ***ent*** y guarda el valor 7 en una variable local, llamada *siguiente_ranura_libre*.
- Justo entonces ocurre una interrupción de reloj y la CPU decide que el proceso A se ha ejecutado durante un tiempo suficiente, por lo que conmuta al proceso B.
- El proceso B también lee ***ent*** y también obtiene un 7. De igual forma lo almacena en su variable local *siguiente_ranura_libre*. Ahora, ambos procesos piensan que la siguiente ranura libre es la 7.
- Ahora el proceso B continúa su ejecución. Almacena el nombre de su archivo en la ranura 7 y actualiza ***ent*** para que sea 8. Después realiza otras tareas.
- En cierto momento el proceso A se ejecuta de nuevo, partiendo del lugar en el que se quedó. Busca en *siguiente_ranura_libre*, encuentra un 7 y escribe el nombre de su archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí. Luego calcula y fija ***ent*** para que sea 8.

Tras esta secuencia de procesamiento en CPU, el directorio de ***spooler*** es ahora internamente **consistente**, por lo que el **demonio de impresión no detectará nada incorrecto**, pero el **proceso B nunca recibirá ninguna salida**. El usuario B esperará en el cuarto de impresora por años, deseando con vehemencia obtener la salida que nunca llegará.

1.2 Ejemplo 2

Suponga un **sistema operativo multiprogramado** (hace cambios de contexto entre procesos pero no procesa a dos procesos de forma paralela) para un **monoprocesador** (aunque esto es extensible a varios procesadores).

A continuación se muestra un procedimiento se encarga de incrementar el valor de una variable global o variable compartida entre diferentes procesos o hilos **(global=0),** concretamente de dos procesos o hilos llamados P1 y P2.

El sistema puede saltar de un hilo a otro (también aplicable para procesos). Si un hilo lee la variable **global** y justo entonces es interrumpido, otro hilo puede alterar la variable antes de que el primer hilo pueda utilizar su valor, produciéndose una carrera.

```
//Función usada por dos hilos, P1 y P2.
void sumadora ()
{
        int loc = global; // Hilo P1 sale de la CPU tras leer el valor de "global". (1)
        loc++;            // Hilo P1 retomará esta linea después de que P2 se ejecute entero. (2)
        global = loc;     // (3)
}

/*Se han usado estas tres sentencias para simular que la operación global++ podría conllevar varias instrucciones a
nivel ensamblador y que un hilo puede salir de la CPU al terminar cualquiera de esas instrucciones a más bajo nivel
*/

// Cuando P1 sale, P2 entra y ejecuta entero, con lo que global=1
//Posteriormente el hilo P1 se reanuda, incrementa loc++ (que valdría 1) y asigna su valor a global, por lo que
global=1.
//Por tanto, no se ha hecho el incremento correctamente, ya que global debería ser igual a 2
```


Considere la siguiente secuencia para dos hilos, P1 y P2:

1. La variable "**global**" está inicializada a 0. El hilo P1 invoca el procedimiento *sumadora()* y es interrumpido inmediatamente después de la sentencia (1). En este punto, el valor de "**global**" se ha asignado a una variable local "***loc***".

2. El hilo P2 se activa e invoca al procedimiento *sumadora()*, que ejecuta hasta concluir, habiendo leído e incrementado el valor de la variable "global", que tendrá el valor de 1.

3. Se retoma el proceso P1 y se ejecuta la sentencia (2). En este instante "*loc*" tiene el valor de 0, incrementa "*loc*" de forma que vale 1 y asigna a "global" el valor 1.

4. La variable global no se ha incrementado en 2 como cabría esperar...

1.3 Ejemplo 3, incoherencia de datos

Veamos un ejemplo de lo que se denomina **incoherencia de datos**. Considérese una aplicación de contabilidad en la que pueden ser actualizados varios datos individuales compartidos o accesibles desde diferentes funciones. Suponga que dos datos individuales *a* y *b* han de ser mantenidos en la **relación *a* = *b*.** Esto es, cualquier proceso o hilo que actualice un valor, debe también actualizar el otro para mantener la relación. Si el estado es inicialmente consistente (tanto *a* como *b* valen lo mismo), cada proceso o hilo tomado por separado dejará los datos compartidos en un estado consistente.

Si P1 se ejecuta entero y después P2, la **secuencia de ejecución P1.1, P1.2, P2.1 y P2.2** dejará el **directorio consistente:**

P1.1, P1.2, P2.1, P2.2

P1:	P2:
a = a + 1; //P1.1	*b = 2 * b; //P2.1*
b = b + 1; //P1.2	*a = 2 * a; //P2.2*

Ahora **considere la siguiente ejecución de pseudoparalelismo**, en la cual los dos procesos o hilos no actualizan *a* y *b* a la misma vez, pero que en un momento dado son interrumpidos y salen del procesador e intercalan sentencias de uno y otro sobre las variables que comparten, como por ejemplo en esta secuencia **P1.1, P2.1, P1.2, P2.2:**

P1.1, P2.1, P1.2, P2.2

a = a + 1; //P1.1
*b = 2 * b; //P2.1*
b = b + 1; //P1.2
*a = 2 * a; //P2.2*

Al final de la ejecución de esta secuencia, la condición ***a* = *b* ya no se mantiene**. Por ejemplo, si se comienza suponiendo que *a=b=1*, al final de la ejecución de esta secuencia, tendremos *a* = 4 y *b=3*. El problema se puede evitar declarando en cada proceso la secuencia completa de actualización de *a* y *b* como una sección crítica (se estudiará a continuación).

1.4 Conclusiones sobre los ejemplos, condiciones de carrera

En estos ejemplos se ha visto que es necesario proteger las variables y recursos compartidos, y que la única manera de hacerlo es **controlar el código que accede a esos recursos compartidos.**

Los problemas anteriores fueron enunciados con la suposición de que se ejecutaban en un sistema operativo multiprogramado para un **monoprocesador**. Por tanto, se muestra que los problemas de la concurrencia suceden incluso cuando hay un único procesador. El motivo principal es que una **interrupción** pare la ejecución de instrucciones en cualquier punto de un proceso.

En el caso de un sistema **multiprocesador**, se tienen los mismos motivos y, además, puede suceder porque dos procesos pueden estar ejecutando simultáneamente y ambos intentando acceder al mismo recurso compartido, sin embargo, la solución a ambos tipos de problema es la misma: **controlar los accesos a los recursos compartidos.**

Situaciones como ésta, donde dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, se conocen como **condiciones de carrera**. Depurar programas que contienen condiciones de carrera no es nada divertido, los resultados de la mayoría de las ejecuciones de prueba están bien, pero en algún momento poco frecuente ocurrirá algo extraño e inexplicable.

Pruebe el siguiente código de hebras en C, donde cada hebra incrementa 100.000 veces una variable global, y compare los resultados obtenidos en varias ejecuciones:

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

void * hilo (void *);
int global;  //También se podría haber creado la variable en el main() y pasar su dirección de memoria a las hebras

int main ()
{
        global = 0;
        int i;
        pthread_t hilos[2];

        for(i=0 ; i <2 ; i++)
                pthread_create(hilos+i, NULL, hilo, NULL); //Debería completarse con un control de error

        for(i=0 ; i < 2 ; i++)
                pthread_join(hilos[i],NULL); //Debería completarse con un control de error

        printf("Variable global = %d\n",global);
        exit(EXIT_SUCCESS);
}
void * hilo (void * nada)
{
        int i;
        for(i=0 ; i < 100000 ; i++)
                global++;
}
```

2 Sección critica y exclusión mutua

¿Cómo evitamos las condiciones de carrera? La clave para evitar problemas aquí y en muchas otras situaciones en las que se involucran la memoria o recursos compartidos, es buscar alguna manera de **prohibir que más de un proceso lea y escriba sobre dichos recursos al mismo tiempo** (o por multiplexaciones entre procesos dentro de un mono-procesador). Esa parte del programa en la que se accede a datos compartidos se conoce como región crítica o **sección crítica**. Por tanto, un proceso está en su sección critica cuando ejecuta código que accede a datos compartidos con otros procesos.

Dicho esto, lo que necesitamos es **exclusión mutua** (nunca más de un proceso ejecutando la sección crítica), es decir, cierta forma de asegurar que si un proceso está utilizando una variable o recurso compartido, los demás procesos se excluirán de hacer lo mismo, evitando la carrera.

```
/* PROCESO 1 */
void P1
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

```
/* PROCESO 2 */
void P2
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

• • •

```
/* PROCESO n */
void Pn
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

Figura 5.1. Ilustración de la exclusión mutua.

La **Figura 5.1 ilustra el mecanismo de exclusión mutua** en términos abstractos. Hay ***n* procesos para ser ejecutados** de manera concurrente, y cada proceso incluye:

1. Una **sección crítica** que opera sobre algún recurso compartido**.**
2. Un **código adicional que precede y sucede a la sección crítica** y que permite el acceso a ella o salida de la misma en función de un valor ***Ra*** que actúa a modo de semáforo.

Dado que todos los procesos acceden al mismo recurso compartido, se desea que sólo **un proceso esté en su sección crítica al mismo tiempo.**

Para aplicar exclusión mutua se proporcionan dos funciones, ***entrarcritica(Ra)*** y ***salircritica(Ra)*** con las siguientes responsabilidades respectivamente:

- Establecer los criterios o comprobaciones necesarias sobre el valor ***Ra*** para entrar en la sección crítica y **bloquearla mientras algún proceso esté en ella.** Por tanto, a cualquier proceso que intente entrar en su sección crítica mientras otro proceso está en su sección crítica, por el mismo recurso, se le hace esperar.
- **Desbloquear la sección crítica** cuando finalice su tratamiento y modificar el valor de ***Ra*** para dejarla libre.

Si volvemos al ejemplo del procedimiento ***sumadora()***, y teniendo en cuenta los mecanismos anteriores, que consistiría en introducir *entrarCritica()* y *salirCritica()* antes y después de la función *sumadora(),* el comportamiento de la solución resultaría como sigue:

1. El proceso P1 invoca el procedimiento *sumadora()* y es interrumpido inmediatamente después de que concluya la sentencia (1), por lo que pasa a estado Listo.
2. El proceso P2 se activa e invoca al procedimiento *sumadora()*. Sin embargo, dado que P1 está todavía dentro del procedimiento *sumadora()*, aunque actualmente en estado **Listo**, a P2 se le impide entrar en el procedimiento. Por tanto, P2 pasa a estado **Bloqueado** esperando la disponibilidad del procedimiento *sumadora()*.
3. En algún momento posterior, el proceso P1 pasa a estado **Ejecutando** y completa la ejecución de *sumadora()*. "global" tendrá el valor de 1.
4. Cuando P1 sale de *sumadora()*, esto elimina el bloqueo de P2, pasando a estado **Listo**. Cuando P2 sea más tarde retomado, invocará satisfactoriamente el procedimiento *sumadora()* y "global" valdrá 2.

```
entrarCritica(Ra)
    void sumadora ()
    {
        int loc = global; // (1)
        loc++;            // (2)
        global = loc;     // (3)
    }
salirCritica(Ra)
```

ó

```
void sumadora ()
{
    entrarCritica(Ra)
        int loc = global; // (1)
        loc++;            // (2)
        global = loc;     // (3)
    salirCritica(Ra)
}
```

Por tanto, el **comportamiento que deseamos tener** se muestra en la siguiente Figura:

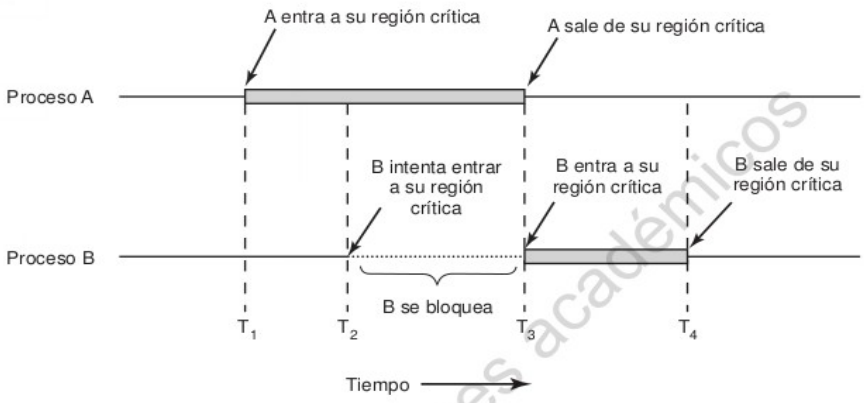


Figura 2-22. Exclusión mutua mediante el uso de regiones críticas.

3 Interbloqueo

Se puede definir el interbloqueo como el **bloqueo permanente de un conjunto de procesos** que compiten por recursos del sistema. **N**ingún proceso que se ejecute fuera de su región crítica debería bloquear otros procesos.

Un conjunto de procesos está interbloqueado cuando cada proceso del conjunto está esperando un evento, normalmente la **liberación de algún recurso requerido, que sólo puede generar otro proceso bloqueado del conjunto**. En casos de interbloqueos poniendo como ejemplo solo dos procesos, un proceso A está bloqueado esperando por un recurso que tiene otro proceso B, que a su vez está bloqueado a la espera de un recurso que tiene el proceso A.

Considere este ejemplo con dos procesos, **P1 y P2**, y dos recursos, una **impresora** y un **escaner**. Si se diera la secuencia de ejecución (1), (2), (3) y (4) se produciría un interbloqueo, ya que hasta que no se usen la impresora y el escáner no se podrán liberar, y para ello P1 y P2 tendrían que continuar por las lineas en color rojo, cosa que no harán al estar bloqueados esperando el recurso que necesita cada uno de ellos:

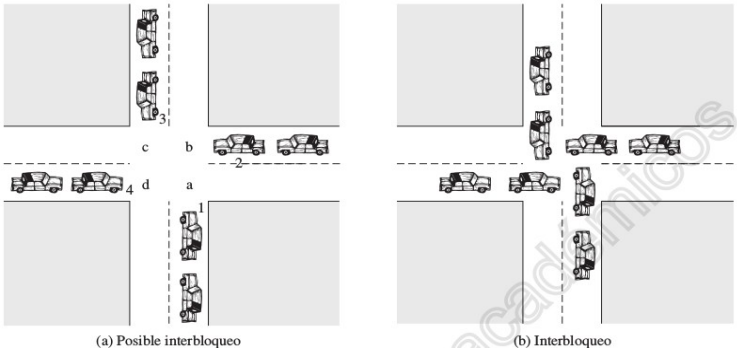
P1	**P2**
(1) - Pide (escáner)	*(2) - Pide (impresora)*
(3) - Pide (impresora)	*(4) - Pide (escaner)*
Usa impresora y escaner	***Usa impresora y escaner***
Libera (impresora)	***Libera (escaner)***
Libera (escaner)	***Libera (impresora)***

Es interesante resaltar dos aspectos que caracterizan a **este tipo de situación de interbloqueo**:

- Los procesos involucrados **piden los recursos en un orden diferente**. Ese orden viene determinado por las necesidades de cada proceso.
- Los procesos **requieren utilizar** en algún punto de su ejecución **varios recursos de carácter exclusivo simultáneamente.**

El responsable de tratar este problema es el sistema operativo, por ejemplo mediante algún algoritmo de **asignación y planificación de dispositivos de E/S**.

Existen algoritmos y **técnicas para prevenir y para detectar interbloqueos** entre procesos. Entre esos algoritmos existen algunos que permiten dar "**un paso atrás**", de manera que se pasaría al estado previo en el cual se produjo la problemática para intentar salvar el interbloqueo (marcha atrás de los vehículos en la siguiente figura).



Este tipo de técnicas son relativamente complejas, pero si está interesado en ellas puede consultar algunas de las más básicas en la bibliografía recomendada de la asignatura.

3.1 Interbloqueo entre hebras de la misma aplicación

Incluso utilizando los mecanismos de acceso en exclusión mutua a una sección crítica, se puede dar **interbloqueo**.

Considere el siguiente programa con 3 hebras que usan técnicas de tipo *entrarCritica()* y *salirCritica()* para poder sincronizarse:

Hilo P1	**Hilo P2**	**Hilo P3**
/* código anterior */	/* código anterior */	/* código anterior */
entrarCritica(Ma) //1)	*entrarCritica*(Mb) //2)	*entrarCritica*(Mc) //3)
//Sección crítica. Tarea1.1	//Sección crítica. Tarea2.1	//Sección crítica. Tarea3.1
entrarCritica(Mb) //5)	*entrarCritica*(Mc) //6)	*entrarCritica*(Ma) //4)
//Sección crítica. Tarea1.2	//Sección crítica. Tarea2.2	//Sección crítica. Tarea3.2
salirCritica(Ma)	*salirCritica*(Mb)	*salirCritica*(Mc)
//Sección crítica. Tarea1.3	//Sección crítica. Tarea2.3	//Sección crítica. Tarea3.3
salirCritica(Mb)	*salirCritica*(Mc)	*salirCritica*(Ma)
/* código posterior */	/* código posterior */	/* código posterior */

El siguiente orden de solicitud de recursos causaría un interbloqueo que afectaría a las 3 hebras:

1) P1: *entrarCritica*(Ma)
2) P2: *entrarCritica*(Mb)
3) P3: *entrarCritica*(Mc)
4) P3: *entrarCritica*(Ma) → se bloquea puesto que el recurso no está disponible, lo tiene P1.
5) P1: *entrarCritica*(Mb) → se bloquea puesto que el recurso no está disponible, lo tiene P2.
6) P2: *entrarCritica*(Mc) → se bloquea por recurso no disponible, lo tiene P3. → Hay **interbloqueo**.

Es importante resaltar en este ejemplo que el **hilo P1** necesita ejecutar tanto el código correspondiente a **Tarea1.1** como a la **Tarea1.2**, manteniendo todo ese tiempo la posesión del semáforo **Ma**. No sería válida, por tanto, una solución que intente eliminar el interbloqueo haciendo que P1 usase momentáneamente *salirCritica*(Ma) al finalizar "//Sección crítica. Tarea1.1" para ejecutar *entrarCritica*(Ma) inmediatamente justo antes de *entrarCritica*(Mb).

En este tipo de escenarios, **la responsabilidad final corresponde a los desarrolladores** o **implementadores** del programa: el diseño de la aplicación debe tenerlo en cuenta y su aparición es un error en el mismo.

4 Inaniciones por entrada a la sección crítica

Por otro lado, se debe evitar la **inanición** en cuanto a la imposibilidad o retraso para acceder a la sección crítica. Se da inanición cuando **uno o más procesos están esperando** (en estado Bloqueado por ejemplo) **recursos ocupados por otros procesos**, los cuales no se encuentran necesariamente en **ningún punto muerto** por requerir un recurso que no libera otro proceso.

Tenga como ejemplo la **utilización de prioridades** en muchos sistemas operativos multitarea, la cual podría causar que **procesos de alta prioridad estuvieran ejecutándose siempre**, y no permitieran la ejecución de procesos de baja prioridad, causando inanición en estos. En este caso, no se da el que cada uno de los procesos de un conjunto tenga un recurso que necesita alguno de ellos, sino que existe algún proceso que quiere acceder a ese recurso compartido pero que nunca se le da acceso al mismo.

Suponga tres procesos (**P1, P2, P3 –** P1 tiene mayor prioridad que P2 y éste a su vez mayor que P3) donde todos solicitan accesos periódicos en el tiempo a un **recurso *R* compartido**. Vamos a ver la situación en que dos de los procesos se multiplexan en el tiempo **dejando fuera al tercero con menor prioridad, al que se le deniega indefinidamente el acceso al recurso**, aunque no suceda un interbloqueo:

1. El proceso **P1 entra** en la SC del recurso R.
2. El proceso P2 solicita entrar en la SC del recurso R.
3. El proceso P3 solicita entrar en la SC del recurso R.
4. El proceso P1 abandona la SC del recurso R.
5. El proceso **P2 entra** en la SC del recurso R.
6. El proceso P1 solicita entrar en la SC del recurso R.
7. El proceso P2 abandona la SC del recurso R.
8. El proceso **P1 entra** en la SC del recurso R.
9. ...

Solución al problema anterior:

1) Para evitar estas situaciones los planificadores modernos incorporan algoritmos para asegurar que todos los procesos reciben un mínimo de tiempo de CPU para ejecutarse, aunque haya prioridades. Se verán ejemplos en el tema de Planificación.
2) Además de eso, hay mecanismos como el de los semáforos fuertes (usados en las prácticas), que se estudiarán en este tema. Esta técnica, por su implementación de cola interna, no permiten que se produzca inanición durante largos periodos de tiempo.

Un **caso especial de inanición** en procesos de alta prioridad se conoce como **inversión de prioridad**, que consistiría en que si un proceso de alta prioridad está pendiente del r**esultado o de un recurso de un proceso de baja prioridad que no se ejecuta nunca**, entonces este **proceso de alta prioridad** también **experimenta inanición**.

5 Soporte software para la exclusión mutua

Se han implementado diferentes técnicas de software para procesos concurrentes que se ejecutan en un único procesador o en una máquina multiprocesador con memoria principal compartida. De aquí hasta el final de este tema se estudiaran algunas de estas técnicas, pasando desde unas primeras aproximaciones o tentativas hasta llegar hasta los semáforos actuales.

Todas estas técnicas software pueden llegar a funcionar solo si se asume lo siguiente:

1. **Exclusión mutua elemental** a **nivel de acceso a memoria**.

   Es decir, se serializan accesos simultáneos (lectura y/o escritura) a la misma ubicación de memoria principal por alguna clase de árbitro de memoria (**controladora de memoria y de acceso a los buses del sistema**), aunque no se especifique por anticipado el orden de acceso resultante. Esto es un control hardware del que como programadores no debemos preocuparnos.

2. Los procesos involucrados en la sección crítica **no se caen del sistema mientras estén dentro de ella**.

   Si un proceso termina antes de salir de su sección crítica **las técnicas software no funcionan**, habría que implementar mecanismos mas complejos para su control y evitar bloqueos por esta razón, como el desbloqueo de semáforos generales por parte de otros procesos que no hayan producido su bloqueo. Si prueba a realizar una implementación como **práctica de la asignatura**, donde una hebra se quede bloqueada en la sección crítica o termine sin salir de ella (por ejemplo con *pthread_exit()*), verá que se produce un bloqueo de otras hebras que quieran acceder a dicha sección crítica. Lo mismo pasaría si con mecanismos IPC un proceso cayese dentro de la sección crítica de una zona de memoria compartida con otro proceso mediante el uso de semáforos.

   En los sistemas operativos actuales, **a nivel de núcleo**, se implementan mecanismos para que los procesos que están utilizando un recurso compartido del sistema y que terminan de manera abrupta cuando lo están usando, no lo bloqueen permanentemente y éste sea liberado para que lo usen otros procesos.

A continuación se muestran una serie de tentativas software que se utilizaron en las primeras investigaciones para resolver el problema de acceso en exclusión mutua a una sección crítica. Aunque ya están anticuadas, servirán para mostrar muchos de los errores típicos del desarrollo de programas concurrentes, además de para hacerle pensar de manera más autocrítica y más cuidadosa cuando requiera desarrollar este tipo de soluciones.

5.1 Algoritmo de Dekker

Edsger Dijkstra implementó un algoritmo de exclusión mutua para dos procesos, diseñado por el matemático holandés Dekker. Se hicieron varias versiones hasta llegar a un algoritmo final, que al tiempo fue sustituido por otro más simple creado por Peterson.

Todas estas tentativas tienen problemas de **espera activa** y de **bloqueos** si hay procesos que **caen del sistema** en su competición por acceder a la sección crítica.

5.1.1 Tentativa primera

En esta primera tentativa:

- Se reserva una ubicación de memoria compartida etiqueta como *turno=0*.
- Un proceso (*P0* ó *P1*) que desee ejecutar su sección crítica examina primero el contenido de *turno*.
- Si el valor de *turno* es igual al número del proceso, entonces el proceso puede acceder a su sección crítica.
- En caso contrario, si turno no es igual al número del proceso, está forzado a esperar.
- Después de que un proceso ha obtenido el acceso a su sección crítica y después de que ha completado dicha sección, debe **actualizar el valor de *turno*** (equivalencia a *SalirCrítica()*) para el otro proceso.

En términos formales, hay una **variable** compartida que **controla el acceso a la sección crítica**, que se **inicia a 0:**

A continuación se muestra el programa para dos procesos P0 y P1:

int turno = 0; //Compartida

Proceso 0	**Proceso 1**
… *while (turno = = 1);* *// No hacer nada* *//Sección crítica* *turno = 1; //Le da turno al Proceso 1* *…*	*…* *while (turno = = 0);* *// No hacer nada* *//Sección crítica* *turno = 0; //Le da turno al Proceso 0* *…*

Esta solución **garantiza la propiedad de exclusión mutua**, pero tiene las siguientes **desventajas**:

1. Los procesos deben **alternarse estrictamente** en el uso de su sección crítica; por tanto, **el ritmo de ejecución viene dictado por el proceso más lento**. Si *P0* utiliza su sección crítica sólo una vez por hora, pero *P1* desea utilizar su sección crítica a una ratio de 1000 veces por hora, *P1* está obligado a seguir el ritmo de *P0*.
2. Un problema mucho más serio es que **si un proceso cae antes de establecer el valor de *turno***, el otro proceso se encuentra **permanentemente bloqueado**, concretamente en espera activa.
3. Hay **espera activa** en el bucle *while()*. El término espera activa (***busy waiting***), o espera cíclica (***spin waiting***) se refiere a una técnica en la cual un proceso no puede hacer nada hasta obtener permiso para entrar en su sección crítica, pero continúa ejecutando una instrucción o conjunto de instrucciones que comprueban la variable apropiada para conseguir entrar. La espera activa consume ciclos de reloj sin hacer nada productivo.

   En esta tentativa, **el proceso que espera lee de forma repetida el valor de la variable *turno*** hasta que puede entrar en la sección crítica.

5.1.2 Tentativa segunda

El problema de la primera tentativa es que cada proceso debería tener su propia llave para entrar en la sección crítica al ritmo que desee. Para alcanzar este requisito, se realiza lo siguiente:

- Se define un vector compartido estado, con estado[0] para *P0* y estado[1] para *P1*.

  int estado[2]={0,0}; *//Cada elemento del vector que puede tener 2 valores: 0 ó 1*

- Cada proceso puede **examinar el estado del otro pero no alterarlo**, solo puede cambiar su propio valor de la variable *estado[i]*.
- Cuando un proceso desea entrar en su sección crítica, periódicamente **comprueba el estado del otro hasta que tenga el valor *0***, lo que indica que el otro proceso no se encuentra en su sección crítica.
- Si esto último es así, el proceso que está realizando la comprobación inmediatamente **establece su propio estado a *1* (va a entrar en la sección crítica)** y procede a acceder a su sección crítica.
- Cuando un proceso **deja su sección crítica**, establece su **estado a *0***.

El pseudocódigo que contempla esta metodología es el siguiente:

int estado[2]={0,0}; //Compartida

Proceso 0	**Proceso 1**
… *while (estado[1] == 1); //1. ¿Esta P1 en SC?* *// No hacer nada* *estado[0] = 1; //2. Estoy en SC* *//Sección crítica //3. SC* *estado[0] = 0; //4. No estoy en SC* *…*	*…* *while (estado[0] == 1); //5. ¿Esta P0 en SC?* *// No hacer nada* *estado[1] = 1; //6. Estoy en SC* *//Sección crítica //7. SC* *estado[1] = 0; //8. No estoy en SC* *…*

En esta tentativa, **se resuelve la alternancia**, un proceso puede entrar en su sección crítica tan frecuentemente como desee, pero se dan las siguientes **desventajas**:

- **No se garantiza la exclusión mutua** en todas las situaciones. Considérese la **secuencia 1-5-2-6** por multiplexaciones de los procesos en la CPU (o por multiprocesamiento). En esa secuencia ambos procesos estarían en condiciones de entrar en la sección crítica.
- Si un proceso **cae antes del código de establecimiento de estado a *0*,** es decir, *estado[i]= 0*, el otro proceso se queda **bloqueado**.
- Sigue habiendo **espera activa.**

5.1.3 Tentativa tercera

El problema del **acceso a la vez a la sección crítica** por parte de los dos procesos de la segunda tentativa se puede **arreglar** con un simple intercambio de dos sentencias, poniendo la línea de código ***estado[i]=1*** **antes** de la comprobación ***while()***:

int estado[2]={0,0}; //Compartida

Proceso 0	**Proceso 1**
… *estado[0] = 1; //1. Intención de entrar en SC* *while (estado[1] == 1); //2. ¿Esta P1 en SC?* *// No hacer nada* *//Sección crítica //3. SC* *estado[0] = 0; //4. No estoy en SC* *…*	*…* *estado[1] = 1; //5. Intención de entrar en SC* *while (estado[0] == 1); //6. ¿Esta P0 en SC?* *// No hacer nada* *//Sección crítica //7. SC* *estado[1] = 0; //8. No estoy en SC* *…*

Con este cambio puede comprobar que **se garantiza la exclusión mutua,** pero crea otro **nuevo problema:**

- Si ambos procesos establecen su estado a 1 antes de que se haya ejecutado la sentencia *while*, es decir, se da la **secuencia 1-5-2-6**, cada uno de los procesos pensará que está

ocupada la sección crítica, causando un **interbloqueo**.

- Además, al igual que en la tentativa anterior, si un proceso **cae antes del código de establecimiento de estado a *0*,** es decir, *estado[i]= 0*, el otro proceso se queda **bloqueado**.
- Y por último, sigue habiendo **espera activa.**

Se hicieron algunas tentativas más del algoritmo de Dekker hasta llegar a una solución final difícil de seguir en cuanto a implementación y entendimiento, pero no se van a mostrar en este tema, ya que con las tentativas propuestas se da una buena visión de la problemática de crear algoritmos para exclusión mutua. Si se mostrará a continuación la solución propuesta por Peterson, mucho más sencilla que la solución final propuesta por Dekker.

5.2 Algoritmo de Peterson

Como en el caso de Dekker, en el algoritmo de Peterson hay una **variable compartida *estado*** que indica la posición de cada proceso con respecto a la exclusión mutua:

- Se define un vector compartido estado, con estado[0] para P0 y estado[1] para P1.

  //Cada elemento del vector puede tener 2 valores: 0 ó 1

  int estado[2]={0,0}; *//Global*

- Se tiene por otro lado una **variable compartida *turno*** que resuelve conflictos simultáneos:

  int turno; *//Global*

Tome la variable ***turno*** como el siguiente **símil**:

"*Hola P1, soy P0 y deseo entrar en la sección crítica (estado[0]=1), pero soy muy gentil, de modo que si tu también deseas entrar (estado[1]=1) te voy a dejar pasar primero (turno=1)*".

Pseudocódigo del algoritmo de Peterson (se añade lo sombreado respecto a Dekker):

int estado[2]={0,0}; //Compartida

int turno; //Compartida

Proceso 0		**Proceso 1**	
…		…	
estado[0] = 1;	*//1*	*estado[1] = 1;*	*//6*
turno = 1;	*//2*	*turno = 0;*	*//7*
while (estado[1]==1 && turno==1);	*//3*	*while (estado[0]==1 && turno==0);*	*//8*
// No hacer nada		*// No hacer nada*	
//Sección crítica	*//4*	*//Sección crítica*	*//9*
estado[0] = 0;	*//5*	*estado[1] = 0;*	*//10*
…		…	

Esta solución tiene las siguientes **ventajas**:

- Garantiza la **exclusión mutua.**
- **Evita interbloqueos en el bucle while** con respecto a la solución anterior, porque ambos procesos puedan pensar a la misma vez que está ocupada la sección crítica.

  Este caso se resuelve porque llegará el momento en que, por ejemplo *P0,* ponga *turno = 1*, por lo que *P1* es capaz de entrar en su sección crítica (secuencia 1-6-2-7-3-8).
- **Evita** que los procesos puedan estar **utilizando su sección crítica repetidamente** (siempre que deseen hacerlo) y por tanto, monopolizando su acceso.

  Esto no puede suceder, porque por ejemplo, *P0* está obligado a dar una oportunidad a *P1* estableciendo el *turno = 1* (o viceversa), antes de cada intento por entrar a su sección crítica.

A pesar de resolverse las tentativas anteriores, la solución de Peterson sigue teniendo dos **problemas**:

- Si un proceso **cae antes del código de establecimiento de estado a *0*,** es decir, *estado[i]= 0*, el otro proceso se queda **bloqueado**.
- Sigue habiendo **espera activa.**

El algoritmo de Peterson se puede extender para más de dos procesos, si está interesado consulte la bibliografía recomendada y la Web

(https://es.wikipedia.org/wiki/Algoritmo_de_Peterson).

El problema de que un proceso caiga en determinadas zonas de su código o en la sección crítica es un caso difícilmente resoluble, pero el caso de **la espera activa si se puede resolver**, por ejemplo con **semáforos**, los cuales se estudiarán en este tema como técnicas software actuales.

6 Soporte hardware para la exclusión mutua

Existen también posibles soluciones a nivel de hardware para resolver problemas de concurrencia, pero también tienen sus inconvenientes como se verá a continuación.

6.1 Instrucción Test and Set (TSL)

A nivel hardware el acceso a una posición de memoria excluye cualquier otro acceso a la misma posición por más de un proceso a la vez. Con este fundamento, los diseñadores de procesadores han propuesto **instrucciones máquina que se pueden invocar desde código ensamblador** o mediante *wrappers*, y que llevan a cabo dos acciones **atómicamente**[1] (no está sujeta a interrupción), comprobar y escribir sobre una única posición de memoria con un **único ciclo de búsqueda-ejecución de instrucción**.

TSL son las siglas de ***Test and set lock***, una instrucción hardware utilizada por ciertos procesadores para facilitar la creación de semáforos y otras herramientas necesarias para la

1 El término atómico significa que la instrucción se realiza en un único paso y no puede ser interrumpida.

programación concurrente en computadores. TSL realiza dos acciones atomicamente:

1) Comprobar (leer) el contenido de una palabra de la memoria y
2) Almacenar (escribir) un valor distinto de cero en dicha palabra de memoria.

Dicho esto, el comportamiento de la instrucción *test and set* podría definirse como sigue:

```
boolean testset (int i)      //i=1 significa cerrojo cerrado; i=0 significa cerrojo abierto.
{
        if (i==0) //Si el cerrojo está abierto se podrá acceder a la SC.
        {
                i=1; //Se cierra el cerrojo.
                return true;
        }
        else       //Si el cerrojo está cerrado.
        {
                return false;
        }
}

// La instrucción TSL implementa una comprobación a cero del contenido de una variable en la
memoria, al mismo tiempo que varía su contenido en caso de que la comprobación se realizó
con el resultado verdadero.
```

A continuación se muestra una codificación en pseudocódigo, que habría que reproducir en lenguaje ensamblador o mediante algún *wrapper* que haga uso de TSL, sobre la resolución de exclusión mutua a sección crítica basado en el uso de esta instrucción.

```
/* Programa en pseudocódigo para exclusión mutua a SC, basado en la idea de TSL*/

int cerrojo;  //Variable compartida para control de acceso a la sección crítica.

void Proceso( )
{
        ...
        while ( test&set(cerrojo) == false );   //Se le pasa el valor de la variable cerrojo a
testset()
                        //No hacer nada, espera activa
            // Sección crítica;
           cerrojo=0;
        ...
}
```

```
void main()
{
    cerrojo = 0;
    Paralelos(P1, P2, P3, ...., PN);
}
```

Ejemplo en *C++* con *wrappers* usando la función *TSL* definida en el fichero de cabecera <*atomic*>. *C* no dispone de *wrapper*:

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic_flag
#include <thread>         // std::thread
#include <vector>         // std::vector
#include <sstream>        // std::stringstream
// atomic_flag as a spinning lock
std::atomic_flag   lock_stream = ATOMIC_FLAG_INIT;  //Esta sería la función "test&set()"
std::stringstream   stream;

void append_number(int x)
{
  while (lock_stream.test_and_set())   //Equivalente al while ( test&set(cerrojo) == false );
    {}  //Espera activa
  stream << "thread #" << x << '\n';      // SC. Es la salida estándar, es decir, la pantalla.
  lock_stream.clear();                    // Equivalente al cerrojo=0;
}

int main ()
{
  std::vector<std::thread> threads;

  for (int i=1; i<=10; ++i)
    threads.push_back(std::thread(append_number,i));

  for (auto& th : threads)
    th.join();

  std::cout << stream.str();
  return 0;
}

//Una posible salida podría ser la siguiente (el orden puede variar):
thread #3
thread #5
thread #1
```

```
thread #4
thread #2
thread #6
thread #7
thread #8
thread #9
thread #10
```

La solución hardware ***TSL*** seguiría teniendo los siguientes **problemas**:

- Hay **espera activa**.
- Si un proceso **cae antes del código de establecimiento de cerrojo a *0,*** el resto de los procesos se quedan **bloqueados**.
- **No hay orden de acceso a la sección crítica**, entrará aquel proceso que llegue a **ejecutar antes la instrucción *TSL,*** aunque hubiera un proceso previo que hubiera hecho el intenta anteriormente. Esto se verá que no ocurre en los **semáforos fuertes**, ya que el orden de ejecución es FIFO y viene marcado por la intención de acceder a la sección crítica, evitándose la inanición.

7 Semáforos binarios

El primer avance fundamental en el tratamiento de los problemas de programación concurrente ocurre en 1965 con lo que se denominan semáforos, que **resuelven la espera activa** de los algoritmos clásicos de Dekker y Peterson.

El principio fundamental de los semáforos se basa en lo siguiente:

Dos o más procesos pueden cooperar por medio de simples **señales**, tales que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica. Para esta señalización se pueden utilizan unas variables especiales llamadas **semáforos**.

En esta sección se especifican los semáforos binarios, una versión más restringida de lo que se verá después, los semáforos generales. A un semáforo binario se les conoce también como **mutex, o barrera.** En principio debería ser más fácil implementar un semáforo binario, y puede demostrarse que tiene la misma potencia expresiva que uno general cuando lo usamos solo para acceder en exclusión mutua a una sección crítica.

Los semáforos binarios también poseen una cola, una variable que puede tomar dos valores y dos primitivas atómicas. Concretamente, un semáforo binario **sólo puede tomar los valores 0 (cerrado) y 1 (abierto)** y se puede definir por las siguientes **tres operaciones**:

1. Un semáforo binario puede ser **inicializado a un valor 0** (cerrado) **o 1** (abierto)**.**
2. La operación ***semWaitB()*** **comprueba el valor del semáforo:**
   - Si el valor = ***1*** (abierto), entonces se **cambia el valor a = *0*,** y el proceso continúa su ejecución.

- Si el valor = ***0*** (cerrado), entonces el proceso que invoca esta operación se **bloquea**.

3. La operación ***semSignalB()*** **comprueba si hay algún proceso bloqueado** en el semáforo:
   - Si algún **proceso bloqueado se desbloquea** a uno de ellos.
   - **Si no hay procesos bloqueados**, entonces el valor del semáforo **se establece a = *1*** (abierto)**.**

A continuación se muestra una definición en pseudocódigo de las primitivas *semWaitB()* y *semSignalB()* para un semáforo binario inicializado a 1 (abierto):

```
struct semaforo_binario {
    int valor = 1; //Abierto
    tipoCola cola;
}

void semWaitB(semaforo_binario s)
{
    if (s.valor == 1)
        s.valor = 0;
    else
    {
        poner este proceso P en s.cola;   //Dependiendo del tipo de semaforo sera FIFO o no
        poner el proceso P en estado bloqueado;
    }
}

void semSignalB(semaforo_binario s)
{
    if (estaVacia(s.cola) == true)
        s.valor = 1;
    else
    {
        extraer un proceso P de s.cola;   //Dependiendo del tipo de semaforo sera FIFO o no
        poner el proceso P en estado Listo;
    }
}
```


Las acciones de comprobar el valor, modificarlo y pasar a bloqueado, se realizan en conjunto como una sola **acción atómica indivisible (se encarga de ello el núcleo del sistema operativo)**. Se garantiza que, una vez que empieza una operación *semWaitB()* o *semSignalB()*, ningún otro proceso podrá acceder al semáforo sino hasta que la operación se haya completado. Es decir, no se permite que se produzcan interrupciones mientras se ejecutan esas dos primitivas, al igual que ocurre con ***Test&Set()***.

8 Semáforos generales

A continuación se describen los **semáforos generales** o **semáforos con contador**, que además de proteger una sección crítica, tienen dos diferencias con respecto a los semáforos binarios:

1. Pueden **desbloquearse por cualquier otro hilo (o proceso)** que no lo haya bloqueado previamente (caso de los lectores-escritores que se estudiará después).
2. Permiten **sincronizar hilos (o procesos)** en determinadas situaciones o **condiciones** que los semáforos binarios no pueden controlar (caso de los productores-consumidores que se estudiará después).

Para conseguir el efecto deseado, el semáforo puede implementar una variable que contiene un valor entero, sobre el cual sólo están definidas **tres operaciones**:

1. Un semáforo puede ser **inicializado** a un **valor no negativo,** normalmente a **1**.
2. La operación ***semWait()*** **decrementa el valor del semáforo (*s.cuenta--*)**.

   Si el valor pasa a ser ***<0*** (negativo), entonces el proceso que ha invocado la operación se **bloquea**, no habiendo espera activa.

   En otro caso, el proceso continúa su ejecución.
3. La operación ***semSignal()*** **incrementa el valor del semáforo (*s.cuenta*++)**.

   Si el valor es ***<=0***, entonces se **desbloquea uno de los procesos bloqueados** previamente por la onvocación de una operación *semWait()*.

Igualmente se garantiza que una vez que empieza una operación *semWait()* o *semSignal()*, ningún otro proceso podrá acceder al semáforo sino hasta que la operación se haya completado al ser operaciones **atómicas**.

A continuación se muestra una definición en pseudocódigo de las primitivas *semWait()* y *semSignal()* de un semáforo general inicializado a 1 (abierto):

```
struct semaforo {
    int cuenta=1; //Abierto
    tipoCola cola;
}

void semWait(semaforo s)
{
    s.cuenta--;
    if (s.cuenta < 0)
    {
        poner este proceso P en s.cola;   //Dependiendo del tipo de semáforo sera FIFO o no
        poner el proceso P en estado bloqueado;
    }
}
```

```
void semSignal(semaforo s)
{
    s.cuenta++;
    if (s.cuenta <= 0)
    {
        extraer un proceso P de s.cola; //Dependiendo del tipo de semáforo sera FIFO o no
         poner el proceso P  en estado Listo;
    }
}
```

La siguiente figura muestra una posible secuencia de acceso en exclusión mutua a una SC de tres procesos (A, B, C). El semáforo tiene el valor 1 en el momento de su iniciación:

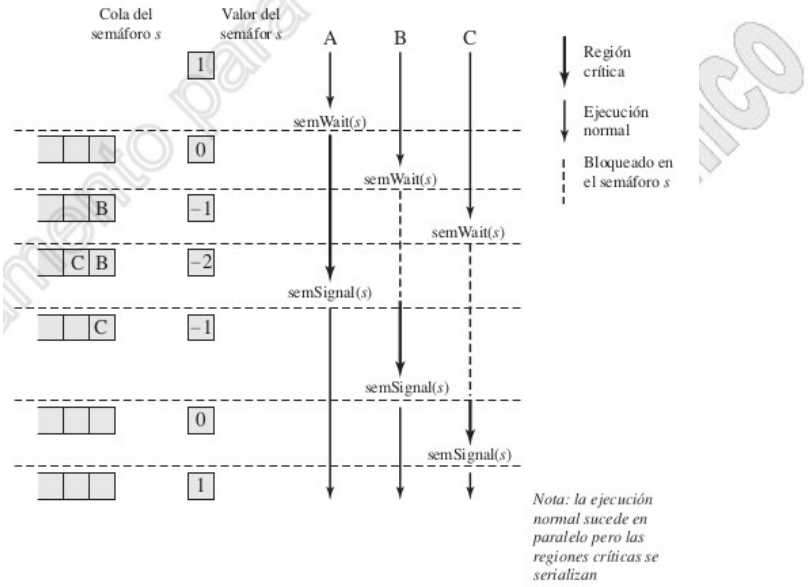


Figura 5.7. Procesos accediendo a datos compartidos protegidos por un semáforo.

- Cada proceso ejecuta un *semWait(s)* (el semáforo se inicializa a 1) justo antes de entrar en su sección crítica.
- El primer proceso que ejecute un *semWait()* será capaz de entrar en su sección crítica inmediatamente, poniendo el valor de *s* a 0.
- Cualquier otro proceso que intente entrar en su sección crítica la encontrará ocupada y se bloqueará, poniendo el valor de ***s =-1***.
- Cualquier número de procesos puede intentar entrar, de forma que cada intento insatisfactorio conllevará otro decremento del valor de ***s***.
- Cuando el proceso que inicialmente entró en su sección crítica salga de ella, ***s*** se incrementa y uno de los procesos bloqueados se extrae de la lista de procesos bloqueados asociada con el semáforo y se pone en estado Listo, pasando seguidamente a entrar en la sección crítica cuando el planificador a corto plazo lo ponga en estado Ejecutando.

Surge la cuestión sobre el **orden en que los procesos deben ser extraídos de la cola**:

- La política más favorable es FIFO (primero-en-entrar-primero-en-salir): El proceso que lleve más tiempo bloqueado es el primero en ser extraído de la cola. Un semáforo cuya definición incluye esta política se denomina **semáforo fuerte,** los cuales **garantizan estar libres de inanición** en cuanto a poder entrar en la sección crítica.
- Un semáforo que no especifica el orden en que los procesos son extraídos de la cola es un **semáforo débil**. Estos semáforos no garantizan la no inanición de procesos interesados en acceder a una sección crítica.

Si se necesita que se permita **más de un proceso en su sección crítica a la vez,** casos muy específicos y en los que el diseñador y programador deben tener especial cuidado, simplemente hay que **iniciar el semáforo** al valor correspondiente al **número de procesos** que podrán estar a la vez en la sección crítica**, en vez de inicializarlo a 1.**

9 Problema de los lectores-escritores

El problema de los lectores-escritores se describe como sigue:

- Hay un **objeto de datos compartido** (por ejemplo un fichero de texto, una base de datos, una variable, etc) que es utilizado por varios procesos, unos que leen (consultan) y otros que escriben (modifican).
- Solo puede **utilizar el objeto recurso por un proceso y solo uno**, es decir, o bien un proceso estará escribiendo o bien leyendo, pero nunca ocurrirá simultáneamente.

La solución de este problema se basa en implementar un algoritmo eficiente en el manejo de semáforos y memoria compartida, de manera que si consideramos el objecto como un fichero:

- **Cualquier número de lectores puedan leer** del fichero simultáneamente.
- **Solo un escritor** al tiempo **puede escribir** en el fichero.
- Si un escritor está escribiendo en el fichero ningún lector puede leerlo, es decir, **no se puede escribir y leer a la vez**.

Caso especial controlado por el semáforo general de este problema y que no podrían controlar los semáforos binarios a no ser que se implementase algún tipo de espera improductiva:

- Cualquier lector puede desbloquear la sección crítica siempre que sea el último.

A continuación se muestra una definición en pseudocódigo de la solución al problema de los lectores-escritores, **con prioridad a los lectores**, es decir, si hay lectores leyendo pueden seguir entrando en la sección crítica más lectores, aunque ya haya un escritor bloqueado porque haya intentado acceder antes que estos nuevos lectores en el tiempo a dicha sección crítica. Es el último lector el que dará acceso al escritor con *semSignal(sc)*:

```
/* Programa lectores-escritores */

int cuentalect = 0;       //Cuenta los lectores que hay en SC.
semaforo  x = 1;          //Semáforo general a 1 (o binario) para actualizar correctamente la variable cuentalect.
semaforo  sc = 1;         //Semáforo general a 1 para proteger la SC. El semSignal(sc) lo puede hacer cualquier
                            //lector que sea el último en salir, no tiene porque ser el que hizo el primer semWait(sc).

void lector()
{
    while (true)              //Simula que el lector puede querer leer de manera indefinida
    {
     semWait(x);              //Para protección de cuentalect.
     Cuentalect++;            ///Actualización de cuentalect. Se quiere leer.
     if(cuentalect == 1)      //Si soy el primer lector compruebo antes si hay ya un escritor en SC.
         semWait(sc);
     semSignal(x);            //Para protección de cuentalect.
     LeerDato();
     semWait(x);              //Para protección de cuentalect.
     cuentalect--;            //Actualiza cuentalect. Ya se ha leido.
     if(cuentalect == 0)      //Si soy el ultimo lector dejo posibilidad al escritor para acceso a SC.
         semSignal(sc);
     semSignal(x);            //Para protección de cuentalect.

    }
}

void escritor()
{
    while (true)        //Simula que el escritor puede querer escribir de manera indefinida
    {
     semWait(sc);       //Si hay un lector me bloqueo.
     escribirDato();
     semSignal(sc);
    }
}

//Aquí solo hay un escritor y un lector, pero podría haber varios escritores y lectores.
void main()
{
    paralelos(lector, escritor);
}
```

El proceso de resolución obtenido con el pseudocódigo anterior es el siguiente:

- El semáforo ***sc*** se utiliza para cumplir la **exclusión mutua.** Mientras un bibliotecario esté accediendo al área de datos compartidos, ningún otro bibliotecario y ni lector puede acceder.
- La variable compartida ***cuentalect*** se utiliza para llevar la cuenta del número de lectores.
- El semáforo ***x*** se usa para asegurar que ***cuentalect*** se actualiza adecuadamente.
- Para permitir múltiples lectores, necesitamos que cuando no hay lectores leyendo, el primer lector (*cuentalect == 1*) que intenta leer debe llamar a *semWait()* en ***sc***.
- Cuando ya haya al menos un lector leyendo, los siguientes lectores no necesitan hacer dicha llamada, ya pueden entrar en la sección crítica de manera directa.
- Cuando el último lector salga de la sección crítica (*cuentalect == 0*), la libera (*semSignal(sc)),* ya sea para que entre un escritor (podría haber alguno bloqueado) o nuevos lectores.

10 Problema del productor-consumidor

El problema del productor-consumidor también es conocido como el problema del almacenamiento limitado o acotado. El enunciado general es éste:

- Hay un proceso productor generando algún tipo de datos (registros, caracteres, etc) y poniéndolos en un buffer.
- Hay un proceso consumidor que está extrayendo datos de dicho buffer de uno en uno.
- El sistema está obligado a impedir la superposición de las operaciones sobre los datos, es decir, sólo un agente (productor o consumidor) puede acceder al buffer en un momento dado. Así el productor no sobrescribe un elemento que esté leyendo el consumidor, y viceversa.

La solución aportada necesita de **3 semáforos**, cuyas funciones son las siguientes:

1. Semáforo ***s***: Se utiliza para cumplir la exclusión mutua en el acceso al buffer, iniciado a 1.
2. Semáforo ***cn***: Sirve para el consumidor e indica el número de espacios ocupados en el *buffer* por la información que aún no se ha consumido. **Se inicializa a 0.**

   Además controla que el consumidor se bloquee **si no hay nada que consumir,** es decir, si ***cn.cuenta=0***.
3. Semáforo ***pr***: Sirve para el productor y cuenta el número de espacios vacíos o libres del *buffer*. **Se inicializa a *N***, siendo *N* el tamaño del *buffer*.

   Además controla que el productor se bloquee **si no tiene elementos o espacios libres** donde almacenar información, es decir, si ***pr.cuenta=0***.

Por tanto, **los casos especiales condicionados y controlados por los semáforos generales** de este problema, y que no podrían controlar los semáforos binarios a no ser que se implementase algún tipo de espera improductiva, son:

- **CASO CONDICIONANTE O EXTREMO 1:** En el caso de que el buffer esté completo, el productor debe esperar hasta que el consumidor lea al menos un elemento para así poder seguir almacenando datos en el buffer.
- **CASO CONDICIONANTE O EXTREMO 2:** En el caso de que el buffer esté vacío el consumidor debe esperar a que se escriba información nueva por parte del productor.

Aquí los semáforos se usan con dos objetivos diferentes, la EXCLUSIÓN MUTUA y SINCRONIZACIÓN bajo ciertas condiciones.

A continuación se muestra una definición en pseudocódigo de la solución al problema del productor-consumidor con *buffer* circular acotado:

```
/* Programa productor-consumidor */
semaforo s = 1;                          // Semáforo general a 1 (o  binario) para la sección crítica.
semaforo  cn = 0;                        // Semáforo general para el consumidor. Cuenta los espacios ocupados,
semaforo pr = /*tamaño del buffer*/      // Semáforo general para el productor. Cuenta los espacios libres.

void productor()
{
    while (true)          //Simula que el productor puede querer producir de manera indefinida
    {
      semWait(pr);        //Un espacio libre menos. Si pr.cuenta<0 me bloqueo.
      semWaits(s);
      añadir();           //Poner en el buffer (SC) un valor aleatorio generado.
      semSignal(s);
      semSignal(cn);      //Un espacio más ocupado por un dato generado.
    }
}

void consumidor()
{
    while (true)          //Simula que el consumidor puede querer consumir de manera indefinida
    {
      semWait(cn);        //Un espacio menos ocupado, si cn.cuenta<0 me bloqueo.
      semWaits(s);
      extraer();          //Leer del  buffer (SC) un valor.
      semSignal(s);
      semSignal(pr);      //Un espacio libre más.
    }
}

void main()
{
    paralelos(productor, consumidor);
}
```

```
/* Posible implementación del control del índice del buffer acotado en añadir() del PRODUCTOR
   int bufin=0;
   buffer[bufin]=aleatorioGenerado;
   buffin=(buffin+1)%TAM_BUFFER;

   Posible implementación del control del índice del buffer acotado en extraer() del CONSUMIDOR
   int buffout=0;
   datoLeido=buffer[buffout];
   buffout=(buffout+1)%TAM_BUFFER;

   Para un buffer de 5 elementos:  1%5 =1;  2%5=2;  3%5=3;  4%5=4;  5%5=0;  6%5=1;  ...*/
```

En el código del productor puede observarse que:

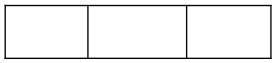
- En primer lugar se espera a que haya elementos libres en el *buffer,* es decir, que el semáforo ***pr*** sea no nulo. En este caso, el productor puede poner información en el *buffer*.
- En una segunda etapa, se comprueba si hay algún proceso que se encuentre en la sección crítica. Si es así, el productor quedará esperando; si no lo es, entrará en la sección crítica y pone la información producida en el *buffer*.
- La tercera operación consiste en incrementar el semáforo ***cn***, lo que indica que se ha introducido un nuevo elemento en el *buffer*.

En el código del consumidor puede observarse que:

- En primer lugar se espera a que haya elementos ocupados en el *buffer,* es decir, a que ***cn*** tenga un valor no nulo.
- En segundo lugar se realiza la comprobación sobre ***s*** para evitar problemas de exclusión mutua.
- En tercer lugar, tras entrar en la sección crítica, se actualiza el semáforo ***pr***.

Veamos una traza del problema para observar cómo se van actualizando los contadores de los semáforos ***cn*** y ***pr.*** Supongamos un *buffer* circular o acotado de 3 elementos:

1. Consumidor C1 va a consumir
   - cn.cuenta = -1
   - pr.cuenta = 3
   - C1 se queda en estado Bloqueado en *semWait(cn)* porque no hay nada que consumir.

2. Productor P1 va a producir
   - cn.cuenta = 0 //Parece que momentáneamente está desactualizado
   - pr.cuenta = 2

- C1 se desbloquea y pasa a estado Listo (vamos a suponer que no le dan tiempo de CPU).

3		

3. Productor P2 va a producir
   - cn.cuenta = 1 //Parece que momentáneamente está desactualizado
   - pr.cuenta = 1

3	-5	

4. Productor P3 va a producir
   - cn.cuenta = 2 //Parece que momentáneamente está desactualizado
   - pr.cuenta = 0

3	-5	7

5. Productor P4 va a producir
   - cn.cuenta = 2 //Parece que momentáneamente está desactualizado
   - pr.cuenta = -1
   - P4 se queda en estado Bloqueado en *semWait(pr)* porque está todo ocupado.

3	-5	7

6. Consumidor C1 pasa a estado Ejecutando (suponemos que el planificador ya le da CPU)
   - cn.cuenta = 2 //Ya está actualizada
   - pr.cuenta = 0 //Parece que momentáneamente está desactualizado
   - P4 pasa a estado Listo.

	-5	7

7. Productor P4 pasa a estado Ejecutando (suponemos que el planificador ya le da CPU)
   - cn.cuenta = 3
   - pr.cuenta = 0 //Está actualizado

10	-5	7

Las desactualizaciones momentáneas se dan cuando un productor o consumidor tienen **intención de producir o consumir**, pero, o se quedan bloqueados (casos extremos), o el planificador no les da tiempo de CPU antes de que puedan terminar la producción o consumición.

11 Señales como método IPC (*InterProcess Comunication*)

Las señales son otra manera de crear comunicaciones entre procesos (además de los semáforos, monitores, *pipelines* y colas de mensajes), por tanto se encuentran dentro de los diferentes mecanismos de la **comunicación inter-proceso o IPC** (*InterProcess Comunication*).

Son un mecanismo de comunicación **rápido y unidireccional**, por el cual se envía un **número** mediante **interrupciones de tipo software**. **El número y tipo de señales** viene **impuesto por el sistema operativo** y cada una de ellas será empleada en un caso concreto.

Las señales son **producidas:**

- **Directamente por el kernel** y tienen como finalidad **parar o desviar el curso normal de las instrucciones que se ejecutan en un determinado proceso** (por ejemplo división por cero, error de segmentación, etc)**.**
- Evidentemente, **un proceso** puede enviar una señal **a otro proceso**, pero eso se hace a través de una llamada ***wrapper*** al sistema (*kill()* en Posix) que desemboca en un **cambio a modo núcleo** para invocar a la primitiva nativa correspondiente.

Las señales, de manera general, se pueden **agrupar** de la siguiente manera:

- Señales relacionadas con la **terminación de procesos**, por ejemplo *SIGTERM* (terminación controlada de un proceso), *SIGKILL* en el caso de GNU/linux.
- Señales relacionadas con las **excepciones inducidas por los procesos.** Por ejemplo, el intento de acceder fuera del espacio de direcciones de usuario (*SIGBUS*), si se produce un error de acceso a memoria no reservada o fuera de rango (*SIGSEGV*), errores producidos al utilizar números en coma flotante o divisiones por cero (*SIGFPE)*.
- Señales relacionadas con los **errores irrecuperable**s originados en el transcurso de una **llamada al sistema**, por ejemplo *SIGSYS*, *SIGTRAP* en GNU/Linux.
- **Señales propias** originadas desde un proceso y e interpretadas por el programador. Por ejemplo *SIGUSR1* y *SIGUSR2* en GNU/Linux.
- Señales relacionadas con la **interacción con el termina**l. Por ejemplo *SIGINT* en GNU/Linux al pulsar las teclas *ctrl+c*.
- Señales relacionadas con **problemas de hardware**, por ejemplo las señales *SIGIOT* y *SIGEMT* de GNU/Linux.

Cuando un **proceso** en ejecución **recibe una señal** tiene que **interrumpirse**, se **carga la rutina ISR** correspondiente al tipo de señal recibida, que puede tratarla de tres formas diferentes, y se salva su **contexto**:

1. **Ignorar la señal** mediante la asociación de su recepción con un tratamiento que no tiene efecto (**ni tan siquiera su comportamiento por defecto**). Por ejemplo, en C con la llamada al sistema *signal(SIGINT, SIG_IGN)*.
2. Invocar a una rutina sin ese parámetro específico, la cual establezca en su cuerpo un

tratamiento personalizado de la señal, ignorando por tanto su comportamiento por defecto. A este tipo de rutinas se les conoce como ***callbacks*** o ***retrollamadas.***

3. **Realizar el tratamiento por defecto** correspondiente al número de señal. Esta rutina no la codifica el programador, sino que la aporta el kernel y normalmente tiene como fin el **terminar el proceso que recibe la señal**.

11.1 Señales en GNU/Linux

En sistemas GNU/Linux, cada señal tiene un nombre ***SIGXXX*** con un significado específico. En el fichero de cabecera ***signal.h*** están definidas todas las señales, número y nombre.

```
…
#define SIGHUP  1       /* hangup */
#define SIGINT  2       /* interrupt (rubout) */
#define SIGQUIT 3       /* quit (ASCII FS) */
#define SIGILL  4       /* illegal instruction (not reset when caught) */
#define SIGTRAP 5       /* trace trap (not reset when caught) */
#define SIGIOT  6       /* IOT instruction */
#define SIGABRT 6       /* used by abort, replace SIGIOT in the future */
#define SIGEMT  7       /* EMT instruction */
#define SIGFPE  8       /* floating point exception */
#define SIGKILL 9       /* kill (cannot be caught or ignored) */
#define SIGBUS  10      /* bus error */
#define SIGSEGV 11      /* segmentation violation *
…
```


Las más usuales, junto con su número identificador asociado, son:

- **SIGHUP (1) - *Hangup:*** Esta señal se envía a todos los procesos de un grupo cuando su líder de grupo termina su ejecución. Por ejemplo la envía una terminal cuando se cierra a todos los procesos que cuelgan de ella. La acción por defecto de esta señal es terminar la ejecución del proceso que la recibe.
- **SIGINT (2) - *Interrupción:*** Es enviada cuando en una terminal en medio de un proceso se pulsan las teclas de interrupción (*Ctrl* + *c*). Por defecto se termina la ejecución del proceso que recibe la señal.
- **SIGFPE (8) - *Error en coma flotante:*** Es enviada cuando el hardware detecta un error en coma flotante, como el uso de número en coma flotante con un formato desconocido, errores de *overflow* o *underflow*, división por cero. Por defecto se termina el proceso que la recibe.
- **SIGKILL (9) - *Kill:*** Esta señal provoca irremediablemente la terminación del proceso. No puede ser ignorada ni tampoco se puede modificar la rutina por defecto. Siempre que se recibe se ejecuta su acción por defecto, que consiste en terminar el proceso.
- **SIGSEGV (11) - *Violación de segmento:*** Es enviada a un proceso cuando intenta acceder a datos que se encuentran fuera de su segmento de datos. Su acción por defecto es terminar el

proceso.

- **SIGALRM (14) -** ***Alarm clock.*** Cada proceso tiene asignados un conjunto de temporizadores. Si se ha activado alguno de ellos y este llega a cero, se envía esta señal al proceso. Por defecto se termina el proceso.
- **SIGTERM (15) -** ***Finalización software controlada:*** Es la señal utilizada para indicarle a un proceso que debe terminar su ejecución. Esta señal no es tajante como SIGKILL y puede ser ignorada. Lo correcto es que la rutina de tratamiento de esta señal se encargue de tomar las acciones necesarias antes de terminar un proceso, como **por ejemplo**, **borrar los archivos temporales**, **conexiones**, **buffers**, etc y llame a la rutina *exit()*. Esta señal es enviada a todos los procesos cuando se produce una parada del sistema. Si no se sobreescribe con un *callback*, su **acción por defecto es terminar el proceso**.
- **SIGUSR1 (16) -** ***Señal número 1 de usuario.*** Esta señal está reservada para el usuario. Su interpretación dependerá del código desarrollado por el programador. Ninguna aplicación estándar va a utilizarla y su significado es el que le quiera dar el programador en su aplicación.
- **SIGUSR2 (17) -** ***Señal número 2 de usuario.*** Su significado depende de la interpretación del programador, como en el caso de SIGUSR1.

11.2 Envío de señales

Un proceso puede enviar señales a otro proceso mediante la función ***kill()***[2] de Posix. Esta función admite dos parámetros:

```
#include <signal.h>
void kill(pid_t pid, int sig)
```

- El primer parámetro ***pid_t*** es el identificador de proceso al que queremos enviar la señal. Si ponemos un número estrictamente mayor que **0**, se enviará al proceso cuyo ID de proceso coincida con el número.
- El segundo parámetro **sig** es el número o MACRO de la señal que queremos enviar.

Desde un terminal también podemos enviar una señal a un proceso mediante el **comando *kill*,** seguido de la macro correspondiente a la señal (definida en *signal.h*) a enviar y de su PID. Por ejemplo, el siguiente comando mandaría al proceso 3265 la señal *SIGKILL*, finalizándolo:

kill -SIGKILL 3265

La llamada al sistema **kill()**, ya sea a través de su programación en Posix o invocada desde un terminal (wrappers), llega al núcleo a través de ***system_call***, invocando a la función ***sys_kill***.

2 http://pubs.opengroup.org/onlinepubs/009695399/functions/kill.html

11.3 Asignación y recepción de señales

La función C que permite redefinir por parte del programador un tratamiento de señales es *signal()*[3]. Esta función se usa para capturar señales, y admite dos parámetros:

```
#include <signal.h>
void* signal(int sig, void (*func)(int))
```

- El primer parámetro ***sig*** es un número entero con el identificador o macro de la señal.
- El segundo parámetro ***func*** indica cómo se manejará la señal, es la función ***callback***.
  - Si el valor de *func* es **SIG_DFL**, se usará el manejo por defecto para esa señal. Se puede usar para sobreescribir una asignación anterior de una señal determinada a un *callback*, de forma que ya no se use dicho *callback*.
  - Si el valor de *func* es **SIG_IGN**, la señal será ignorada.
  - De lo contrario se apuntará a una función ***callback*** o **manejador de señal** que hay que implementar y que se llamará cuando la señal se active.

La estructura para crear y usar un manejador de señal es la siguiente:

```
void callback (int);
...
signal (SIGINT, callback);
```

A nivel de núcleo, después de invocar un *signal()* por parte de un proceso en ejecución, **si se recibe una señal** en un instante determinado ocurrirá lo siguiente:

1. Se interrumpe la ejecución del proceso.
2. Se ejecuta la función manejadora ISR.
3. Se salva el contexto del proceso, pasando a estado Listo.

3 http://pubs.opengroup.org/onlinepubs/009695399/functions/signal.html

4. Cuando el planificador lo pase a estado Ejecutando se comienza a ejecutar el proceso por la función *callback()*.
5. Cuando se termine la función *callback()* se continua por la sentencia donde el proceso se interrumpió.

11.4 Alarmas y pausas

Se puede generar una alarma con la función ***alarm()***[4]**.** Esta llamada se invoca por un proceso para decirle al núcleo que le envíe la señal *SIGALRM* al cabo de un cierto tiempo.

Una vez invocada esta función, pasando como parámetro un tiempo en segundos, se inicia la cuenta atrás de esa duración y el proceso continua su ejecución.

Cuando se recibe la señal **alarm()**, por defecto el proceso termina.

```
#include <unistd.h>
unsigned int alarm(unsigned seconds)
```

- El parámetro ***seconds*** es el número de segundos que transcurrirán antes de que se ejecute la señal por parte del núcleo.

  Un proceso sólo puede tener una petición de alarma pendiente. Las peticiones sucesivas de alarmas no se encolan, **cada nueva petición anula la anterior**.

  Si se invoca a *alarm* con el parámetro cero, si hubiera alarmas pendientes se cancelarían, *alarm(0)*.

Por otro lado, puede que a veces sea interesante parar la ejecución de un proceso hasta que se produzca una señal. Para ello se puede usar la llamada ***pause()***[5].

```
#include <unistd.h>
  int pause (void)
```

- Esta llamada hace que el proceso invocador se bloquee (quede en espera) hasta la llegada de una señal cualquiera.
- Cuando esto ocurre, y después de ejecutarse la rutina de tratamiento de la señal, *pause()* devuelve el valor -1 y la ejecución se continúa con la sentencia que sigue a esta llamada.

4 http://pubs.opengroup.org/onlinepubs/009695399/functions/alarm.html
5 http://pubs.opengroup.org/onlinepubs/9699919799/

11.5 Ejemplos de señales en Posix

El siguiente ejemplo en C muestra como se pueden capturar señales y modificar su comportamiento predeterminado mediante los ***callbacks***.

Concretamente este ejemplo muestra la captura de la señal *alarm()* y la impresión por consola de su identificador definido en *signal.h*:

```
#include<signal.h>
#include<stdio.h>
#include<unistd.h>
#include <stdlib.h>
#include <errno.h>

void sig_alrm(int signo)
{
        printf("In the sig_alrm function!!!\n");
        printf("Signal value:%d\n",signo);
        return;   /* nothing to do, just return to wake up the pause */
}

unsigned int f1(unsigned int nsecs)
{
        alarm(nsecs);      /* start the timer */
        pause();  /* pause - suspend the thread until a signal is received */
}

int main ()
{
   /* Si la solicitud de tratamiento de senial se puede llevar a cabo, la funcion signal() devolverá
   el nombre de la función (* void) que la tratará, en caso contrario se devuelve el valor de la macro
   SIG_ERR y se pone errno a un valor positivo*/

   if (signal(SIGALRM, sig_alrm) == SIG_ERR)
   {
       perror("Signal error");
       printf("errno value= %d\n", errno);
       exit(EXIT_FAILURE);
   }

   printf ("Una alarma en 3 segundos....\n");
   f1(3);
   printf ("Una alarma en 2 segundos...\n");
   f1(2);
   printf ("Fin del programa\n");
}
```

Seguidamente se muestra **otro ejemplo de capturas de señales algo más completo que le anterior en cuanto a la cantidad de señales capturadas**. Puede probarlo en su ordenador usando el comando ***kill*** desde un terminal para mandar señales. Para ello **deje ejecutar en un terminal el proceso** y abra **otro terminal para el envío de señales con el comando *kill***.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>

/* Este será nuestro manejador (personalizado) de señales SIGINT.
   SIGINT se genera cuando el usuario pulsa Ctrl-C.
   El comportamiento por defecto de un programa al pulsar Ctrl-C es salir.
   Con nuestro nuevo manejador, no saldrá.
*/
void mi_manejador_sigint(int signal)
{
   printf("Capturé la señal SIGINT y no salgo!\n");
   return;
}

/* Nuestro manejador para SIGHUP */
void mi_manejador_sighup(int signal)
{
   printf("Capturé la señal SIGHUP y no salgo!\n");
   return;
}

/* Nuestro manejador para SIGTERM */
void mi_manejador_sigterm(int signal)
{
   int i;
   printf("Capturé la señal SIGTERM y voy a salir de manera ordenada\n");
   for(i=0; i<3; i++)
   {
      printf("Hasta luego... cerrando ficheros...\n");
      sleep(1);
   }
   exit(0);
}

/* Nuestro manejador para SIGKILL. ¿Podemos asociar SIGKILL? */
void mi_manejador_sigkill(int signal)
{
   printf("Capturé la señal SIGKILL y no salgo!\n");
   return;
}
```

```
int main()
{
   /* Si la solicitud de tratamiento de senial se puede llevar a cabo, la funcion signal() devolverá
   el nombre de la función (* void) que la tratará, en caso contrario se devuelve el valor de la macro
   SIG_ERR y se pone errno a un valor positivo*/

   if (signal(SIGINT, mi_manejador_sigint) == SIG_ERR)
      printf("No puedo asociar la señal SIGINT al manejador!\n");
   if (signal(SIGHUP, mi_manejador_sighup) == SIG_ERR)
      printf("No puedo asociar la señal SIGHUP al manejador!\n");
   if (signal(SIGTERM, mi_manejador_sigterm) == SIG_ERR)
      printf("No puedo asociar la señal SIGTERM al manejador!\n");
   /* ¿Podemos asociar SIGKILL? */
   if (signal(SIGKILL, mi_manejador_sigkill) == SIG_ERR)
      printf("No puedo asociar la señal SIGKILL al manejador!\n");

    for(;;)
        pause();  //No se hace nada, solo queda a la espera de señales

   //¿Llegaremos aquí?
   exit(0);
}
```

12 Bibliografía

El contenido de este documento se ha elaborado principalmente a partir de las siguientes referencias bibliográficas, y con propósito meramente académico y no lucrativo:

- W. Stallings. *Sistemas operativos, 5ª edición.* Prentice Hall, Madrid, 2005.
- A. S. Tanenbaum. *Sistemas operativos modernos, 3a edición*. Prentice Hall, Madrid, 2009.
- A. Silberschatz, G. Gagne, P. B. Galvin. *Fundamentos de sistemas operativos, séptima edición*. McGraw-Hill, 2005.
- A. Mclver, I. M. Flynn. *Sistemas operativos, 6ª edición*. Cengage Learning, 2011.
- J. A. Alamansa, M. A. Canto Diaz, J. M. de la Cruz García, S. Dormido Bencomo, C. Mañoso Hierro. *Sistemas operativos, teoría y problemas*. Editoral Sanz y Torres, S.L, 2002.
- F. Pérez, J. Carretero, F. García. *Problemas de sistemas operativos: de la base al diseño, 2ª edición*. McGraw-Hill. 2003.
- S. Candela, C. Rubén, A. Quesada, F. J. Santana, J. M. Santos. *Fundamentos de Sistemas Operativos, teoría y ejercicios resueltos*. Paraninfo, 2005.
- J. Aranda, M. A. Canto, J. M. de la Cruz, S. Dormido, C. Mañoso. *Sistemas Operativos: Teoría y problemas*. Sanz y Torres S.L, 2002.
- J. Carretero, F. García, P. de Miguel, F. Pérez, *Sistemas Operativos: Una visión aplicada.* Mc Graw Hill, 2001