



UNIVERSIDAD DE CÓRDOBA
ESCUELA POLITÉCNICA SUPERIOR
DEPARTAMENTO DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

ASIGNATURA ***SISTEMAS OPERATIVOS***

2º DE GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 3

Comunicación y sincronización entre hilos mediante semáforos

Juan Carlos Fernández Caballero

jfcaballero@uco.es

Índice de contenido

1	Objetivo de la práctica.....	3
2	Recomendaciones.....	3
3	Conceptos teóricos.....	4
3.1	Concurrencia.....	4
3.2	Condiciones de carrera.....	4
3.3	Sección crítica y exclusión mutua.....	7
3.4	Soluciones algorítmicas clásicas a la exclusión mutua.....	8
3.5	Cerrosos, barreras o mutexes en hebras.....	8
3.5.1	Inicialización de un mutex (<i>pthread_mutex_init()</i>).....	10
3.5.2	Petición de bloqueo de un mutex (<i>pthread_mutex_lock()</i>).....	11
3.5.3	Petición de desbloqueo de un mutex (<i>pthread_mutex_unlock()</i>).....	11
3.5.4	Destrucción de un mutex (<i>pthread_mutex_destroy()</i>).....	12
3.5.5	Ejemplos de uso de mutexes.....	12
3.6	Semáforos generales.....	14
3.6.1	Inicialización de un semáforo (<i>sem_init()</i>).....	16
3.6.2	Petición de bloqueo o bajada del semáforo (<i>sem_wait()</i>).....	16
3.6.3	Petición de desbloqueo o subida del semáforo (<i>sem_post()</i>).....	17
3.6.4	Destrucción de un semáforo (<i>sem_destroy()</i>).....	17
3.6.5	Ejemplos.....	17
3.7	Resumen sobre mutexes y semáforos sin nombre y con nombre.....	19
4	Ejercicios prácticos.....	20
4.1	Usando semáforos binarios o mutexes.....	20
4.1.1	Ejercicio 1.....	20
4.1.2	Ejercicio 2.....	20
4.2	Usando semáforos generales.....	21
4.2.1	Ejercicio 3.....	21

1 Objetivo de la práctica

La presente práctica persigue instruir al alumnado con la comunicación y sincronización entre procesos ligeros, hilos o hebras (indistintamente) para el acceso en exclusión mutua a recursos compartidos.

En una primera parte se dará una breve introducción teórica sobre el problema de la concurrencia y sus posibles soluciones con algoritmos clásicos (en desuso) y **semáforos binarios o mutexes**.

La segunda parte de la práctica se dedica a teoría sobre la utilización de **semáforos generales**.

En la tercera parte, mediante programación en C, se practicarán los conceptos sobre las temáticas vistas, utilizando para ello las rutinas de interfaz de sistema que proporcionan a los programadores la biblioteca ***semaphore***, basada en el estándar POSIX.

2 Recomendaciones

El lector debe completar las nociones dadas en las siguientes secciones con consultas bibliográficas, tanto en la Web como en la biblioteca de la Universidad, ya que unos de los objetivos de las prácticas es potenciar su capacidad autodidacta y su capacidad de análisis de un problema.

Es recomendable que, aparte de los ejercicios prácticos que se proponen, pruebe y modifique otros que se encuentren en la Web (se dispone de una gran cantidad de problemas resueltos en C sobre esta temática), ya que al final de curso deberá acometer un examen práctico en ordenador como parte de la evaluación de la asignatura.

Al igual que se le instruyó en las asignaturas de Metodología de la Programación, es recomendable que siga unas normas y estilo de programación claro y consistente. No olvide tampoco comentar los aspectos más importantes de sus programas, así como añadir información de cabecera a sus funciones (nombre, parámetros de entrada, parámetros de salida, objetivo, etc).

No olvide que debe consultar siempre que lo necesite el estándar *POSIX* en:

<http://pubs.opengroup.org/onlinepubs/9699919799/>

y la biblioteca GNU C (*glibc*) en:

<http://www.gnu.org/software/libc/libc.html>

Los **conceptos teóricos** que se exponen a continuación **se aplican tanto a procesos como a hilos**, pero la parte práctica de este guión va dirigida a la resolución de problemas de concurrencia mediante hilos. Para obtener soluciones usando procesos el lector debería utilizar las rutinas de interfaz para **memoria compartida entre procesos**. *POSIX* proporciona también un estándar para ello (*POSIX Shared Memory*¹) que implementan los sistemas basados en el.

1 http://en.wikipedia.org/wiki/Shared_memory

3 Conceptos teóricos

A continuación se exponen los conceptos teóricos sobre los cuales se trabajará para resolver los ejercicios prácticos.

3.1 Concurrency

Un sistema operativo multitarea actual también permite que coexistan varios procesos activos a la vez a nivel de proceso y a nivel de hilo (multiprocesamiento), es decir, varios procesos que se están ejecutando de forma **paralela y/o concurrente** (existencia simultánea de varios procesos en ejecución, en desorden, sin afectar el resultado final). En el caso de que exista un solo procesador con un solo núcleo (los procesadores actuales poseen varios núcleos), el sistema operativo se encarga de ir repartiendo el tiempo del procesador entre los distintos procesos, intercalando la ejecución de los mismos, dando así una apariencia de ejecución simultánea (multiprogramación). Cuando existen varios procesadores o un procesador con varios núcleos, los procesos no sólo pueden intercalar su ejecución sino también superponerla (paralelizarla). En este caso sí existe una verdadera ejecución simultánea de procesos, pudiendo ejecutar cada procesador o cada núcleo un proceso o hilo diferente.

En ambos casos (uniprocador y multiprocador), el sistema operativo mediante un algoritmo de planificación, otorga a cada proceso un tiempo de procesador para no dejar a otros procesos del sistema sin su uso. La finalización del **tiempo de reloj** (“rodaja de tiempo”) otorgado a los procesos del sistema puede dar lugar a **cambios de contexto** (**pasar a ejecutar otro proceso diferente al actual**) que a su vez pueden dar lugar a **problemas de concurrencia**. Estos problemas de concurrencia hacen que se produzcan inconsistencias en los recursos compartidos por 2 o más procesos (o hilos indistintamente), y hay que emplear las metodologías y técnicas adecuadas para que eso no ocurra. Veamos más sobre ello en las siguientes secciones.

3.2 Condiciones de carrera

Cuando decidimos trabajar con programas concurrentes, uno de los mayores problemas con los que nos podremos encontrar y que es inherente a la concurrencia, es el acceso a variables y/o estructuras compartidas o globales.

Cuando 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 dice que se produce una **condición de carrera**. Veamos algunos ejemplos.

Ejemplo 1:

Este código, que tiene la **variable $i=10$ como variable global o compartida**, aparentemente es inofensivo, pero nos puede traer muchos problemas si se dan ciertas condiciones:

<pre>//Hilo 1 <i>ejecuta primero</i>. Comparte variable global $i=10$ void *funcion_hilo1 (void *arg) { int *resultado = malloc (sizeof(int)); ... if (i!=0) { ... // Cambio de contexto (1) *resultado = i * (int)*arg; //Retoma la CPU (3) ... } pthread_exit((void *) resultado); }</pre>	<pre>//Hilo 2 <i>ejecuta segundo</i>. Comparte variable global $i=10$ void *funcion_hilo2 (void *arg) //Suponemos *arg = 0 { int *otro_resultado = malloc (sizeof(int)); ... if ((int)*arg==0) { i = (int)*arg; ... // Cambio de contexto (2) *otro_resultado = 99; //Retoma la CPU (4) } pthread_exit((void *) otro_resultado); }</pre>
--	--

Supongamos que el hilo 1 se empieza a ejecutar antes que el hilo 2, y que casualmente se produce un cambio de contexto por finalización del tiempo de reloj (“rodaja de tiempo”) justo después de la línea que dice “*if (i != 0)*”. La entrada en ese *if* se producirá si se cumple la condición, que vamos a suponer verdadera, es decir, suponemos que i vale por ejemplo 10. Justo después de ese momento el sistema pone a ejecutar al hilo 2, que se ejecuta el tiempo suficiente como para ejecutar la línea “ $i = (int)*arg$ ”, donde podríamos suponer que arg vale por ejemplo $*arg = 0$.

Al poco rato, hilo 2 deja de ejecutarse y el planificador vuelve a optar por ejecutar el hilo 1, entonces, ¿qué valor tiene ahora i ? El hilo 1 está “suponiendo” que tiene el valor de i que comprobó al entrar en el *if*, pero i ha tomado el valor que le asignó hilo 2, con lo que el resultado que devolverá el hilo 1 después de sus cálculos posiblemente será totalmente inconsistente e inesperado.

Todo esto puede que no sucediese si el sistema tuviera muy pocos procesos en ese momento, con lo cual el sistema podría optar por que cada proceso se ejecute por más tiempo, si el procesador fuera muy rápido o y si el código del hilo 1 fuera lo suficientemente corto como para no sufrir ningún cambio de contexto en medio de su ejecución. **Pero NUNCA deberemos hacer suposiciones de éste tipo, porque no sabremos hasta dónde y cuándo se van a ejecutar nuestros procesos.**

El problema que tienen estos *bugs* es que son difíciles de detectar si a la hora de implementar nuestro código no nos fijamos en qué parte puede haber condiciones de carrera. Puede que a veces vaya todo a la perfección y que en otras ocasiones salga todo mal debido a las condiciones de carrera no controladas.

Ejemplo 2:

Aquí tiene otro ejemplo sobre problemas de concurrencia de procesos. Considere el siguiente procedimiento. 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):

```
//Función compartida entre dos procedimientos P1 y P2. Una única instancia de la función
void eco ()
{
    cent = getchar(); //Proceso P1 sale de la CPU tras leer una "x"
    csal = cent; //Proceso P1 retomará esta línea después de que P2 se ejecute entero
    putchar(csal);
}
//El proceso P2 ejecuta entero después de que P1 salga de la CPU y recoge una "y"
//¿Qué se imprime por pantalla? Dos veces la "y"
/*¿Por que no ponemos directamente csal = getchar();?
Para simular que la operación podría conllevar varias instrucciones a nivel ensamblador y que un proceso puede
salir de la CPU al terminar cualquiera de esas instrucciones a más bajo nivel */
```

Este procedimiento muestra los elementos esenciales de un **programa que proporcionará un procedimiento de eco de un carácter**; la entrada se obtiene del teclado, una tecla cada vez. Cada carácter introducido se almacena en la variable *cent*. Luego se transfiere a la variable *csal* y se envía a la pantalla. Cualquier programa puede llamar repetidamente a este procedimiento para aceptar la entrada del usuario y mostrarla por su pantalla.

El sistema puede saltar de una aplicación a otra, y cada aplicación utiliza el mismo teclado para la entrada y la misma pantalla para la salida. Dado que cada aplicación necesita usar el procedimiento *eco*, tendría sentido que éste fuera un procedimiento compartido que esté cargado en una porción de memoria global para todas las aplicaciones. De este modo, sólo se utiliza una única copia del procedimiento *eco*, economizando espacio.

La esencia de este problema es la variable compartida global, *cent*. Múltiples procesos tienen acceso a esta variable. Si un proceso actualiza la variable global y justo entonces es interrumpido, otro proceso puede alterar la variable antes de que el primer proceso pueda utilizar su valor.

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

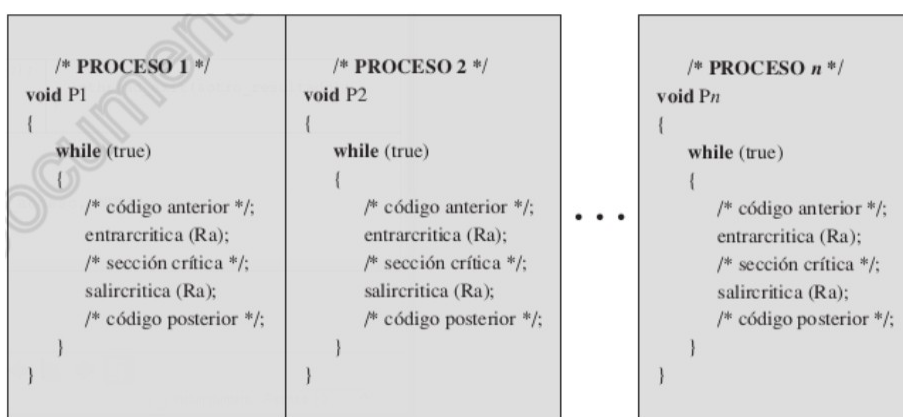
1. El proceso P1 invoca el procedimiento *eco* y es interrumpido inmediatamente después de que *getchar()* devuelva su valor y sea almacenado en *cent*. En este punto, el carácter introducido más recientemente, "x", está almacenado en *cent*.
2. El proceso P2 se activa e invoca al procedimiento *eco*, que ejecuta hasta concluir, habiendo leído y mostrado en pantalla un único carácter, "y".
3. Se retoma el proceso P1. En este instante, el valor "x" ha sido sobrescrito en *cent* y por tanto se ha perdido. En su lugar, *cent* contiene "y", que es transferido a *csal* y mostrado.
4. Así, el primer carácter se pierde y el segundo se muestra dos veces.

3.3 Sección crítica 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 recursos compartidos, es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo o mediante multiplexaciones (intercambios) no controladas. Esa parte del programa en la que se accede a un recurso compartido se le conoce como **región crítica** o **sección crítica**. Por tanto, un proceso está en su sección crítica cuando accede a datos compartidos modificables.

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 un recurso compartido, los demás procesos se excluirán de hacer lo mismo (se evita la carrera).

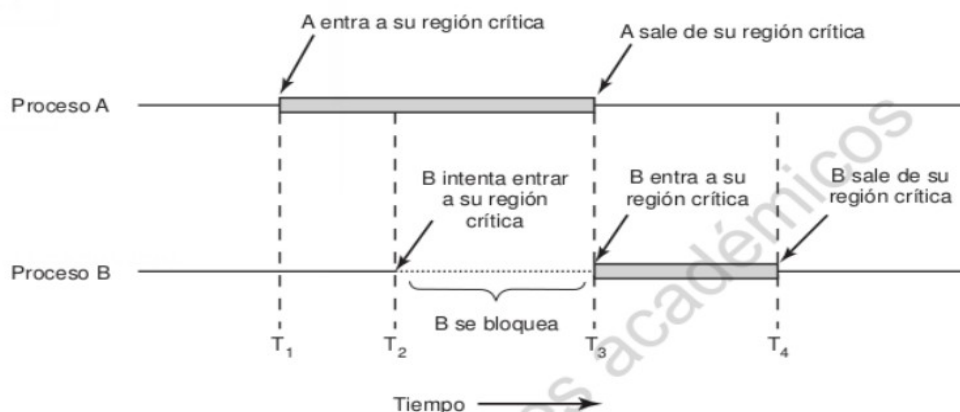
La siguiente figura ilustra el mecanismo de exclusión mutua en términos abstractos:



Hay n procesos para ser ejecutados concurrentemente, cada proceso incluye:

1. Una sección crítica (SC) que opera sobre algún **recurso R_a** compartido,
2. Código adicional que **precede** y **sucede** a la SC y que no involucra acceso a R_a .

Se desea que sólo un proceso esté en su SC al mismo tiempo, por lo que se proporcionan dos funciones o mecanismos abstractos: **entrarcritica()** y **salircritica()**, que hacen que a cualquier proceso que intente entrar en su SC mientras otro proceso está en su SC por el mismo recurso, **se le hace esperar**. Así, si dos procesos A y B intentasen entrar en una sección crítica, la solución proporcionada debería permitir la exclusión mutua reflejada en la siguiente figura:



3.4 Soluciones algorítmicas clásicas a la exclusión mutua

Para controlar el problema del acceso a una sección crítica y que se produzca sin incidencia la exclusión mutua existen varias técnicas, siendo una de ellas los algoritmos de sincronización o comunicación clásicos como los algoritmos de Dekker², Peterson³ y Lamport⁴. Si está interesado en estas soluciones clásicas consulte la bibliografía, en esta practica nos centraremos en las soluciones actuales, ya que en algunos casos las clásicas conllevan a soluciones poco eficientes.

Hay también técnicas hardware como la inhabilitación de interrupciones e instrucciones máquina especiales, pero en esta práctica solo trataremos soluciones software⁵.

3.5 Cerrojos, barreras o mutexes en hebras

La biblioteca de hilos *pthread* proporciona una serie de funciones o llamadas al sistema basadas en el estándar *POSIX* para la resolución de problemas de exclusión mutua, lo cual hace su uso relativamente más sencillo y eficiente con respecto a los algoritmos clásicos.

Estos mecanismos se llaman indistintamente **mutexes**, **barreras**, **cerrojos** (*locks*) o incluso **semáforos binarios**. Son muy similares a lo que se conoce como **semáforos** (se estudiarán posteriormente), aunque **tienen algunas diferencias**. Por ejemplo, **un mutex solo puede liberarse por el hilo que lo adquirió**, mientras que un semáforo puede liberarse desde cualquier otro hilo (o proceso), por lo que los semáforos son más adecuados para algunos problemas de sincronización como el problema del productor-consumidor⁶.

Un **mutex** (*Mutual Exclusion*) se asemeja a un semáforo vial (en este caso símil con señal de circulación) porque puede tener dos estados, **abierto** o **cerrado**, los cuales servirán para proteger el acceso a una sección crítica. Cuando un semáforo está abierto (verde), al primer *thread* que pide entrar en una sección crítica se le permite el acceso y no se deja pasar a nadie más por el semáforo. Mientras que si el semáforo está cerrado (rojo) (algún *thread* ya lo usó y accedió a sección crítica poniéndolo en rojo), el *thread* que lo pide parará su ejecución hasta que se vuelva a abrir el semáforo (puesta en verde). Dicho esto, solo podrá haber un *thread* poseyendo el bloqueo de un semáforo binario o barrera, mientras que **puede haber más de un thread esperando para entrar en una sección crítica**.

Con estos conceptos se puede implementar el acceso a una sección crítica:

1. Se pide el bloqueo del semáforo antes de entrar.
2. El semáforo es otorgado al primero que llega.
3. Los demás hilos que intentan acceder se registran en una **cola FIFO** y se quedan bloqueados en espera de un evento, que será una señal de liberación. Por tanto, estos hilos **deben esperar a que el hilo que entró primero libere el bloqueo y se envíe una señal sobre ello**.
4. Una vez el que el hilo que entró en la sección crítica sale de ella debe notificarlo a la biblioteca *pthread* para que mire si había algún otro *thread* esperando para entrar en la cola y dejarlo entrar.

2 http://es.wikipedia.org/wiki/Algoritmo_de_Dekker

3 http://es.wikipedia.org/wiki/Algoritmo_de_Peterson

4 http://es.wikipedia.org/wiki/Algoritmo_de_la_panader%C3%ADa_de_Lamport

5 <http://sistemaoperativo.wikispaces.com/Exclusion+Mutua>

6 http://es.wikipedia.org/wiki/Problema_Productor-Consumidor

A continuación se muestra una definición de un semáforo binario o *mutex*:

```
struct binary_semaphore {
    enum {cero, uno} valor;
    queueType cola ;
};

void semWaitB (binary_semaphore s) /* Operación atómica entrarcritica() para un mutex "s" */
{
    if (s.valor == 1)
        s.valor = 0;
    else
    {
        Poner proceso en s.colas;
        Bloquear al proceso;
    }
}

void semSignalB (binary_semaphore s) /* Operación atómica salircritica() para un mutex "s" */
{
    if (esta_vacia(s.colas))
        s.valor = 1;
    else
    {
        Extraer un proceso P de s.colas;
        Poner al proceso P en la lista de Listos;
    }
}
```

En el caso de *semSignalB()*, si la cola no está vacía se extrae una hebra registrada en ella, pero no se abre el semáforo, de esta manera, esa hebra podrá continuar por la sección crítica sin que ninguna otra hebra del sistema "se cuele" por delante.

A continuación se expondrán brevemente las funciones más utilizadas para llevar a cabo exclusión mutua con hebras y que ofrece la biblioteca *pthread*.

3.5.1 Inicialización de un mutex (*pthread_mutex_init()*)

Esta función inicializa un *mutex*⁷. Hay que llamarla antes de usar cualquiera de las funciones que trabajan con *mutex*.

```
#include <pthread.h>

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

- **mutex:** Es un puntero a un parámetro del tipo *pthread_mutex_t*, que es el tipo de datos que usa la biblioteca *pthread* para controlar los mutex.
- **attr:** Es un puntero a una estructura del tipo *pthread_mutexattr_t*⁸, y sirve para definir qué tipo de *mutex* queremos:
 - Normal (PTHREAD_MUTEX_NORMAL).
 - Rekursivo (PTHREAD_MUTEX_RECURSIVE).
 - Errorcheck (PTHREAD_MUTEX_ERRORCHECK).

Si el valor de *attr* es **NULL**, la biblioteca le asignará un valor por defecto, concretamente **PTHREAD_MUTEX_NORMAL**.

Si desea ampliar conocimientos busque en la Web información sobre los tipos de *mutex* anteriormente citados, en esta práctica se usarán los *mutex* por defecto.

pthread_mutex_init() devuelve 0 si se pudo crear el *mutex* o distinto de cero si hubo algún error. Algunos tipos de error que se definen en *OpenGroup* son [EAGAIN], [ENOMEM], [EPERM], [EINVAL], consúltelos.

También es posible inicializar un *mutex* por defecto sin usar la función *pthread_mutex_init()*, basta con declararlo de esta manera:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Un *mutex* debe ser declarado en la zona de memoria que comparten los hilos que forman parte de un mismo proceso, en este caso podría ser en la **zona de variables globales o en el montículo**.

⁷ http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_init.html

⁸ https://www.sourceware.org/pthreads-win32/manual/pthread_mutexattr_init.html

3.5.2 Petición de bloqueo de un mutex (*pthread_mutex_lock()*)

Esta función⁹ pide el bloqueo para entrar en una sección crítica. Si queremos implementar una sección crítica, todos los *threads* tendrán que pedir el bloqueo sobre el mismo *mutex*.

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- **mutex:** Es un puntero al *mutex* sobre el cual queremos pedir el bloqueo o sobre el que nos bloquearemos en caso de que ya haya alguna hebra dentro de la sección crítica.

Si la sección crítica ya se encuentra ocupada, es decir, alguna otra hebra bloqueó el *mutex* previamente, entonces el sistema operativo bloquea a la hebra actual que hace la invocación de *pthread_mutex_lock()* hasta que el *mutex* se libere.

Como resultado, **devuelve 0 si no hubo error, o diferente de 0 si lo hubo**. Consulte los tipos de error que se definen en *OpenGroup*.

3.5.3 Petición de desbloqueo de un mutex (*pthread_mutex_unlock()*)

Esta es la función¹⁰ contraria a la anterior. Libera el bloqueo que tuviéramos sobre un *mutex*.

```
#include <pthread.h>

int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

- **mutex:** Es el semáforo donde tenemos el bloqueo y queremos liberarlo. Al liberarlo, la sección crítica queda disponible para otra hebra que esté registrada en la cola (FIFO), o bien para otra hebra que lo solicite si cola está vacía, ya que el semáforo quedaría abierto.

Retorna 0 como resultado si no hubo error o diferente de 0 si lo hubo. Consulte los tipos de error que se definen en *OpenGroup*.

Cuando estamos utilizando *mutex*, es responsabilidad del programador el asegurarse de que cada hebra que necesite utilizar un *mutex* lo haga, es decir, si por ejemplo 4 hebras están actualizando los mismos datos pero solo una de ellas usa un *mutex*, los datos podrían corromperse.

Otro punto importante es que si una hebra utiliza un *pthread_mutex_lock()* para proteger su sección crítica, esa misma hebra que lo adquirió es la que debe desbloquear ese *mutex* mediante la invocación a *pthread_mutex_unlock()* (excepto cuando se usen variables de condición ¹¹, las cuales permiten bloquear a un hilo hasta que ocurra algún suceso, pero no se abordarán en esta práctica).

Si un hilo intenta desbloquear un *mutex* que no está bloqueado, *OpenGroup* no define el comportamiento de nuestro programa.

⁹ http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_lock.html

¹⁰ https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_unlock.html

¹¹ <https://computing.lln.gov/tutorials/pthreads/#ConditionVariables>

3.5.4 Destrucción de un mutex (`pthread_mutex_destroy()`)

Esta función¹² le dice a la biblioteca que el *mutex* que le estamos indicando no lo vamos a usar más (ninguna hebra lo va a bloquear más en el futuro), y que puede liberar toda la memoria ocupada en sus estructuras internas por ese *mutex*.

Esa destrucción se debe producir inmediatamente después de que se libera el *mutex* o barrera por alguna hebra. **Si se intenta destruir un *mutex* que está bloqueado por una hebra, el comportamiento de nuestro programa no está definido en *OpenGroup*.** Si se quiere reutilizar un *mutex* destruido debe volver a ser reinicializado con `pthread_mutex_init()`, de lo contrario los resultados que se pueden obtener después de que ha sido destruido no están definidos.

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

- **mutex:** El mutex que queremos destruir.

La función, como siempre, **devuelve 0 si no hubo error, o distinto de 0 si lo hubo**. Consulte los tipos de error que se definen en *OpenGroup*.

3.5.5 Ejemplos de uso de mutexes

Veamos como se reescribe el código de una de las secciones anteriores usando *mutexes*. En color azul se han añadido líneas que antes no estaban. Como se puede observar, lo que hay que hacer es inicializar el *mutex* (lo suponemos inicializado), pedir el bloqueo antes de la sección crítica y liberarlo después de salir de la misma. Mientras más pequeñas hagamos las secciones críticas, más concurrentes serán nuestros programas, porque tendrán que esperar menos tiempo en el caso de que haya bloqueos.

<pre>//Hilo 1. Comparte variable global i=10 y // un mutex pthread_mutex_t mutex_acceso void *funcion_hilo1 (void *arg) { int *resultado = malloc (sizeof(int)); ... pthread_mutex_lock(&mutex_acceso); if (i!=0) { ... *resultado = i * (int)*arg; ... } pthread_mutex_unlock(&mutex_acceso); pthread_exit((void *) resultado); }</pre>	<pre>//Hilo 2. Comparte variable global i=10 y // un mutex pthread_mutex_t mutex_acceso void *funcion_hilo2 (void *arg) { int *otro_resultado = malloc (sizeof(int)); ... if ((int)*arg==0) { pthread_mutex_lock(&mutex_acceso); i = (int)*arg; pthread_mutex_unlock(&mutex_acceso); ... *otro_resultado = 99; } pthread_exit((void *) otro_resultado); }</pre>
--	---

¹² http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_destroy.html

En la **demo1.c** se muestra otro ejemplo del uso de *mutex* para proteger una variable global que se incrementa de forma concurrente por dos hebras. Puede ejecutarlo por ejemplo con “./a.out 5”, donde “5” indica el número de “*loops*” a realizar en el bucle de la función que se le asignan a las hebras implicadas, dos en este caso. Puede probar un número de incrementos mayor si los hilos no se intercambian en el incremento al ser demasiado cortos sus tiempos de procesamiento.

En la **demo2.c** se muestra una serie de hebras que irán aumentando el valor de una variable llamada *avail*, la cual será consumida por la hebra principal o *main()*, que va decrementando su valor hasta que llega a cero. El código funciona bien, pero en el segundo bucle *for()* del *main()* se está consumiendo recursos de CPU sin hacer nada productivo, en el caso de que la variable *avail* sea 0 y el *main()* tenga asignado actualmente el procesador, es decir, la hebra *main()* no puede consumir valores *avail* mientras que no sean mayor que cero y sus comprobaciones no sirven para nada productivo.

Ejecute el programa varias veces para ver qué ocurre según las decisiones del planificador. Puede hacerlo de la siguiente manera:

`./a.out 2 3 2` → Se crean 3 hebras, la primera incrementa 2 veces *avail*, la segunda incrementa 3 veces y la tercera 2 veces.

`./a.out 3 2 4 2` → Se crean 4 hebras, la primera incrementa 3 veces *avail*, la segunda 2 veces, y así consecutivamente.

3.6 Semáforos generales

Un **semáforo general** es un objeto o estructura con un valor entero al que se le puede asignar un **valor inicial no negativo**, con una cola de procesos, y al que sólo se puede acceder utilizando dos operaciones atómicas: **wait()** y **signal()**.

Su filosofía es parecida a los *mutexes* que estudió en el apartado anterior, ya que se basan en que dos o más hilos (o 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 de otro. Por otro lado, los semáforos generales pueden desbloquearse por cualquier otro hilo (o proceso) que no lo haya bloqueado previamente y controlar determinadas situaciones o condiciones que los semáforos binarios o *mutexes* no pueden controlar, como se refleja en el problema del productor-consumidor ¹³. Esas condiciones se podrían tratar con el uso de *mutexes* junto con variables de condición ¹⁴, las cuales permiten bloquear a un hilo hasta que ocurra algún suceso, pero no se abordarán en esta práctica.

De manera abstracta (veremos más adelante la implementación POSIX), las operaciones *wait()* y *signal()* de un semáforo general o con contador (se entiende de aquí en adelante) son las siguientes:

```
struct semaphore {
    int contador;
    queueType cola ;
};

wait (semaphore s) /* Operación atómica wait() para un semáforo "s" */
{
    s. contador --;
    if (s. contador < 0)
    {
        Poner proceso en s.colas;
        Bloquear al proceso;
    }
}

signal (semaphore s) /* Operación atómica signal() para un semáforo "s" */
{
    s. contador ++;
    if ( s. contador <= 0 )
    {
        Extraer un proceso de s.colas bloqueado en la operación wait();
        Poner el proceso listo para ejecutar;
    }
}
```

La **operación wait()** decrementa el valor del semáforo "s". Si el valor pasa a ser negativo, entonces el proceso que está ejecutando *wait()* se bloquea. Cuando al decrementar "s" el valor es cero o positivo, el proceso entra en la sección crítica. **El semáforo se suele inicializar a 1** (pero se puede poner cualquier valor positivo), para que el primer proceso que llegue, A, lo decremente a cero y entre en su sección crítica. Si algún otro proceso llega después, B, y A todavía está en la sección

¹³ https://es.wikipedia.org/wiki/Problema_productor-consumidor

¹⁴ <https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables>

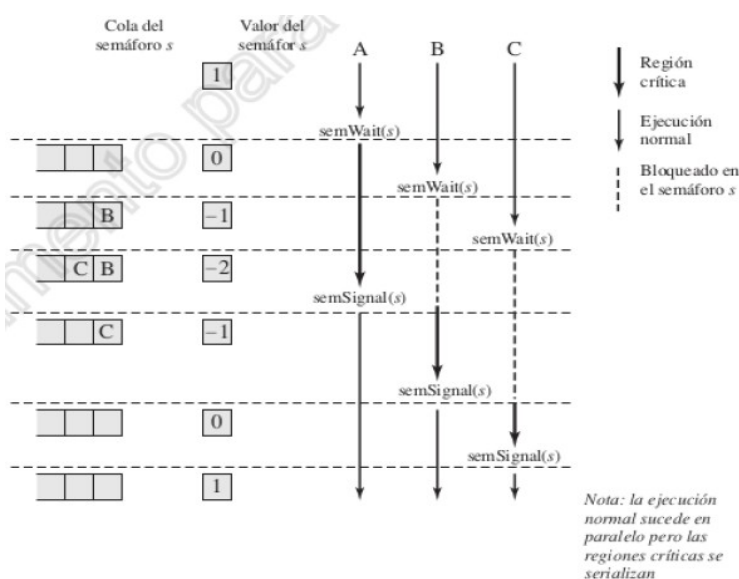
crítica, B se bloquea porque al decrementar el contador pasa a ser negativo.

Si el valor inicial del semáforo fuera, por ejemplo, 2, entonces dos procesos podrían ejecutar la llamada *wait()* sin bloquearse y por tanto se permitiría que ambos ejecutaran de forma simultánea dentro de la sección crítica. Algo peligroso si se realizan operaciones de modificación en la sección crítica y no se controlan.

La **operación *signal()*** incrementa el valor del semáforo. Si el valor es menor o igual que cero, entonces se desbloquea uno de los procesos bloqueados en la operación *wait()* (**normalmente orden FIFO de desbloqueo**).

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

Ejemplo de 3 procesos (A, B, C) que acceden a recurso compartido protegido por el semáforo “s”.



El estándar *POSIX* especifica una interfaz de semáforos generales para procesos y hebras que no es parte de la biblioteca *pthread* (los binarios solo sirven para hebras). *Pthread* no proporciona funciones para el manejo de semáforos generales, se proporcionan a partir de la biblioteca *semaphore*, definida en *semaphore.h*. Esta biblioteca se basa en el estándar *POSIX* y sirve para sincronización de hebras y también de procesos. **En el caso de las hebras los semáforos se deben declarar en la zona de variables globales o en el montículo, en el caso de procesos hay que hacer uso de la técnica de IPC de memoria compartida.**

Existe otra implementación de semáforos, conocida como semáforos en “*Unix System V*”, que es la implementación tradicional de Unix, pero son más complejos de utilizar y no aportan más funcionalidad, además de quedar ya obsoletos en los *kernels* actuales. Por último, hay que diferenciar entre lo que son **semáforos sin nombre** y **semáforos con nombre** ¹⁵ (se usan como si fueran ficheros). En las prácticas de la asignatura se trabajará con semáforos sin nombre.

15 https://man7.org/linux/man-pages/man7/sem_overview.7.html

3.6.1 Inicialización de un semáforo (*sem_init()*)

En POSIX un semáforo se identifica mediante una variable del tipo *sem_t*. Los semáforos deben inicializarse antes de usarlos. Para ello utilizaremos la siguiente función¹⁶:

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, int value);
```

- **sem**: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.
- **pshared**: Entero que determina si el semáforo sólo puede ser utilizado por hilos del mismo proceso que inicia el semáforo, en cuyo caso pondremos como parámetro el **valor 0**, o se puede utilizar para sincronizar procesos que heredan el semáforo a través de *fork()*, en cuyo caso pondremos un **valor de 1**. Se usará un 1 cuando se utilice memoria compartida entre procesos.

Por tanto, un semáforo debe ser declarado en la zona de memoria que comparten los hilos que forman parte de un mismo proceso, en ese caso podría ser en la **zona de variables globales o en el montículo**; o se debe hacer uso de **memoria compartida** en el caso de comunicar procesos herederos a través de *fork()*. Los semáforos con nombre si permiten comunicar procesos independientes que no son hijos de procesos que han llamado a *fork()*.

- **value**: Entero que representa el valor que se asigna inicialmente al semáforo.

Esta función **devuelve 0 en caso de que que pueda inicializar el semáforo o -1 en caso contrario estableciendo el valor del error en *errno***. Consulte *OpenGroup*.

3.6.2 Petición de bloqueo o bajada del semáforo (*sem_wait()*)

Para entrar en la sección crítica por parte de un proceso antes se debe consultar el semáforo con *sem_wait()*¹⁷:

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

- **sem**: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.

Como se comento al principio de la práctica, al realizar esta llamada, el contador del semáforo se decrementa. Si el valor pasa a ser negativo, el proceso que está ejecutando *sem_wait()* se bloquea.

Esta función **devuelve 0 en caso de que no haya problemas con la llamada o -1 en caso contrario estableciendo el valor del error en *errno***.

¹⁶ http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_init.html

¹⁷ http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_wait.html

3.6.3 Petición de desbloqueo o subida del semáforo (*sem_post()*)

Cuando un proceso va a salir de la sección crítica es necesario que envíe una señal indicando que ésta queda libre, y eso lo hace con la función *sem_post()*¹⁸:

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

- **sem:** Puntero a parámetro de tipo *sem_t* que identifica el semáforo.

La operación *sem_post()* incrementa el valor del semáforo. Si al hacer el incremento el valor es menor o igual que cero, entonces se **desbloquea uno de los procesos bloqueados** en la operación *sem_wait()*, **normalmente se suele emplear para ello un esquema FIFO (First In First Out)**. Si al hacer el incremento el valor es > 0 , entonces es que no hay hilos bloqueados y pueden entrar en sección crítica tanto hilos como valor tenga el número positivo. Es responsabilidad del programador el controlar posibles *sem_post()* que incrementen un semáforo a un valor positivo > 1 .

Esta función **devuelve 0 en caso de que no haya problemas con la llamada o -1 en caso contrario estableciendo el valor del error en *errno*. Consulte *OpenGroup*.**

3.6.4 Destrucción de un semáforo (*sem_destroy()*)

Para destruir un semáforo previamente creado con *sem_init()* se utiliza la siguiente función¹⁹:

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

- **sem:** Puntero al semáforo a destruir para liberar recursos.

Esta función **devuelve 0 en caso de que que pueda destruir el semáforo o -1 en caso contrario estableciendo el valor del error en *errno*. Consulte *OpenGroup* para el tratamiento de errores.**

3.6.5 Ejemplos

La **demo3.c** es similar a la **demo1.c**, pero en este caso usando semáforos generales. Concretamente se crean dos hilos, y cada hilo hace exactamente lo mismo, incrementar una variable global un número de veces. El resultado final debe ser el doble del número de veces que se incrementa la variable, ya que cada hilo la incrementa el mismo número “NITER” de veces. Si no hubiera semáforo general, el resultado podría ser inconsistente. Pruebe a ejecutarlo con semáforos y quitando los semáforos.

La **demo4.c** es un programa de sincronización entre hilos que suma los números impares entre 1 y 20, es decir, los números $1+3+5+7+9+11+13+15+17+19 = 100$. El primer hilo comprueba que el número es impar, si lo es deja que el segundo hilo lo sume, sino comprueba el siguiente número. Cópielo y ejecútelo para comprobar su salida. Haga una traza en papel de su funcionamiento

¹⁸ http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_post.html

¹⁹ http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_destroy.html

teniendo en cuenta varios casos en los que el procesador da rodaja de tiempo al hilo P1 o al hilo P2.

La **demo5.c** es la implementación del problema lectores-escritores donde hay 2 lectores y 2 escritores (prioridad a los lectores). Hay un *buffer* con un solo dato que se va incrementando. Compílelo, ejecútelo y observe su salida. Haga varias trazas del comportamiento del programa para poder entender los semáforos.

Aunque es un problema muy conocido en la Web y en la bibliografía, a continuación se expone el pseudocódigo del problema de los lectores-escritores:

```
/* 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, no tiene porque ser el que hizo el primer semWait(sc).

void lector()
{
    while (true)
    {
        semWait(x);      //Actualiza cuentalect. Se quiere leer.
        cuentalect++;
        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);
        cuentalect--;     //Actualiza cuentalect. Ya se ha leído.
        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)
    {
        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);
}
```

3.7 Resumen sobre mutexes y semáforos sin nombre y con nombre

A continuación se expone un breve resumen sobre el uso de *mutexes* y semáforos con hebras y procesos:

- **Comunicación entre hebras:**
 - **mutex o semáforos binarios:**
 - Las hebras deben pertenecer al mismo proceso.
 - La misma hebra que bloquea debe desbloquear, a no ser que se utilicen de manera adicional variables de condición. Las variables de condición permiten bloquear a un hilo hasta que ocurra algún suceso o condición.
 - Se usan principalmente para proteger una sección crítica.
 - DEBEN alojarse en zona de MEMORIA GLOBAL O MONTÍCULO.
 - **semáforos generales (semáforos sin nombre):**
 - Las hebras deben pertenecer al mismo proceso.
 - Pueden desbloquearse por cualquier hebra.
 - Además de poder usarse para proteger una sección crítica permiten controlar determinadas condiciones por su contador configurable.
 - DEBEN alojarse en zona de MEMORIA GLOBAL O MONTÍCULO.
 - **semáforos generales (semáforos con nombre):**
 - Las hebras pueden pertenecer a procesos diferentes, no tienen porque estar en el mismo proceso, aunque también pueden usarse entre hebras del mismo proceso.
 - Pueden desbloquearse por cualquier hebra.
 - Además de poder usarse para proteger una sección crítica permiten controlar determinadas condiciones por su contador configurable.
 - NO necesitan alojarse en zona de MEMORIA GLOBAL O MONTÍCULO NI en una zona de memoria común mediante la técnica de MEMORIA COMPARTIDA.
- **Comunicación entre procesos:**
 - **semáforos generales (semáforos sin nombre):**
 - Los procesos deben haber heredado el semáforo a partir de una invocación `fork()` del proceso padre, es decir, se utiliza entre procesos hijos de un mismo proceso padre.
 - Pueden desbloquearse por cualquier proceso hijo del mismo proceso padre.
 - Además de poder usarse para proteger una sección crítica permiten controlar determinadas condiciones por su contador configurable.
 - DEBEN alojarse en una zona de memoria común mediante la técnica de MEMORIA COMPARTIDA.
 - **semáforos generales (semáforos con nombre):**
 - Los procesos pueden ser totalmente independientes, no tienen porque heredar el

semáforo a partir de una llamada a `fork()` de otro proceso.

- Pueden desbloquearse por cualquier proceso.
- Además de poder usarse para proteger una sección crítica permiten controlar determinadas condiciones por su contador configurable.
- NO necesitan alojarse en una zona de memoria común mediante la técnica de MEMORIA COMPARTIDA.

4 Ejercicios prácticos

4.1 Usando semáforos binarios o mutexes

A continuación se exponen una serie de ejercicios a realizar mediante hebras y *mutexes*.

4.1.1 Ejercicio 1

Una tienda que vende camisetas guarda en una base de datos (*buffer*) las cantidades de camisetas según el modelo. Por ejemplo, un *buffer* de camisetas[5] indica que existen 5 modelos de camisetas y cada elemento de este *buffer* guarda las cantidades iniciales de cada una de ellas.

Implementar un programa que genere N clientes y M proveedores (la misma cantidad de proveedores que modelos de camiseta). Pruebe primero a hacerlo con un *buffer* de tamaño 5 (camisetas[5]) y después de manera que el número de modelos se pida desde línea de argumentos.

Para **simular una compra**, cada hilo Cliente debe generar un valor aleatorio para el modelo de camiseta y otro para la cantidad a comprar. Esta cantidad debe decrementar el *stock* de la camiseta en cuestión.

Para **simular un suministro**, cada Proveedor debe hacer lo mismo que el Cliente pero en este caso, incrementando el *stock* de la camiseta en cuestión.

Utilice semáforos binarios para resolver este problema de concurrencia imprimiendo el *buffer* antes de generar los hilos y al final del programa para comprobar que se ha ejecutado correctamente.

4.1.2 Ejercicio 2

Implemente un programa que cree un número N de hilos, donde N será un argumento al programa por la línea de comandos.

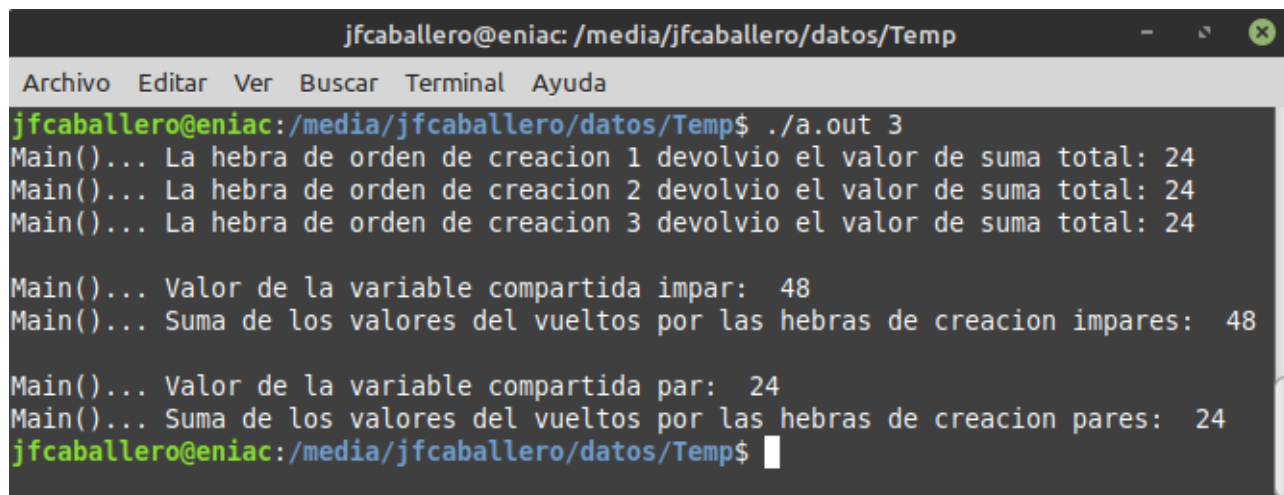
Tenga en cuenta los siguientes supuestos y condiciones:

- Los N hilos comparten dos variables, una llamada "par" e "impar" inicializadas a 0.
- Los hilos deben recibir un valor entero de 1 a N que indica el orden en el cuál se ha creado el hilo. Así, el primer hilo creado recibe 1, el segundo 2, etc.
- Cada hilo debe generar 5 números aleatorios entre 0 y 10.
- Los hilos de índice de creación par deben ir sumando a la variable "par" los números que generan.
- Los hilos de índice de creación impar deben ir sumando a la variable "impar" los números

que generan.

- Una vez finalizado cada hilo, éste debe devolver al programa principal la suma total de números que ha generado.
- El programa principal debe mostrar la suma devuelta por cada hebra, el valor de las variables “par” e “impar” y la suma total devuelta por las hebras de índice de creación pares e impares.

Ejemplo de salida del ejercicio:



```
jfcaballero@eniac: /media/jfcaballero/datos/Temp
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jfcaballero@eniac:/media/jfcaballero/datos/Temp$ ./a.out 3
Main()... La hebra de orden de creacion 1 devolvio el valor de suma total: 24
Main()... La hebra de orden de creacion 2 devolvio el valor de suma total: 24
Main()... La hebra de orden de creacion 3 devolvio el valor de suma total: 24

Main()... Valor de la variable compartida impar:  48
Main()... Suma de los valores del vueltos por las hebras de creacion impares:  48

Main()... Valor de la variable compartida par:  24
Main()... Suma de los valores del vueltos por las hebras de creacion pares:  24
jfcaballero@eniac:/media/jfcaballero/datos/Temp$
```

4.2 Usando semáforos generales

A continuación se exponen una serie de ejercicios a realizar mediante hebras y semáforos generales.

4.2.1 Ejercicio 3

En concurrencia, el problema del productor-consumidor²⁰ es un problema típico, y su enunciado general es el siguiente:

Hay un proceso generando algún tipo de datos (registros, caracteres, aumento de variables, modificaciones en arrays, modificación de ficheros, etc) y poniéndolos en un *buffer*. Hay un 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). Estaríamos hablando de la sección crítica.

Si suponemos que el *buffer* es limitado y está completo, el productor debe esperar hasta que el consumidor lea al menos un elemento para así poder seguir almacenando datos. En el caso de que el *buffer* esté vacío el consumidor debe esperar a que se escriba información nueva por parte del productor.

²⁰ http://es.wikipedia.org/wiki/Problema_Productor-Consumidor

A continuación se muestra una solución en pseudocódigo:

```
/* programa productor consumidor */
semaphore s = 1;
semaphore n = 0;
semaphore e = /* tamaño del buffer */;
void productor()
{
    while (true)
    {
        producir();
        semWait(e);
        semWait(s);
        anyadir();
        semSignal(s);
        semSignal(n);
    }
}
void consumidor()
{
    while (true)
    {
        semWait(n);
        semWait(s);
        extraer();
        semSignal(s);
        semSignal(e);
        consumir();
    }
}
void main()
{
    paralelos (productor, consumidor);
}
```

Figura 5.13. Una solución al problema productor/consumidor con *buffer* acotado usando semáforos.

Una vez haya estudiado detenidamente el problema del productor-consumidor:

- a) Implemente el problema para hilos, teniendo en cuenta que la sección crítica va a ser un *array* de enteros con una capacidad de 5 elementos y donde habrá un productor y un consumidor. Se podrán producir-consumir 10 elementos.
- b) Implemente el problema para hilos, teniendo en cuenta que la sección crítica va a ser un *array* de enteros con una capacidad de 3 elementos y donde habrá 3 productores y 3 consumidores, que podrán producir-consumir 1 elemento cada uno.