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

ASIGNATURA *SISTEMAS OPERATIVOS*

2º DE GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 3

Comunicación entre procesos ligeros – Semáforos binarios y semáforos generales

Profesorado: Juan Carlos Fernández Caballero
Enrique García Salcines

Índice de contenido

1 Objetivo de la práctica

La presente práctica persigue familiarizar al alumnado con la comunicación y sincronización entre procesos ligeros o hilos 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 y semáforos binarios o *mutexs*.

La segunda parte de la práctica se dedica a explicar brevemente la 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 librería *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). Estos son algunos de los aspectos que se también se valorarán y se tendrán en cuenta en el examen práctico de la asignatura.

No olvide que debe consultar siempre que lo necesite el estándar POSIX http://pubs.opengroup.org/onlinepubs/9699919799/ y la librería 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 que proporcione el sistema que esté utilizando. POSIX (*POSIX Shared Memory)*[1] proporciona también un estándar para ello que implementan los sistemas basados en el.

3 Conceptos teóricos

3.1 Concurrencia

Un sistema operativo multitarea permite que coexistan varios procesos activos a la vez (a nivel de proceso o a nivel de hilo), es decir, varios procesos que se están ejecutando de forma concurrente,

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

paralela. En el caso de que exista un solo procesador con un solo núcleo, 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. 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 (uniprocesador y multiprocesador), 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 problemas de concurrencia. Estos problemas de concurrencia hacen que se produzcan inconsistencias en los recursos compartidos por 2 o más procesos (o hilos indistintamente). 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** . Vease un ejemplo:

```
Hilo 1
void *funcion_hilo_1(void *arg)
{
    int resultado;
    ...
    if (i == valor_cualquiera) {
         ...
         resultado = i *
(int)*arg;
         ...
    }
     pthread_exit(&resultado);
}
```

```
Hilo 2
void *funcion_hilo_2(void *arg)
{
    int otro_resultado;
    ...
    if (funcion_sobre_arg(*arg) ==
0) {
         ...
         i = *arg;
         ...
    }
   pthread_exit(&otro_resultado);
}
```

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

Supongamos que el hilo 1 se empieza a ejecutar antes que el hilo 2, y que casualmente se produce un cambio de contexto (el sistema operativo suspende la tarea o hilo actual y pasa a ejecutar la siguiente) justo después de la línea que dice "*if (i == valor_cualquiera)"*. La entrada en ese *if* se producirá si se cumple la condición, que suponemos que sí. Pero justo después de ese momento el sistema hace el cambio de contexto comentado anteriormente y pone a ejecutar al hilo2, que se ejecuta el tiempo suficiente como para ejecutar la línea "*i = *arg"*. 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 programas.**

El problema que tienen estos *bugs* es que son difíciles de detectar en el caso de que no nos fijemos, a la hora de implementar nuestro código, 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.

Aquí tiene otro ejemplo sobre problemas de concurrencia de procesos. Considere el siguiente procedimiento (suponga un sistema operativo multiprogramado para un monoprocesador, pero esto es extensible a varios procesadores):

```
void eco ()
{
        cent = getchar();
        csal = cent;
        putchar(csal);
}
```

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.

Considere ahora que se tiene un sistema multiprogramado de un único procesador y para un único usuario (es extensible a multiprocesadores). El usuario 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*, tiene sentido que éste sea 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 compartición de memoria principal entre procesos es útil para permitir una interacción eficiente y próxima entre los procesos. No obstante, esta interacción puede acarrear problemas.

Considere la siguiente secuencia:

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.

Así, el primer carácter se pierde y el segundo se muestra dos veces. 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.

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 involucra la memoria compartida, los archivos compartidos y cualquier otra cosa que se comparta en el sistema, es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo o mediante multiplexaciones no controladas. Esa parte del programa en la que se accede a un recurso compartido de le conoce como **región crítica** o **sección crítica**. Por tanto, un proceso está en su sección critica 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 una variable o archivo 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 que opera sobre algún recurso *Ra*, 2) código adicional que precede y sucede a la sección crítica y que no involucra acceso a *Ra*. Dado que todos los procesos acceden al mismo recurso *Ra*, se desea que sólo un proceso esté en su sección crítica al mismo tiempo.

Para aplicar exclusión mutua se proporcionan dos funciones o mecanismos abstractos: *entrarcritica()* y *salircritica()*. 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.

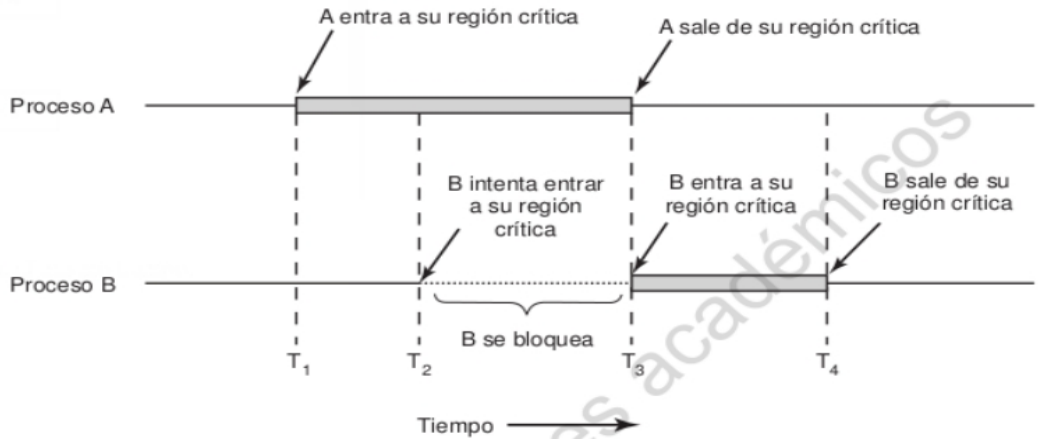
```
/* PROCESO 1 */
void P1
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

```
/* 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 */;
    }
}
```

Faltaría examinar mecanismos específicos para proporcionar los mecanismos *entrarcritica()* y *salircritica()*. Así, si dos procesos A y B intentasen entrar en una sección crítica, la solución proporcionada debería permitir que la exclusión mutua reflejada en la siguiente figura se hiciera correctamente:



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 Dekker[2], Peterson[3] y Lamport[4], aunque existen muchas más soluciones de este tipo (Hyman, Knuth, Kesell,...). Hay también técnicas hardware (inhabilitación de interrupciones e instrucciones máquina especiales) para problemas de exclusión mutua, pero en esta práctica solo trataremos soluciones software[5] y veremos que ante los algoritmos clásicos de exclusión mutua hay soluciones o mecanismos mucho más efectivos (*mutexs*, variables de condición y semáforos).

3.5 Cerrojos o barreras mediante semáforos binarios o mutexs

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 (Dekker, Peterson, Lamport, etc). Estos mecanismos de *pthread* se llaman *mutexs* y variables de condición aunque también se conocen como semáforos binarios.

Un *mutex* (*Mutual Exclusion*) se asemeja a un semáforo 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: Se pide el bloqueo del semáforo antes de entrar, éste es otorgado al primero que llega, mientras que los demás se quedan bloqueados esperando en una cola (suele ser FIFO) a que el que entró primero libere el bloqueo.

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

Una vez el 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.

La siguiente figura muestra una definición de semáforo binario o *mutex*:

```
struct binary_semaphore {
    enum {cero, uno} valor;
    queueType cola;
};
void semWaitB(binary_semaphore s)
{
    if (s.valor == 1)
        s.valor = 0;
    else
        {
            poner este proceso en s.cola;
            bloquear este proceso;
    }
}
void semSignalB(binary_semaphore s)
{
    if (esta_vacia(s.cola))
        s.valor = 1;
    else
    {
            extraer un proceso P de s.cola;
            poner el proceso P en la lista de listos;
    }
}
```

Figura 5.4. Una definición de las primitivas del semáforo binario.

A continuación se expondrán brevemente las funciones más utilizadas y que ofrece la librería *pthread* para llevar a cabo exclusión mutua. Posteriormente se expondrá algún ejemplo de su uso. El alumno deberá hacer un estudio más profundo en la Web y en otros recursos bibliográficos de los que dispone en la biblioteca de la Universidad.

3.5.1 Inicialización de un mutex *(pthread_mutex_init)*

Esta función inicializa un *mutex*[6]. 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*[7], y sirve para definir qué tipo de *mutex* queremos:

  -Normal (PTHREAD_MUTEX_NORMAL).

  -Recursivo (PTHREAD_MUTEX_RECURSIVE).

  -Errorcheck (PTHREAD_MUTEX_ERRORCHECK).

6 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_mutex_init.html
7 https://www.sourceware.org/pthreads-win32/manual/pthread_mutexattr_init.html

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

Para crear un mutex diferente al usado por defecto, tendremos que indicarlo previamente a *pthread_mutex_init()*. Busque en la Web información sobre los tipos de *mutex* anteriormente citados.

pthread_mutex_init() devuelve 0 si se pudo crear el *mutex* o distinto de cero si hubo algún error. Use en sus prácticas los tipos de error que considere más interesantes de los que se definen en *OpenGroup* ([EAGAIN], [ENOMEM], [EPERM], [EINVAL]).

También podemos inicializar un *mutex* sin usar la función *pthread_mutex_init(),* basta con declararlo de esta manera (inicializa un *mutex* por defecto). Es como se hacía en antiguas implementaciones de POSIX pero todavía se permite utilizar:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

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

Esta función[8] 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. Use en sus prácticas los tipos de error que considere más interesantes de los que se definen en *OpenGroup*.

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

Esta es la función[9] 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 ya queda disponible para otra hebra.

Retorna 0 como resultado si no hubo error o diferente de 0 si lo hubo. Use en sus prácticas los

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

tipos de error que considere más interesantes de los 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* para hacerlo, los datos podrían corromperse.

Otro punto importante del que el programador se debe responsabilizar es que si una hebra utiliza un *pthread_mutex_lock()* bajo un determinado *mutex* 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()*. Es decir, no debemos permitir que un hilo adquiera un candado y otro hilo diferente sea el que lo libere, excepto cuando se usen variables de condición, las cuales se estudiarán en las siguientes secciones.

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

3.5.4 Destrucción de un mutex *(pthread_mutex_destroy)*

Esta función[10] 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.** 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. Use en sus prácticas los tipos de error que considere más interesantes de los que se definen en *OpenGroup*.

3.5.5 Ejemplos de uso de mutexs

Veamos como se reescribe el código de una de las secciones anteriores usando *mutexs*. En color azul han sido añadidas las líneas que antes no estaban. Como se puede observar, lo que hay que hacer es inicializar el *mutex*, 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.

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.

En la **demo2.c** se realiza un producto escalar entre dos vectores y luego muestra la suma resultante de los elementos del vector producto que se obtendría. El número de hebras usadas y la

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

longitud de vector que utilizará cada una viene definido en las macros NUMTHRDS y VECLEN del ejemplo. Estúdielo, ejecútelo y observe sus resultados.

```
Variables globales:

pthread_mutex_t mutex_acceso;
int i;
```

```
Hilo 1 (Versión correcta)
void *funcion_hilo_1(void *arg)
{
    int resultado;
    ...

pthread_mutex_lock(&mutex_acceso);
    if (i == valor_cualquiera) {
         ... <
         resultado = i * (int)*arg;
         ... <
    }
pthread_mutex_unlock(&mutex_acceso);
     pthread_exit(&resultado);
}
```

```
Hilo 2 (Versión correcta)
void *funcion_hilo_2(void *arg)
{
    int otro_resultado;
    ...
    if (funcion_sobre_arg(*arg) == 0)
      {
        ...

pthread_mutex_lock(&mutex_acceso);
         i = *arg;
pthread_mutex_unlock(&mutex_acceso);
     }
   pthread_exit(&otro_resultado);
}
```

```
int main(void)
{
     ...
     pthread_mutex_init(&mutex_acceso, NULL);
     ...
}
```

3.6 Variables de condición

Las variables de condición proporcionan una ampliación en sincronización de hilos y permiten implementar condiciones más complejas para entrar a una sección crítica, permitiendo bloquear una hebra hasta que ocurra algún suceso. Mientras los *mutexs* implementan la sincronización mediante el control de acceso a los datos, las variables de condición permiten a los hilos una sincronización basada en el valor real de los datos. Un *mutex* evita el acceso en un mismo tiempo a un recurso compartido por múltiples hilos. Una variable de condición, además, permite a un hilo informar a otros hilos sobre los cambios en el estado de una variable o recurso compartido y permite, en el momento del cambio de estado, que otros hilos que esperan bloqueados sigan su ejecución tras dicha notificación. No hay que ver las variables de condición como otra técnica aparte de los *mutex*, sino como una ampliación de éstos últimos para resolver determinados problemas en caso de que se requieran.

Por tanto, las variables de condición se deben usar junto con un *mutex*. El *mutex* proporciona exclusión mutua para acceder a la variable compartida, mientras que la variable de condición se utiliza para indicar cambios de estado.

Veamos un ejemplo: Imaginemos que tenemos una hebra a la que el procesador le ha dado una determinada rodaja de tiempo para su ejecución. Esa hebra está continuamente haciendo lo

siguiente: 1) Bloquea un mutex, 2) comprueba que una variable tenga un determinado valor para entrar en sección crítica y, por último, 3) desbloquea el mutex. Si la rodaja de tiempo otorgada por el procesador es demasiado grande y esa variable no toma el valor que necesita la hebra para hacer la determinada operación, ésta se pasará constantemente poniendo un cerrojo, comprobando y quitando el cerrojo, sin hacer nada productivo. Las variables de condición nos pueden ayudar a que esa espera activa no se produzca y nuestros programas sean más eficientes. Sin variables de condición, el programador podría tener hebras continuamente sondeando si se alcanza una determinada condición para acceder a una sección crítica (hasta que se acabase la rodaja de tiempo), pero con las variables de condición no es necesario. Es un mecanismo apropiado para bloquearse hasta que ocurra cierta combinación de sucesos. *Los semaforos "e" y "n" del problema del productor-consumidor cuyo pseudocódigo se expone en la parte de ejercicios de la práctica podrían implementarse como variables de condición.*

En la **demo3.c** se puede ver el uso general de variables de condición y *mutexs* (las llamadas a función utilizadas se estudiarán en las siguientes secciones). La hebra principal se queda bloqueada hasta que ocurre una determinada condición en función de la evaluación de un predicado por parte de una hebra secundaria, en este caso que la variable compartida "*shared_data*" sea cero. Esta variable "*shared_data*" es decrementada por una hebra creada en la función *main()* o hebra principal. Una de las tareas de la hebra principal es mostrar un mensaje de salida cuando la variable global llegue a cero, pero sin espera activa.

Estudiemos más concretamente los prototipos basados en el estándar POSIX que ofrece la biblioteca *pthread* para resolver problemas como los anteriores.

3.6.1 Inicialización de una variable de condición (int pthread_cond_init())

La siguiente función[11] inicializa una variable de condición:

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

- cond: Identifica la variable de condición a ser inicializada.
- attr: Al igual que con los *mutex* podemos especificar un argumento atributo inicializado previamente y que determina los atributos de la variable de condición. Si este argumento es NULL, se inicializa la variable de condición con unos valores establecidos por defecto.

Esta función devuelve 0 en caso de éxito y un número distinto de 0 en caso de error. Estudie los posibles casos de error de esa llamada en la página de OpenGroup.

3.6.2 Bloqueo de una hebra hasta que se cumpla una condición(pthread_cond_wait())

Para suspender a un proceso ligero hasta que otra hebra señalice una variable condicional se utiliza la siguiente función[12]:

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

11 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_cond_destroy.html
12 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_cond_timedwait.html

- cond: Variable condicional por la cual se esperará bloqueado hasta que otro hilo envíe una señal de desbloqueo de esta condición en base a la evaluación de un predicado.
- mutex: Barrera o *mutex* que se desbloqueará para que otra hebra tenga acceso a la sección crítica.

Lo que ocurre cuando realizamos una llamada a *pthread_cond_wait()* es lo siguiente:

1) Se bloquea la hebra que ha realizado la llamada a *pthread_cond_wait()*.
2) Se desbloquea la barrera especificada por mutex, es decir, se pone a 1.
3) Cuando otra hebra envíe una señal de cambio de estado asociado a la variable de condición *cond* (*invocación de pthread_cond_signal()*), la hebra que estaba esperando en el *pthread_cond_wait()* del punto 1) se libera y se bloquea el *mutex* asociado a ese *pthread_cond_wait(),* es decir, la barrera se pone a 0. Observe que un hilo que está bloqueado en un *pthread_cond_wait()* no se puede desbloquear con un *pthread_mutex_unlock()*, sino que necesita un *pthread_cond_signal()*.

3.6.3 Desbloquear o reactivar un proceso ligero bloqueado en un *pthread_cond_wait() (pthread_cond_signal())*

Para reactivar al menos un proceso ligero (puede haber varios bloqueados) que esté bloqueado en una llamada *pthread_cond_wait()* ante una variable condicional *cond*, se utiliza la siguiente función[13]. Si no hay hebras bloqueadas se devuelve inmediatamente la invocación:

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond);
```

- cond: Variable condicional que mantiene bloqueada a otro proceso ligero.

Esta función devuelve 0 en caso de éxito y un número distinto de 0 en caso de error. No tiene efecto si no hay ningún proceso ligero esperando, en caso contrario se extrae un proceso de la cola.

3.6.4 Destrucción de una variable de condición (pthread_cond_destroy)

Cuando sepa que una variable de condición ya no se va a usar más, podemos liberar los recursos ocupados por la misma con la siguiente función:

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond);
```

- cond: Variable de condición que se destruye solo cuando no hay hebras esperando en ella.

Esta función devuelve 0 en caso de éxito y un número distinto de 0 en caso de error.

Úsela solamente cuando esté seguro de que no haya hebras esperando. Una variable de condición destruida puede volver a ser inicializada con *pthread_cond_init()*.

13 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_cond_signal.html

3.6.5 Más ejemplos de mutex junto con variables de condición

En la **demo4.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()*, decrementando su valor. 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 (hay espera activa).

Ejecute el programa varias veces para ver qué ocurre según el tiempo que el planificador asigne a cada hebra en un instante dado.

El programa puede ejecutarlo 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.

Note que en el ejemplo anterior no hay *pthread_join()*, pero se está controlando la ejecución de las hebras con el bucle "*while (avail > 0)*" y con la condición "*if (done)* ". Aún así podríamos poner un *pthread_join()* fuera del bucle "*for (;;)*", el cual se rompe cuando se han consumido por la hebra principal todos los *items* producidos por las restantes hebras.

En la **demo5.c** se muestra el ejemplo anterior de la espera improductiva pero usando variables de condición para solucionar el problema. Compárelo con la solución adoptada sin utilizar variables de condición, estúdielo en profundidad, ejecútelo y compruebe sus resultados.

3.7 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 *mutexs* que estudió en el apartado anterior, ya que se basan en que 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 de otro.

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:

```
wait (s) /* Operación atómica wait() para un semáforo "s" */
{
        s = s –1;
        if (s < 0)
        {
                Poner proceso en cola;
                Bloquear al proceso;
        }
}
```

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, 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 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.

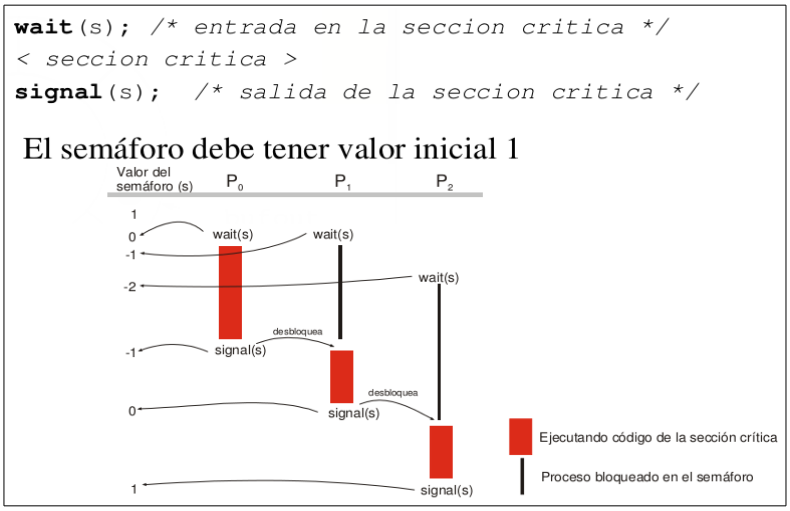
```
signal (s) /* Operación atómica signal() para un semáforo "s" */
{
        s = s + 1;
        if ( s <= 0 )
        {
                Extraer un proceso de la cola bloqueado en la operación wait();
                Poner el proceso listo para ejecutar;
        }
}
```

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()*.

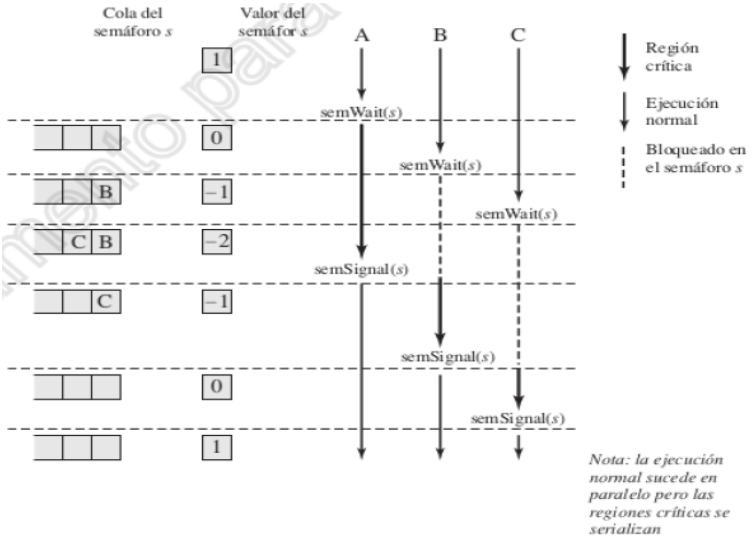
El número de procesos, que en un instante determinado se encuentran bloqueados en una operación *wait()*, viene dado por el valor absoluto del semáforo si es negativo. Cuando un proceso ejecuta la operación *signal()*, el valor del semáforo se incrementa, y en el caso de que haya algún proceso bloqueado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

Para resolver el problema de la sección crítica utilizando semáforos debemos proteger el código que constituye la sección crítica de la siguiente forma (ver figura):

```
wait(s); /* entrada en la seccion critica */
< seccion critica >
signal(s);  /* salida de la seccion critica */
```



La Figura siguiente figura muestra una posible secuencia de tres procesos usando la disciplina de exclusión mutua explicada anteriormente. En este ejemplo, tres procesos (A, B, C) acceden a un recurso compartido protegido por el semáforo "s". El proceso A ejecuta semWait(), y dado que el semáforo tiene el valor 1 en el momento de la invocación, A puede entrar inmediatamente en su sección crítica y el semáforo toma el valor 0. Mientras A está en su sección crítica, ambos B y C realizan una operación semWait() y son bloqueados, pendientes de la disponibilidad del semáforo. Cuando A salga de su sección crítica y realice semSignal(), B, que fue el primer proceso en la cola, podría entonces entrar en su sección crítica. Cuando B haga otro semSignal() y salga de la sección crítica, entonces C podrá entrar en la misma.



El estándar POSIX especifica una interfaz para los semáforos generales (los binarios son los mutexs). Esta interfaz no es parte de la librería *pthread,* es decir, *pthread* no proporciona funciones para el manejo de semáforos generales. A pesar de ello, la mayoría de los sistemas Unix actuales que implementan hilos también proporcionan semáforos generales a partir de otra librería, concretamente la librería *semaphore*, definida en *"semaphore.h"*. Esta librería se basa en el estándar POSIX.

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. Se define en <*sys/sem.h*>. A partir de la versión de kernel 2.6 de los sistemas Linux actuales, los semáforos POSIX se soportan por el sistema operativo y se puede utilizar sin problema *semaphore.h*. Si alguna vez tuviéramos que usar semáforos en máquinas con sistemas operativos muy antiguos, sería más conveniente usar los semáforos *System V*, ya que son más portables a equipos viejos.

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

- **Semáforos sin nombre.** Permiten sincronizar a los procesos ligeros que ejecutan dentro de un mismo proceso, o a los procesos que heredan el semáforo a partir de una llamada *fork()* (padre e hijo). En este segundo caso habría que utilizar técnicas de memoria compartida entre procesos para el acceso a la sección crítica, por lo que en esta práctica nos ceñiremos

al uso de hilos.

- **Semáforos con nombre**. En este segundo tipo de semáforos, éstos llevan asociado un nombre que sigue la convención de nombrado que se emplea para archivos. Con este tipo de semáforos se pueden sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada *fork()*, en este caso los procesos tienen que abrir el semáforo con el mismo nombre. El estándar tiene en cuenta la posibilidad de su utilización para la sincronización de procesos (además de hilos), pero esta posibilidad no está soportada en todas las implementaciones y por ello, en esta asignatura, sólo serán utilizados los semáforos sin nombre.

3.7.1 Inicialización de un semáforo (sem_init())

Los semáforos deben inicializarse antes de usarlos. Para ello utilizaremos la siguiente función[14]:

```
#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 lo hereden por medio de una llamada a *fork()* (!=0). En esta práctica no se contempla el último caso.
- 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 (*[EINVAL], [ENOSPC], [EPERM]*)*. Consulte *OpenGroup*.

3.7.2 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()*[15]:

```
#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, entonces 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*.

14 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_init.html
15 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_wait.html

3.7.3 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()*[16] (equivalente al *semSignal()* teórico comentado al principio de la práctica):

```
#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*.

3.7.4 Examinar el valor de un semáforo (sem_getvalue())

Si necesitamos saber cuál es el valor de un semáforo podemos consultarlo con la siguiente función[17]:

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

- sem: Puntero a parámetro de tipo *sem_t* que identifica el semáforo.
- sval: Puntero a variable de tipo entero en la que se depositará el valor del semáforo.

El uso de *sem_getvalue()* no altera el valor del semáforo. *sval* mantiene el valor que tenía el semáforo en algún momento no especificado durante la llamada, pero no necesariamente el valor en el momento de la devolución. Tenga en cuenta que puede haber más de un hilo intentando acceder al semáforo indicado como primer parámetro, o simplemente, por ejemplo, antes de que *sem_getvalue()* nos devuelva algo un hilo puede haber salido de la sección crítica.

Si el semáforo está bloqueado, *sval* contendrá el valor cero o un valor negativo que indica, en valor absoluto, el número de subprocesos en espera.

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*

16 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_post.html
17 http://pubs.opengroup.org/onlinepubs/9699919799/functions/sem_getvalue.html

3.7.5 Destrucción de un semáforo (sem_destroy())

Para destruir un semáforo previamente creado con *sem_init()* se utiliza la siguiente función[18]:

```
#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.

3.7.6 Ejemplos

La **demo6.c** crea dos hilos. Cada hilo hace exactamente lo mismo, incrementar una variable global un numero de veces. El resultado final debe ser el doble del numero de veces que se incrementa la variable, ya que cada hilo la incrementa el mismo número de veces. Si no hubiera semáforo general, el resultado podría ser inconsistente. Pruebe a ejecutarlo con semáforos y quitando los semáforos.

La **demo7.c** es un programa de sincronización entre hilos que suma los números impares entre 0 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 teniendo en cuenta varios casos en los que el procesador da rodaja de tiempo al hilo P1 o al hilo P2.

La **demo8.c** es la implementación del problema lectores-escritores donde hay 2 lectores y 2 escritores (prioridad a los lectores). 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.

4 Ejercicios prácticos

4.1 Semáforos binarios o mutexs

4.1.1 Ejercicio1

Una tienda que vende camisetas, guarda en la base de datos (buffer) las cantidades de camisetas según el modelo, (camisetas[5], indica que son 5 modelos de camisetas y cada elemento de este buffer guarda las cantidades iniciales de cada una). Realizar un programa que genere N clientes y M proveedores (lo misma cantidad de proveedores que modelos de camiseta).

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.

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

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 Ejercicio2

Cree un programa a partir del cual se creen tres hilos diferentes. Cada hilo debe escribir un carácter en pantalla 5 veces seguidas. Estructure su programa de manera que no haya intercalado de caracteres entre los hilos, lo cuales deben estar ejecutándose de forma paralela (ojo, no lo haga secuencialmente). El *main* o hebra principal también debe escribir en pantalla. De esta forma, una posible salida por pantalla podría ser esta:

*****?????+++++-----

Evidentemente, la sección crítica será la salida estándar del sistema. Probablemente necesitará usar la función fflush() entre escrituras. Ponga también un *sleep(1)* entre ellas para observar las salidas por pantalla.

4.1.3 Ejercicio3

En concurrencia, el problema del productor-consumidor[19] 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.

Existen variaciones del problema productor-consumidor, como por ejemplo en el caso de que el *buffer* sea ilimitado. Consulte la Web si desea obtener más información sobre ello.

Una vez haya estudiado más detenidamente el problema del productor-consumidor realice las siguientes tareas:

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. Haga una implementación usando mutexs pero no variables de condición, por lo que se producirá espera activa en casos en los que no haya sitio donde producir o no haya items que consumir.

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

b) Modifique el apartado a) usando variables de condición junto con mutexs para evitar espera activa. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente porque al menos hay un hueco donde poner una producción. En caso contrario si el *buffer* se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo. Consulte la documentación de clases de teoría si lo considera oportuno. 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.

4.1.4 Ejercicio4

Implemente un programa que a partir de un fichero de texto, realice concurrentemente tantas copias del mismo como se indique. El programa recibirá por la linea de argumentos dos parámetros, el nombre del fichero y el número de copias a realizar. El nombre de los ficheros de salida será el mismo que el del fichero original seguido por un número de copia, por ejemplo: “fichero0”, “fichero1”, “fichero2” en el caso de que haya que hacer tres copias del fichero original “fichero”.

Haga una hebra que se encargue de leer del fichero original, y tantas hebras más como copias del fichero haya que hacer. La hebra que lee el fichero original irá cargando caracteres en un buffer habilitado para ello y de un tamaño fijo (por ejemplo, un array de char de tamaño 1024). El resto de hebras irán copiando bloques de memoria en su copia de fichero que tienen que crear. Utilice mutexs y variables de condición para resolver el problema.

4.2 Semáforos generales

4.2.1 Ejercicio5

Resuelva el problema del productor-consumidor con un buffer circular y acotado usando semáforos generales en vez de semáforos binarios y variables de condición. Haga su programa genérico para que se pueda indicar el tamaño del buffer y la cantidad de datos a producir-consumir.

4.2.2 Ejercicio6

Otro problema más presentado por Dijkstra es el de los filósofos comensales. Cinco filósofos viven en una casa, donde hay una mesa preparada para ellos. Básicamente, la vida de cada filósofo consiste en pensar y comer, y después de años de haber estado pensando, todos los filósofos están de acuerdo en que la única comida que contribuye a su fuerza mental son los espaguetis. Se presentan las siguientes suposiciones y/o restricciones:

- Debido a su falta de habilidad manual, cada filósofo necesita dos tenedores para comer los espaguetis, cogiendo solo un tenedor a la vez (en cualquier orden, izquierda-derecha o derecha-izquierda).
- La disposición para la comida es simple (ver figura): una mesa redonda en la que está colocado un gran cuenco para servir espaguetis, cinco platos, uno para cada filósofo, y cinco tenedores.
- Un filósofo que quiere comer utiliza,en caso de que estén libres, los dos tenedores situados a cada lado del plato, retira y come algunos espaguetis.

- Si cualquier filósofo toma un tenedor (solo un tenedor a la vez) y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda tomar el otro tenedor, para luego empezar a comer.
- Si un filósofo piensa, no molesta a sus colegas. Cuando un filósofo ha terminado de comer

suelta los tenedores y continua pensando.

- Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.

- Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo (o a su izquierda), entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Se produce un interbloqueo (deadlock o abrazo de la muerte). Entonces los filósofos se morirán de hambre.

El problema: diseñar un ritual (algoritmo) que permita a los filósofos comer. El algoritmo debe satisfacer la exclusión mutua (no puede haber dos filósofos que puedan utilizar el mismo tenedor a la vez) y evitar el interbloqueo e inanición (en este caso, el término inanición tiene un sentido literal, además de algorítmico). A partir del siguiente esquema de pseudocódigo y del que está disponible en Moodle, además de consultar la Web7 para informarse en profundidad del problema, implemente una solución en C utilizando semáforos que resuelva el problema de sincronización e inanición de la cena de los filósofos.

PseudoCódigo

```
#define N               5           /* número de filósofos */
#define IZQ         (i - 1) % N     /* número del vecino izq.  de i */
#define DER         (i + 1) % N     /* número del vecino der. de i */
#define PENSANDO    0               /* el filósofo está pensando */
#define HAMBRIENTO  1               /* el filósofo está hambriento */
#define COMIENDO    2               /* el filósofo está comiento */

int estado[N];                      /* vector para llevar el estado de los filósofos */
semáforo exmut = 1;                 /* exclusión mutua para las secciones críticas */
semáforo s[N];                      /* un semáforo por filósofo */
```

```
void filósofo (int i)               /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
while (1) {                         /* se ejecuta eternamente */
    pensar ( );                     /* el filósofo está pensando */
    coger_tenedores (i);            /* obtiene dos tenedores, bloqueándose si no puede */
    comer ( );
    dejar_tenedores (i);            /* deja ambos tenedores en la mesa */
}
}
```

```
void coger_tenedores (int i)        /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
wait (&exmut);                      /* entra en la sección crítica */
estado[i] = HAMBRIENTO;             /* registra el hecho de que el filósofo i tiene hambre */
prueba(i);                          /* intenta coger dos tenedores */
signal(&exmut);                     /* sale de la sección crítica */
wait(&s[i]);                        /* se bloque si no consiguió los tenedores */
}
```

```
void dejar_tenedores (int i)        /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
wait (&exmut);                      /* entra en la sección crítica */
estado[i] = PENSANDO;               /* registra el hecho de que el filósofo i ha dejado de comer */
prueba(IZQ);                        /* comprueba si el vecino izquierdo puede comer ahora */
prueba(DER);                        /* comprueba si el vecino derecho puede comer ahora */
signal(&exmut);                     /* sale de la sección crítica */
}
```

```
void prueba (int i)                 /* i: de qué filósofo se trata (desde 0 hasta N - 1) */
{
if (estado[i] == HAMBRIENTO && estado[IZQ] != COMIENDO
      && estado[DER] != COMIENDO)
{
    estado[i] = COMIENDO;
    signal(&s[i]);
}
}
```