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 – Algoritmos clásicos, mutexs (semáforos binarios) y variables de condición

Profesorado: Juan Carlos Fernández Caballero
Alberto Cano Rojas

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. Al igual que en las anteriores prácticas, en una primera parte se dará una breve introducción teórica sobre el problema de la concurrencia y sus posibles soluciones con algoritmos clásicos y barreras o mutexs, siendo en la segunda parte de la misma cuando, mediante programación en C, se practicarán los conceptos sobre esta temática, utilizando las rutinas de interfaz del sistema que proporcionan a los programadores la librería *pthread* 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.

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 UNIX.

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, 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

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

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, a esto se le conoce como **condiciones de carrera** . Veamos 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*, y (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.

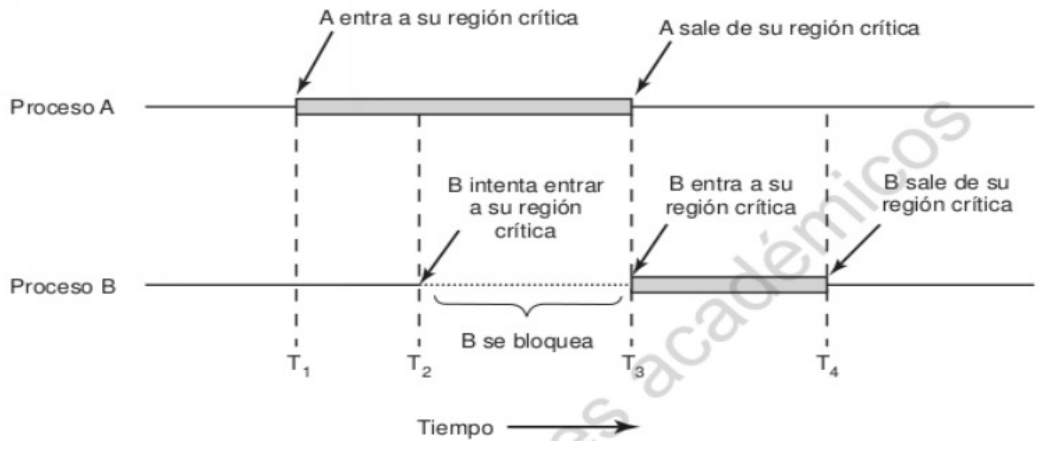
```
/* PROCESO 1 */
void P1
{
    while (true)
    {
        /* código anterior */;
        entrarcritica (Ra);
        /* sección crítica */;
        salircritica (Ra);
        /* código posterior */;
    }
}
```

```
/* 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).

A continuación se expone como ejemplo de este tipo de metodologías clásicas el pseudocódigo del algoritmo de exclusión mutua de Dekker para 2 procesos. El algoritmo de Dekker permite a dos procesos compartir recursos sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra. Si ambos procesos intentan acceder a una sección crítica simultáneamente, el algoritmo elige un proceso según una variable *turno*. Si el otro proceso está ejecutando en su sección crítica, deberá esperar su finalización. El problema que presenta el algoritmo de Dekker se conoce como espera activa, es decir, cuando un proceso está intentando acceder a un recurso que ya ha sido asignado a otro proceso continua consumiendo tiempo de procesador en su intento para conseguir dicho recurso. Esto provoca uso del procesador sin ningún beneficio. La solución eficiente a este problema es el uso de semáforos (se estudiarán posteriormente), que duermen a los procesos y estos sólo consumen memoria.

Algoritmo de Dekker para 2 procesos:

```
boolean p1_puede_entrar, p2_puede_entrar;
int turno;

Proceso1()
{
   while( true )
   {
      [REALIZA_TAREAS_INICIALES]  /* Si las hubiera */
      p1_puede_entrar = true;
      while( p2_puede_entrar )
      {
         if( turno == 2 )  /* Linea 12 */
         {
           p1_puede_entrar = false;
           while( turno == 2 ){}   /* Espera activa, no se hace nada. Linea 15 */
          p1_puede_entrar = true;
         }
      }
      [REGION_CRITICA]  /* Acceso al recurso compartido */
      turno = 2;   /* Linea 20 */
      p1_puede_entrar = false;
      [REALIZA_TAREAS_FINALES] /* Si las hubiera */
   }
 }
```

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

```
Proceso2()
{
  while( true )
  {
     [REALIZA_TAREAS_INICIALES] /* Si las hubiera */
     p2_puede_entrar = true;
     while( p1_puede_entrar )
     {
        if( turno == 1 ) /* linea 34 */
        {
           p2_puede_entrar = false;
           while( turno == 1 ){} /* Espera activa, no se hace nada. Linea 37 */
           p2_puede_entrar = true;
        }
     }
     [REGION_CRITICA] /* Acceso al recurso compartido */
     turno = 1;  /* Linea 42 */
     p2_puede_entrar = false;
     [REALIZA_TAREAS_FINALES] /* Si las hubiera */
  }
}

iniciar()
{
  p1_puede_entrar = false;
  p2_puede_entrar = false;
  turno = 1;
  Proceso1();  // Estos dos procesos se deben lanzar en paralelo, ya sea mediante procesos o mediante hebras.
  Proceso2();
}
```

En el algoritmo representado se realizan las tareas iniciales, luego se verifica si hay otro procesos que puede entrar, si lo hay se entra al ciclo y si es el turno de algún otro proceso (linea 12 y 34) cambia su estado a ya no puede entrar a la sección critica, comprobando nuevamente si es el turno de algún otro proceso (linea 15 y 37). Si lo es, se queda en espera activa (comprobaciones sucesivas, consumiendo CPU) hasta que se da un cambio de turno. Entonces retoma a verdadero su estado de poder entrar a la sección critica, la realiza y da turno a otro proceso poniendo su estado a *false* y realiza sus tareas finales (linea 20 y 42).

3.5 Rutinas de la biblioteca *pthread* para problemas de exclusión mutua

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.

3.5.1 Cerrojos o barreras mediante mutexs

La biblioteca *pthread* nos ofrece unos mecanismos básicos, pero muy útiles, para establecer mecanismos de sincronización y exclusión mutua. Estos mecanismos son una especie de **semáforos binarios** con dos estados llamados ***mutexs*, cerrojos,** o **barreras**, como cada uno quiera llamarlo.

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. 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.

3.5.1.1 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)
```

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

- 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).

  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;
```

3.5.1.2 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*.

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

3.5.1.3 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 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.

3.5.1.4 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.

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

3.5.1.5 Ejemplos de uso de mutexs

Veamos como se reescribe el código de la sección 3.2 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.

```
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);
     ...
}
```

A continuación 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. La parte en rojo corresponde a la sección crítica:

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

int glob = 0;   //GLOBAL VARIABLE
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;   //GLOBAL MUTEX

void * threadFunc(void *arg) /* Loop 'arg' times incrementing 'glob' */
{
```

```
        int loops = *((int *) arg);
        int loc, j, s;
        for (j = 0; j < loops; j++)
        {
                s = pthread_mutex_lock(&mtx); /*Lock the mutex*/
                if (s != 0)
                        printf("mutex_lock error...\n");
                /*Critical Section */
                loc = glob;
                loc++;
                glob = loc;
                printf("Thread %ld increasing the global variable...\n", (unsigned long) pthread_self());
                s = pthread_mutex_unlock(&mtx); /*Unlock the mutex*/
                if (s != 0)
                        printf("mutex_unlock error...\n");
        }
        return NULL;
}

int main(int argc, char *argv[])
{
        pthread_t t1, t2;
        int loops, s;

        if(argc!=2)
        {
                printf("Ussage: ./a.out Number_of_loops\n");
                exit(EXIT_FAILURE);
        }

        //loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;
        loops = atoi(argv[1]);

        s = pthread_create(&t1, NULL, threadFunc, &loops);
        if (s != 0)
                printf("pthread_create error...\n");
        s = pthread_create(&t2, NULL, threadFunc, &loops);
        if (s != 0)
                printf("pthread_create error...\n");

        printf("MAIN Thread, Stopping in the join call...\n");
        s = pthread_join(t1, NULL);
        if (s != 0)
                printf("pthread_join error...\n");
        s = pthread_join(t2, NULL);
        if (s != 0)
                printf("pthread_join error...\n");

        printf("MAIN Thread leaves join calls, the global variable is:%d\n", glob);
        exit(EXIT_SUCCESS);
}
```

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

```
/******************************************************************************
* DESCRIPTION:
*   This example program illustrates the use of mutex variables
*   in a threads program. This version performs a dot product (scalar product)
*   from two vectors. The main data is made available to all threads through
*   a globally accessible  structure. Each thread works on a different
*   part of the data. The main thread waits for all the threads to complete
*   their computations, and then it prints the resulting sum.
*   ESTE CÓDIGO REALIZA EL PRODUCTO ESCALAR DE DOS VECTORES Y MUESTRA LA SUMA
*   DE LOS ELEMENTOS RESULTANTES DE ESE PRODUCTO. TODO ELLO SE HACE DISTRIBUYENDO
*   EL TRABAJO ENTRE HEBRAS.
*  SOURCE: Vijay Sonnad, IBM
*  FROM: LAST REVISED: 01/29/09 Blaise Barney
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMTHRDS 3
#define VECLEN 3

/*
The following structure contains the necessary information to allow the function "dotprod" to access its input data and place its output into the structure. */

typedef struct
{
  int     *a;
  int     *b;
  int     sum;
  int     veclen;
} DOTDATA;

/* Define globally accessible variables and a mutex */
DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;

/*
The function dotprod is activated when the thread is created. As before, all input to this routine is obtained from a structure of type DOTDATA and all output from this function is written into this structure. The benefit of this approach is apparent for the multi-threaded program: when a thread is created we pass a single argument to the activated function - typically this argument is a thread number. All  the other information required by the function is accessed from the globally accessible structure.
*/

void *dotprod(void *arg)
{
/* Define and use local variables for convenience */
int i, start, end, len;
long offset;
int mysum, *x, *y;
offset = *(long *)arg;
```

```
len = dotstr.veclen;
start = offset*len;
end   = start + len;
x = dotstr.a;
y = dotstr.b;

/*
Perform the dot product and assign result to the appropriate variable in the structure.
*/
mysum = 0;
for (i=start; i<end ; i++)
  mysum += (x[i] * y[i]);

/*
Lock a mutex prior to updating the value in the shared structure, and unlock it upon updating.
*/
pthread_mutex_lock (&mutexsum);
dotstr.sum += mysum;
printf("\nThread %ld did %d to %d:  mysum=%d global sum=%d\n",offset,start,end-1,mysum,dotstr.sum);
pthread_mutex_unlock (&mutexsum);

pthread_exit((void*) 0);
}

/*
The main program creates threads which do all the work and then print out result upon completion. Before creating the
threads, The input data is created. Since all threads update a shared structure, we need a mutex for mutual exclusion.
The main thread needs to wait for all threads to complete, it waits for each one of the threads. We specify a thread
attribute value that allow the main thread to join with the threads it creates. Note also that we free up handles  when
they are no longer needed.
*/

int main (int argc, char *argv[])
{
long i[NUMTHRDS],j;
int *a, *b;
void *status;
pthread_attr_t attr;

srand(time(NULL));

/* Assign storage and initialize values */
a = (int*) malloc (NUMTHRDS*VECLEN*sizeof(int));
b = (int*) malloc (NUMTHRDS*VECLEN*sizeof(int));
for (j=0; j<VECLEN*NUMTHRDS; j++)
{
  a[j]=rand()%11;
  b[j]=rand()%11;
}

printf("a:\t");
for (j=0; j<VECLEN*NUMTHRDS; j++)
            printf("%d\t", a[j]);
printf("\nb:\t");
for (j=0; j<VECLEN*NUMTHRDS; j++)
            printf("%d\t", b[j]);
```

```
dotstr.veclen = VECLEN;
dotstr.a = a;
dotstr.b = b;
dotstr.sum=0;

pthread_mutex_init(&mutexsum, NULL);

/* Create threads to perform the dotproduct  */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

/* Each thread works on a different set of data.
 * The offset is specified by 'i'. The size of
 * the data for each thread is indicated by VECLEN.
 */
for(j=0;j<NUMTHRDS;j++)
            i[j]=j;
for(j=0;j<NUMTHRDS;j++)
   pthread_create(&callThd[j], &attr, dotprod, (void *)&i[j]);

pthread_attr_destroy(&attr);
/* Wait on the other threads */

for(j=0;j<NUMTHRDS;j++)
  pthread_join(callThd[j], &status);
  //to complete errors...

/* After joining, print out the results and cleanup */
printf ("Sum =  %d \n", dotstr.sum);
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
}
```

3.5.2 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 el siguiente ejemplo 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:

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

pthread_cond_t  is_zero;
pthread_mutex_t mutex; // Condition variables needs a mutex.
int shared_data = 5; // Or some other large number.

void  *thread_function (void *arg)
{
        pthread_mutex_lock(&mutex) ;
        while ( shared_data > 0)
        {
                --shared_data ;
                printf("Thread - Decreasing shared_data... Value:%d\n",shared_data);
                sleep(1);
        }
        pthread_mutex_unlock(&mutex);
        printf("Thread, loop finished...\n");
        sleep(3);
        // Signal the condition. "shared_data" already is cero. This unlock the Main Thread
        printf("Sending signal to main() thread....shared_data = 0 !!!\n");
        pthread_cond_signal (&is_zero);
        return NULL;
}

int main (void)
```

```
{
        pthread_t thread_ID1;
        void  *exit_status;

        pthread_cond_init (&is_zero , NULL) ;
        pthread_mutex_init (&mutex , NULL) ;
        pthread_create (&thread_ID1 ,NULL, thread_function , NULL) ;

        // Wait for the shared data to reach zero.
        pthread_mutex_lock(&mutex );
        printf("Main thread locking the mutex...\n");
        //Main thread will be locked here.
        while ( shared_data != 0)
        {
                printf("I am the Main thread, shared_data is != 0. I am to sleep and unlock the mutex...\n");
                 /* Elimina la espera activa, se desbloquea mutex  y pasa a lista de bloqueado este hilo*/
                 pthread_cond_wait (&is_zero, &mutex) ;
                /*Cuando se haga un pthread_cond_signal() asociado a is_zero, este hilo se desbloqueará,
                 bloqueando a la vez el semáforo mutex, que se desbloquerá finalmente en el
                 pthread_mutex_unlock(&mutex) que hay tres lineas más abajo*/
        }
        printf("Main awake - Value of shared_data: %d!!!\n", shared_data);
        pthread_mutex_unlock(&mutex) ;

        //To complete handle errors
        pthread_join( thread_ID1 , NULL) ;
       /*Destroy mutex and conditional variable*/
        pthread_mutex_destroy(&mutex) ;
        pthread_cond_destroy (&is_zero) ;
        exit(EXIT_SUCCESS);
}
```

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

3.5.2.1 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.

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

3.5.2.2 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);
```

- 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()*.

3.5.2.3 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.

3.5.2.4 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);
```

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

- 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()*.

3.5.2.5 Más ejemplos de mutex junto con variables de condición

En el siguiente ejemplo hay 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.

```
/*****************************************
Ejemplo de productor-consumidor

Modified from Michael Kerrisk's code, 2014.
*****************************************/
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

typedef enum {FALSE=0, TRUE=1} booleano;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
int avail = 0;

static void * threadFunc(void *arg)
{
        int cnt = atoi((char *) arg);
        int s, j;
        for (j = 0; j < cnt; j++)
        {
                //Lock the mutex
                s = pthread_mutex_lock(&mtx);
                if (s != 0)
                        printf("mutex_lock error...\n");
                /* Let consumer know another unit is available */ /*Critical section*/
                avail++;
                printf("avail variable increased by thread %ld, avail=%d\n", (unsigned long) pthread_self(),avail);
                //Unlock mutex
                s = pthread_mutex_unlock(&mtx);
                if (s != 0)
```

```
                        printf("mutex_unlock error...\n");
        }
        return NULL;
}

int main(int argc, char *argv[])
{

        if(argc<=3)
        {
                printf("Ussage:  ./a.out  Number_of_increasing_for_thread1  Number_of_increasing_for_thread2
Number_of_increasing_for_threadN\n");
                exit(EXIT_FAILURE);
        }

        pthread_t tid;
        int s, j;
        int totRequired;        /* Total number of units that all threads will produce */
        int numConsumed;        /* Total units so far consumed */
        booleano done;

        /* Create all threads */
        totRequired = 0;
        for (j = 1; j < argc; j++)
        {
                totRequired += atoi(argv[j]);  //Sum is the total of increments
                s = pthread_create(&tid, NULL, threadFunc, argv[j]);
                if (s != 0)
                        printf("pthread_create error...\n");
        }

        /* Use a polling loop to check for available units */
        numConsumed = 0;
        done = FALSE;

        for (;;)  //Continuous simulation
        {
                //Lock mutex
                s = pthread_mutex_lock(&mtx);
                if (s != 0)
                        printf("mutex_lock error...\n");

                while (avail > 0)  /* Consume all available units */
                {
                        /* Do something with produced unit */
                        numConsumed ++;
                        avail--;
                        printf("Main, numConsumed=%d\n", numConsumed);
                        //Equivalent to if((numConsumed >= totRequired)==true) done=true else done=false
                        done = numConsumed >= totRequired;
                }
                printf("Unproductive time..., avail is <= 0\n");
                s = pthread_mutex_unlock(&mtx); /*Desbloqueo de la barrera*/
                if (s != 0)
                        printf("mutex_unlock error...\n");

                if (done)
```

```
                {
                        printf("Exiting...All produced items have been consumed\n");
                        break;
                }
                /* Perhaps do other work here that does not require mutex lock */
        }
                exit(EXIT_SUCCESS);
}

/*
The above code works, but it wastes CPU time, because the main thread continually loops, checking the state of the
variable avail. A condition variable remedies this problem. It allows a thread to sleep (wait) until another thread
notifies (signals) itcthat it must do something (i.e., that some "condition" has arisen that the sleeper must now respond
to).cA condition variable is always used in conjunction with a mutex. The mutexccprovides mutual exclusion for
accessing the shared variable, while the conditioncvariable is used to signal changes in the variable's state.
*/
```

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.

A continuación 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.

```
/*****************************************
Modified from Michael Kerrisk's code, 2014.
*****************************************/
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

typedef enum {FALSE=0, TRUE=1} booleano;
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int avail = 0;

void * threadFunc(void *arg)
{
        int cnt = atoi((char *) arg);
        int s, j;
        for (j = 0; j < cnt; j++)
        {
                //lock mutex
                s = pthread_mutex_lock(&mtx);
                if (s != 0)
                        printf("mutex_lock error...\n");
                /* Let consumer know another unit is available */
                avail++;
                printf("avail variable increased by thread %ld, avail=%d\n", (unsigned long) pthread_self(),avail);
                //Unlock mutex
                s = pthread_mutex_unlock(&mtx);
```

```
                if (s != 0)
                        printf("mutex_unlock error...\n");
                /* Wake sleeping consumer */
                s = pthread_cond_signal(&cond);
                if (s != 0)
                        printf("pthread_cond_signal error...\n");
                printf("Sending cond_signal from thread %ld...\n", (unsigned long) pthread_self());
        }
        return NULL;
}

int main(int argc, char *argv[])
{
        pthread_t tid;
        int s, j;
        int totRequired; /* Total number of units that all threads will produce */
        int numConsumed; /* Total units so far consumed */
        booleano done;

        if(argc<=3)
        {
                printf("Ussage:  ./a.out  Number_of_increasing_for_thread1  Number_of_increasing_for_thread2
Number_of_increasing_for_threadN\n");
                exit(EXIT_FAILURE);
        }
        /* Create all threads */
        totRequired = 0;
        for (j = 1; j < argc; j++)
        {
                totRequired += atoi(argv[j]);
                s = pthread_create(&tid, NULL, threadFunc, argv[j]);
                if (s != 0)
                    printf("pthread_create error...\n");
        }

        /* Loop to consume available units */
        numConsumed = 0;
        done = FALSE;
        for (;;) //Continuous simulation
        {
                //lock mutex
                s = pthread_mutex_lock(&mtx);
                if (s != 0)
                        printf("mutex_lock error...\n");
                while (avail == 0)
                {
                        /* Wait for something to consume */
                        printf("Main  entering  in  pthread_cond_wait()  call....  Main  will  be  locked  by  the
planner!!!\n");
                        s = pthread_cond_wait(&cond, &mtx);
                        if (s != 0)
                                printf("pthread_cond_wait error...\n");
                        printf("Sleeping time finished. Main state is Running. No Unproductive time while avail is
<= 0!!!\n");
                }
                while (avail > 0) /* Consume all available units */
                {
```

```
                        /* Do something with produced unit */
                        numConsumed ++;
                        avail--;
                        printf("Main, numConsumed=%d\n", numConsumed);
                        done = numConsumed >= totRequired;
                }
                //Unlock mutex
                s = pthread_mutex_unlock(&mtx);
                if (s != 0)
                        printf("mutex_unlock error...\n");

                if (done)
                {
                        printf("Exiting...All produced items have been consumed\n");
                        break;
                }
          /* Perhaps do other work here that does not require mutex lock */
        }
        exit(EXIT_SUCCESS);
}
```

4 Ejercicios prácticos

4.1 Ejercicio1

a) Busque información en la Web información sobre el algoritmo de Peterson[14] y hágalo general para *N* hilos (en Moodle dispone de una versión de pseudocódigo para 2 procesos). Implemente un programa de prueba que muestre que la concurrencia y el acceso a la sección crítica de algún recurso compartido por *N* hilos se realiza correctamente. La sección crítica puede ser aquello que considere oportuno: variables, fichero, array, etc.

b) (OPCIONAL) Infórmese en la Web sobre el algoritmo de la panadería de Lamport[15] para N procesos (en Moodle dispone de una versión de pseudocódigo), estúdielo, y usando la información accedida implemente una solución en C para hilos.

4.2 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.

14 http://en.wikipedia.org/wiki/Peterson%27s_algorithm
15 http://es.wikipedia.org/wiki/Algoritmo_de_la_panader%C3%ADa_de_Lamport

4.3 Ejercicio3

En concurrencia, el problema del productor-consumidor[16] 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.

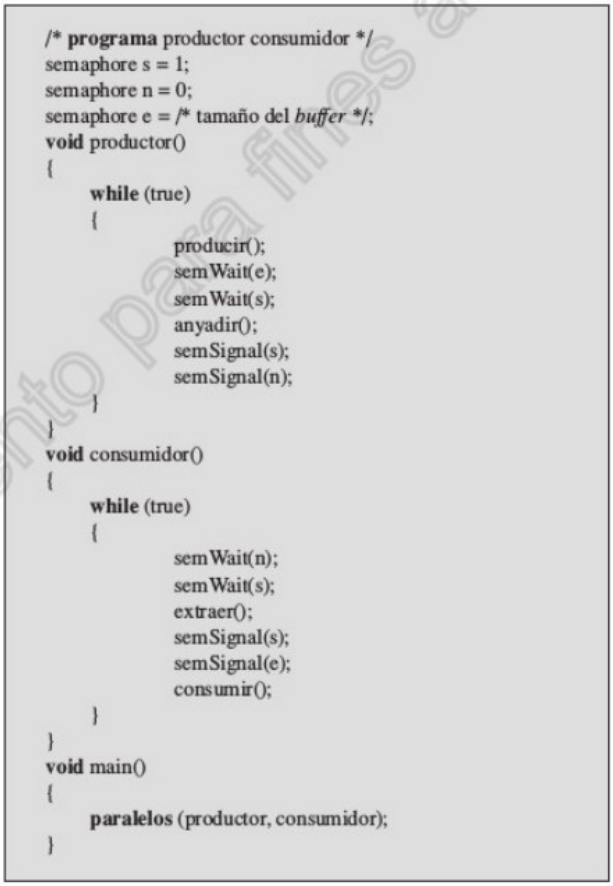
Existen variaciones del problema productor-consumidor, como por ejemplo en el caso de que el *buffer* sea ilimitado. Consulte la Web si desea obtener más información sobre ello.

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.

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:

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



```
/* 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.

4.4 Ejercicio4

Otro caso típico en sincronización de multi-procesos es el problema de los lectores-escritores[17], que se describe como sigue:

Hay un objeto de datos (fichero de texto, base de datos, un bloque principal de memoria) que es utilizado por varios procesos o usuarios, unos que leen y otros que escriben. Solo puede utilizar el recurso un proceso y solo uno, es decir, o bien un proceso estará escribiendo o bien leyendo, pero nunca ocurrirá simultáneamente.

Se considera a cada usuario (lector y escritor) como un proceso distinto, y, por ejemplo, al fichero en cuestión como un recurso. Para que el problema esté bien resuelto se tiene que cumplir:

1) Cualquier número de lectores pueden leer del fichero simultáneamente.
2) Solo un escritor al tiempo puede escribir en el fichero.
3) Si un escritor está escribiendo en el fichero ningún lector puede leerlo.

Para distinguirlo del problema del productor-consumidor, supóngase que el área compartida es un catálogo de biblioteca. Los usuarios ordinarios de la biblioteca leen el catálogo para localizar un libro, pero no eliminan ese catálogo en su lectura. Uno o más bibliotecarios deben poder actualizar el catálogo. En la solución general, cada acceso al catálogo sería tratado como una sección crítica y

17 http://en.wikipedia.org/wiki/Readers%E2%80%93writers_problem

los usuarios se verían forzados a leer el catálogo de uno en uno. Esto claramente impondría retardos intolerables. Al mismo tiempo, es importante impedir a los escritores (bibliotecarios) interferir entre sí y también es necesario impedir la lectura mientras la escritura está en curso para impedir que se acceda a información inconsistente.

Este problema se puede plantear dando prioridad a los lectores sobre escritores o viceversa. Resumidamente, cuando se da prioridad a los lectores, una vez que un lector haya empezado a leer, los demás no tienen que esperar a que éste termine ni tampoco esperar a que escriba un escritor que se ponga en espera de acceder a la sección crítica (esta solución podría dejar a los escritores en inanición, ya que estos deben esperar mientras haya al menos un lector en la sección crítica). Por el contrario, cuando se da prioridad a los escritores, si hay algún lector leyendo u otros lectores quieren hacer lo mismo, y hay un escritor esperando, se le da prioridad al escritor sobre los lectores en el acceso a la sección crítica. Es decir, los escritores deben comenzar a hacer su trabajo tan pronto como sea posible (esta solución puede dejar a los lectores en inanición). Si un escritor está esperando a escribir nadie más debe leer. Si necesita más información consulte la Web.

Una vez haya estudiado detenidamente el problema de los lectores-escritores implemente el problema mediante el uso de hilos dando prioridad a los lectores. Use como sección crítica la estructura de datos que considere oportuna. A continuación tiene un ejemplo en pseudocódigo. Consulte la información de las clases teóricas si lo considera oportuno.

```
/* programa lectores y escritores */
int cuentalect;
semaphore x = 1, sescr = 1;
void lector()
{
     while (true)
     {
          semWait (x);
          cuentalect++;
          if (cuentalect == 1)
               semWait (sescr);
          semSignal (x);
          LEERDATO();
          semWait (x);
          cuentalect—;
          if (cuentalect == 0)
               semSignal (sescr);
          semSignal (x);
     }
}
void escritor()
{
     while (true)
     {
          semWait (sescr);
          ESCRIBIRDATO();
          semSignal (sescr);
     }
}
void main()
{
     cuentalect = 0;
     paralelos (lector, escritor);
}
```

Figura 5.22. Una solución al problema lectores/escritores usando semáforos: los lectores tienen prioridad.

4.5 Ejercicio5

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.