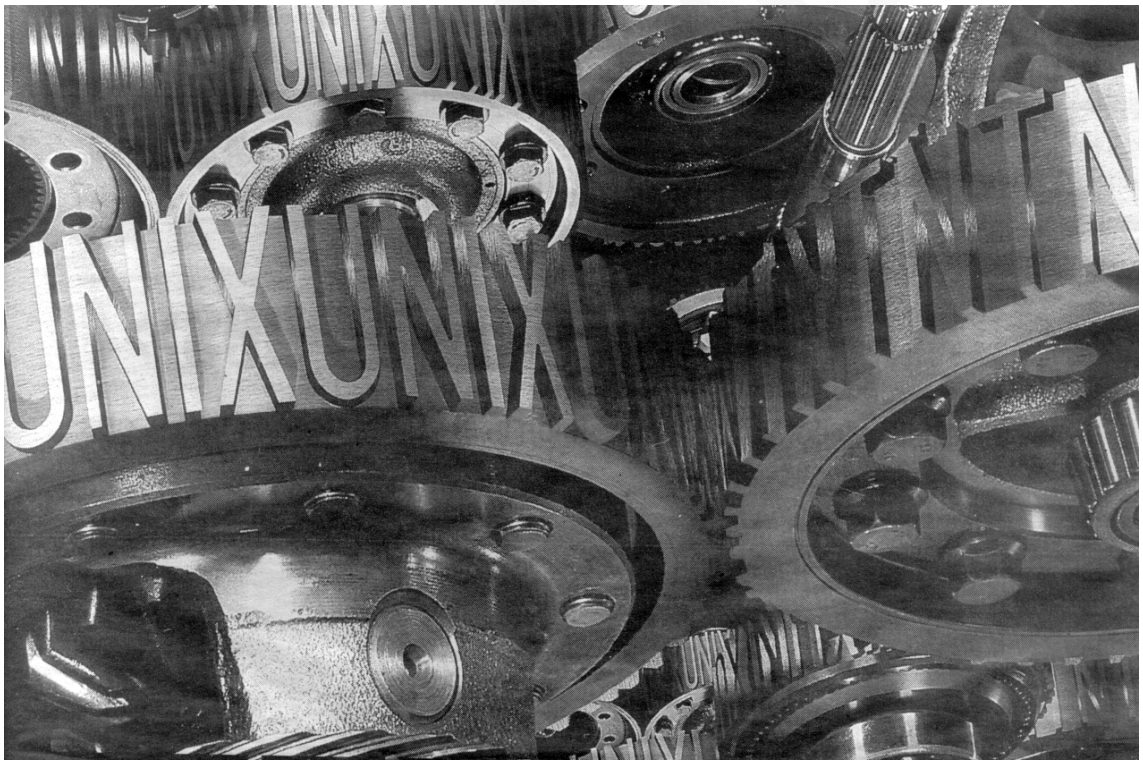
Universidad de Cantabria
Grado de Ingeniería Informática

Sistemas Operativos

Práctica 3 - Parte 2



Rafael Menéndez de Llano Rozas

Guion

3

Sincronización y comunicación entre Hilos

En esta práctica verás cómo los hilos se pueden sincronizar y comunicar a través de memoria compartida.

Sincronización de threads: mutex

Los *mutex* son objetos de sincronización por los que múltiples *threads* acceden de forma mutuamente exclusiva a un recurso compartido. Un *mutex* es un semáforo binario con propietario y sirve para entrar a un recurso y proteger una sección crítica.

Un *mutex* viene definido por una variable de este tipo:

```
pthread_mutex_t
```

Asociada a esa variable están sus atributos (como los *pthreads*) definidos con una variable de tipo:

```
pthread_mutexattr_t
```

El atributo hay que inicializarlo y cuando se acabe destruirlo con:

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

Al igual que los atributos de los hilos, si queremos obtener/cambiar el comportamiento por defecto del *mutex* usaremos:

```
int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                                                    int *restrict type);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

Donde el tipo está definido con alguna de estas constantes de la cabecera `pthread.h`:

1. `PTHREAD_MUTEX_DEFAULT`: El tipo por defecto. Indefinido.
2. `PTHREAD_MUTEX_NORMAL`: No se detecta *deadlock*. Un intento de retomar el *mutex* provocará un bloqueo.
3. `PTHREAD_MUTEX_ERRORCHECK`: Se detectan *deadlock*. Dará error si lo intentamos retomar, así como intentar liberar uno tomado por otro hilo.
4. `PTHREAD_MUTEX_RECURSIVE`: Se puede tomar varias veces (y liberarse las mismas).

Hay que tener en cuenta que no todos los atributos pueden estar en una implementación.

También se puede cambiar el acceso al *mutex*:

```
int pthread_mutexattr_getpshared (const pthread_mutexattr_t restrict *attr,
                                  int * restrict  pshared);
int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared);
```

1. `PTHREAD_PROCESS_SHARED`: hilos que pueden acceder a la memoria donde está *mutex*.
2. `PTHREAD_PROCESS_PRIVATE`: hilos del mismo proceso.

Antes de utilizar un *mutex*, debemos crearlo e inicializarlo, esto se puede hacer de dos maneras: de forma dinámica (con los atributos) o estática (por defecto).

- Para la creación dinámica se usa la llamada:

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

- La creación estática es más rápida (es una asignación de una constante) y sólo se ejecuta una vez (no se chequean errores):

```
pthread_mutex_t mutex = (pthread_mutex_t)PTHREAD_MUTEX_INITIALIZER;
```

En ambos casos el *mutex* está inicializado y desbloqueado. En la práctica si no se pueden poner atributos debido a la implementación, el *mutex* se puede crear dinámicamente por defecto con NULL sin más complicaciones:

```
pthread_mutex_init(&mutex, NULL)
```

Cuando no se necesite más en mutex se puede eliminar con la llamada:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

El siguiente paso será asociar el *mutex* a un recurso o una sección crítica, para lo cual tenemos las operaciones teóricas *wait* y *signal* que se convierten en POSIX en `lock` y `unlock`:

- Para tomar (cerrar) el *mutex* se usa `lock`. Si está libre, se toma, y el p*thread* correspondiente se convierte en propietario; si no, el p*thread* se suspende (o no con `try` que devolverá `busy`) y se añade a una cola asociada al *mutex*. Las dos operaciones son:

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_trylock(pthread_mutex_t *mutex)
```

- Para liberarlo se usa `unlock` (sólo el propietario del *mutex* puede invocar esta operación). Si había algún p*thread* que lo había pedido (`lock`) y por tanto estaba esperando en la cola asociada, se le elimina de ésta y se convierte en el nuevo propietario, si no hay nadie se libera.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Para ver el uso de los *mutex* se propone este ejemplo donde se trata de acceder a una estructura de datos definida como variable global compuesta de tres variables enteras que tienen que ser cambiadas (incrementadas) por un hilo y pintadas por otro. Como se ha visto en teoría, al haber un flujo de control que intenta cambiar el recurso, éste debe ser protegido con un semáforo que se integra por elegancia dentro de la propia estructura. Todo funcionará bien si los valores de a, b y c pintados en pantalla son iguales, si no es que se ha producido una carrera crítica.

```
#include <stdio.h>
#include <pthread.h>
#define LIMITE 100   /* tamano del problema */

struct shared_data {
       pthread_mutex_t mutex;
       int a,b,c;
```

```
} data;

/*****************************************************************/
void * incrementer (void *arg)
{
  int i;
  for (i=1;i<LIMITE;i++)
  {
       pthread_mutex_lock(&data.mutex);  /* protejo */
       data.a++;
       data.b++;
       data.c++;
       pthread_mutex_unlock(&data.mutex);   /* libero */
       sched_yield();   /* para alternar el hilo */
  }
  pthread_exit (NULL);
}

/*****************************************************************/
void * reporter (void *arg)
{
  int i=0;
  do
  {
     pthread_mutex_lock(&data.mutex);   /* cierro semaforo */
     printf("a=%d,b=%d,c=%d\n", data.a, data.b, data.c);
     pthread_mutex_unlock(&data.mutex); /* abro semaforo */
     i++;
     sched_yield();   /* para alternar el hilo */
  } while (i<LIMITE);
  pthread_exit (NULL);
}

/*****************************************************************/
int main()
{
   pthread_t t1,t2;

   data.a = 0; data.b =0; data.c = 0;

   if (pthread_mutex_init(&data.mutex, NULL) != 0)
      perror("mutex_init");

   if (pthread_create(&t1, NULL, incrementer,NULL) != 0)
        perror("pthread_create");

   if (pthread_create (&t2, NULL, reporter, NULL) != 0)
        perror("pthread_create");

   if (pthread_join(t1,NULL) != 0)
     perror("in join");

   if (pthread_join(t2,NULL) != 0)
      perror("in join");

   pthread_exit (NULL);
}
```

1. Bájate el programa “mut.c” y entiéndelo. ¿valdrá siempre lo mismo las variables a, b y c? Cópialo y cambia la forma de crear el semáforo *mutex*, primero de forma estática y segundo creando atributos y poniendo alguna constante para saber si está implementada. Prueba a comentar la cesión del procesador para ver qué pasa.

2. Haz una copia y aumenta hasta tres órdenes de magnitud el LIMITE del programa. Comenta la petición del mutex (`lock` y `unlock`) y haz que el hilo `reporter` pinte sólo un mensaje cuando las variables sean diferentes. ¿se pinta algo? ¿qué indica?

Sincronización de threads: variable condicionales

Las variables condicionales son objetos de sincronización que permiten a un *thread* suspenderse hasta que otro *thread* lo reactiva y se cumple un predicado lógico. Para entender mejor esto, hagámonos las siguientes preguntas cuando un hilo ha tomado un semáforo y entra en una Sección Crítica (SC): ¿qué ocurre si para seguir ejecutándola necesita evaluar una condición? ¿Se quedará esperando con el semáforo tomado? La respuesta lógica es que, si no puede seguir ejecutándose, NO debería seguir teniendo el *mutex* cerrado, ya que esto provocará bloqueos en otros hilos que pidan ese semáforo.

Para evitar esto existen las variables condicionales que ofrecen un mecanismo por el cual:

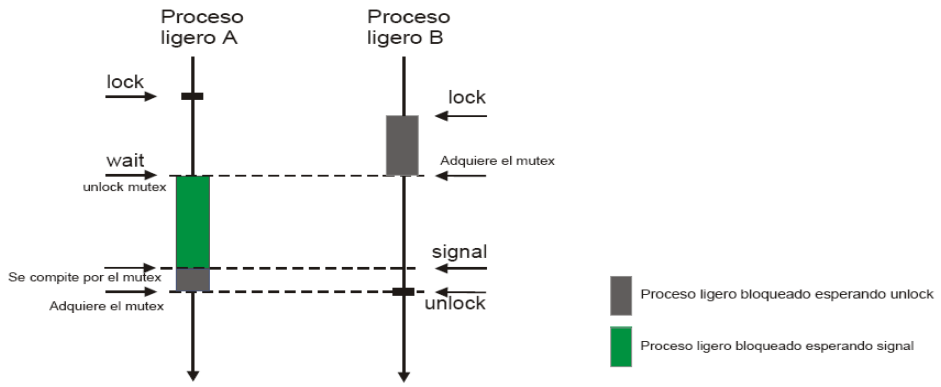
1. Un hilo se puede bloquear esperando a esa condición y a la vez liberar el *mutex* (de forma atómica) para que otros puedan acceder a la SC (recurso).
2. Otro hilo puede tomar el *mutex*, entrar en SC, actuar sobre la condición y liberar a algún hilo que esté esperando por ella.
3. El hilo despertado compite por el *mutex*.
4. Cuando lo tome, comprueba de nuevo si la condición se cumple o no (puede haber ocurrido algo entre medias).
   a) Si no es así, volveremos al punto 1.
   b) Si es así, se hace el trabajo y después se libera el *mutex*.

La condición se representa mediante una expresión booleana. La evaluación de la condición booleana se hace protegida mediante un *mutex* asociado a la variable condicional y que protege a la SC.

Existen dos operaciones atómicas para trabajar con variables condicionales:

- Operación de Espera **WAIT**. Bloquea al hilo. Lleva asociada la variable de condición V y su *mutex*. Dependiendo de la evaluación de la condición, se seguirá tomando el *mutex* o se liberará. Si se libera, se volverá a evaluar la condición cuando se vuelva a poseer el *mutex*.
- Aviso de Activación **SIGNAL**. Liberará del bloqueo al hilo asociado a V. La activación también puede ser hecha en forma de "Broadcast" de una condición: se reactivan todos los threads suspendidos en espera de esa condición.

La sincronización viene reflejada en la siguiente figura donde dos hilos intentan acceder a una SC protegida por un *mutex*, uno de ellos para progresar necesita que se de una determinada condición y otro puede hacer que esa condición cambie dentro de la SC:



El pseudocódigo sería:

- Esperar a una condición: se suspende la tarea hasta que otro *thread* señaliza esa condición; entonces, generalmente se comprueba un predicado lógico y se repite la espera si el predicado es falso.

```
pthread_mutex_lock(mutex);
while (predicado == FALSE)
  WAIT ( cond,  mutex );  /*bloquea al hilo y libera mutex */
  Hago el trabajo en la SC;
pthread_mutex_unlock(un_mutex);
```

- Señalizar una condición: se reactiva uno o más de los threads suspendidos en espera de esa condición.

```
pthread_mutex_lock(un_mutex);
  Hago algo en la SC;
  predicate=TRUE;
  SIGNAL(cond); /* libera a algún hilo bloqueado en la condición */
pthread_mutex_unlock(un_mutex);
```

En cuanto a la programación de variables condicionales en POSIX éstas vienen definidas por una variable de tipo:

```
pthread_cond_t
```

Sobre esa variable (como en los *mutex*) existen una serie de atributos: `pthread_condattr_t` a aplicar (NULL por defecto) que se usan en las operaciones de creación de las variables, que de nuevo pueden ser dinámicas o estáticas (en ambos casos la variable está inicializada):

- Creación dinámica:
```
int pthread_cond_init(pthread_cond_t *restrict cond,
            const pthread_condattr_t *restrict attr);
```

- Creación estática (es más rápido y sólo se ejecuta una vez):
```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Destrucción:
```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Una vez creadas, las operaciones sobre las variables condicionales son las vistas:

- Señalizar una condición: se reactiva uno o más de los *pthreads* suspendidos en espera de esa condición (si no existen, no hace nada).
```
int pthread_cond_signal(pthread_cond_t *cond);
```

- "*Broadcast*" de una condición: se reactivan todos los *pthreads* suspendidos en espera de esa condición.
```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Esperar a una condición: se suspende la tarea hasta que otro *pthread* señaliza esa condición. Está asociada con un *mutex* bloqueado que automáticamente se libera y cuando nos despiertan se intenta coger de nuevo.
```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

Un ejemplo de utilización de las variables condicionales sería el típico problema del productor/consumidor sobre un buffer circular. Cada uno de los dos hilos de este ejemplo tiene esa misión y hay un hilo padre que les crea y establece el entorno. Hay una variable compartida que contiene el número de elementos que existen en el buffer circular y que va a ser leída y escrita por los

dos hilos y por tanto necesita protegerse con una sección crítica con *mutex*. Como puede ocurrir que el productor o el consumidor se encuentren el buffer lleno o vacío y esto produciría su bloqueo dentro de la SC, intervienen dos variables condicionales que representan estas condiciones. Y cada uno de los hilos intervendrá en el contrario para sacarlo de esa condición de bloqueo.

```
/* EJEMPLO DE BUFFER COMPARTIDO CON DOS VARIABLES CONDICIONALES */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_BUFFER        1024    /* tamaño del buffer */
#define DATOS_A_PRODUCIR  10000 /* datos a producir y consumir */

void Productor (void);
void Consumidor (void);

/* mutex para controlar el acceso al buffer compartido */
pthread_mutex_t mutex;
pthread_cond_t lleno;   /* condicion de lleno */
pthread_cond_t vacio;   /* condicion de vacio */
/* son globales y compartidas por los hilos */
int n_elementos=0;      /* número de elementos en el buffer */
int buffer[MAX_BUFFER]; /* buffer circular */

int main(int argc, char *argv[])
{
     pthread_t th1, th2;
     pthread_mutex_init(&mutex, NULL);
     pthread_cond_init(&lleno, NULL);
     pthread_cond_init(&vacio, NULL);

     pthread_create(&th1, NULL, (void *)Productor, NULL);
     pthread_create(&th2, NULL, (void *)Consumidor, NULL);
     pthread_join(th1, NULL);
     pthread_join(th2, NULL);

     pthread_mutex_destroy(&mutex);
     pthread_cond_destroy(&lleno);
     pthread_cond_destroy(&vacio);
     pthread_exit(0);
}

/* codigo del productor */
void Productor(void)
{
     int dato, i, pos = 0;
     for(i=0; i<DATOS_A_PRODUCIR; i++ )
     {
        dato = i;                          /* producir dato */
        pthread_mutex_lock(&mutex);        /* acceder al buffer */
        while (n_elementos == MAX_BUFFER) /* si buffer lleno */
            pthread_cond_wait(&lleno, &mutex);    /* se bloquea */
        buffer[pos] = dato;
        pos = (pos + 1) % MAX_BUFFER;
        n_elementos = n_elementos + 1;
        if (n_elementos == 1)
            pthread_cond_signal(&vacio);    /* buffer no vacío */
```

```
            pthread_mutex_unlock(&mutex);
            printf("Produce %d \n", dato);
        }
        pthread_exit(0);
}

/* codigo del consumidor */
void Consumidor(void)
{
        int dato, i ,pos = 0;
        for(i=0; i<DATOS_A_PRODUCIR; i++ )
        {
            pthread_mutex_lock(&mutex);     /* acceder al buffer */
            while (n_elementos == 0)        /* si buffer vacío */
                 pthread_cond_wait(&vacio, &mutex); /* se bloquea */
            dato = buffer[pos];
            pos = (pos + 1) % MAX_BUFFER;
            n_elementos = n_elementos - 1 ;
            if (n_elementos == MAX_BUFFER - 1);
                 pthread_cond_signal(&lleno);    /* buffer no lleno */
            pthread_mutex_unlock(&mutex);
            printf("Consume %d \n", dato);
        }
        pthread_exit(0);
}
```

3. Bájate el programa `var_cond.c` y analiza el funcionamiento del productor/consumidor. Entiende para qué se utilizan las dos variables condicionales y cuál es su función. Entiende cómo está involucrado un *mutex* para acceder a la sección crítica de forma segura y cómo está asociado a las variables condicionales.

4. Realiza una función implementada con *mutex* y variables condicionales que haga de barrera para hilos, de tal manera que un conjunto de hilos la llaman para esperarse a que sus compañeros también lo hagan. Cuando llegan todos, siguen su ejecución. Está función es tan común que hay un nuevo mecanismo de sincronización, `barrier`, implementado en POSIX.

Problema final.

5. Realiza un programa para calcular el producto escalar de dos vectores de forma paralela. El tamaño de los vectores será múltiplo del número de hilos. La estructura de datos que manejan de forma compartida en una `struct` será: dos punteros a flotante representado los dos vectores, una longitud entera para esos vectores y una suma flotante que será el resultado de la operación. El hilo padre la inicializará. Los hilos hijo esclavos obtendrán su identificación y harán el cálculo de su parte de vectores. Piensa si necesitas algún tipo de sincronización entre los hilos. El hilo padre pondrá el resultado final en pantalla.

Opcional. Sincronización de threads: señales

La ejecución de múltiples flujos de control, puede ocurrir de muy distintas maneras. Si nos interesa una de ellas, la librería de p*threads* nos proporcionará una serie de llamadas de sincronización, unas relacionadas con las señales vista en el guion 2 y otras propias de p*threads* pero comunes con la teoría general de comunicación de procesos, que son principalmente las variables condicionales y los *mutex*.

En cuanto a las primeras, las señales también pueden ser utilizadas por los p*threads* y de hecho las que estén relacionadas con ellos hacen referencia a TID y no a PID. Por ejemplo, para enviar señales a un p*thread* se utiliza la llamada `pthread_kill` que toma como argumento el identificador de p*thread* y la señal a enviar:

```
int pthread_kill(pthread_t thread, int sig);
```

Otras consideraciones que hay que tener en cuenta son:

- Cuando se crea un *thread*
  La máscara es heredada del padre (cada hilo tiene su propia máscara).
  No hay señales pendientes para el nuevo *thread*.

- Cada hilo puede cambiar su máscara de forma parecida al proceso:

  ```
  int pthread_sigmask(int how, const sigset_t *restrict set,
  sigset_t *restrict oset);
  ```

- Hay que tener cuidado con el uso de llamadas para manejar señales:
  No está indicado si tenemos procesos con varios hilos (todos los hilos comparten el manejador). Si llega la señal al proceso ¿Qué hilo la toma?

  a. Si es una señal de error le llega al hilo que la provocó.
  b. Si es enviada con la `pthread_kill` le llega al hilo destino.
  c. Pero si es otra señal le puede llegar a cualquiera que la tenga desbloqueada. Una solución es: Hacer que el hilo principal bloquee todas las señales y esta máscara sea heredada por los hilos hijo. Después el hilo designado manejará las señales él sólo, quedándose bloqueado esperándola (atómica), por ejemplo con la llamada sigwait:

  ```
  int sigwait (const sigset_t *restrict set, int *restrict sig);
  ```

  Que toma dos argumentos:

  1. Las señales que queremos recibir en forma de conjunto (por referencia).
  2. La señal pendiente recibida por referencia.