

Capítulo III:

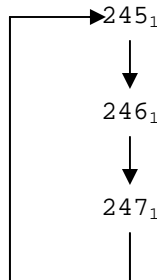
Programación con Posix Threads

III.1. QUE SON LOS THREADS?

Un thread es una secuencia de instrucciones ejecutada dentro de un programa. En otras palabras, cuando se ejecuta un programa, el CPU utiliza el contador de programa del proceso para determinar que instrucción debe ejecutar a continuación. El flujo de instrucciones resultante se denomina *hilo de ejecución del programa* y es el flujo de control para el proceso representado por la secuencia de direcciones de instrucciones indicadas por el contador de programa durante la ejecución del código de éste.

Desde el punto de vista del programa, la secuencia de instrucciones de un hilo de ejecución es un flujo in-interrumpido de direcciones (ver ejemplo 1). En cambio, desde el punto de vista del procesador, los hilos de ejecución de diferentes procesos están entremezclados y el punto en que la ejecución cambia de un proceso a otro se denomina *cambio o conmutación de contexto* (ver ejemplo 2).

Ejemplo 1. Se tiene el PROCESO_1 que ejecuta las instrucciones 245, 246 y 247 en un ciclo. Su thread de ejecución se puede representar como:



Los subíndices identifican el hilo de ejecución. En este caso sólo hay uno.

Ejemplo 2. Se tiene el PROCESO_1 igual que en el ejemplo pasado y el PROCESO_2 que ejecuta sus instrucciones 10, 11, 12, ... El CPU ejecuta instrucciones en el orden:

245₁ → 246₁ → 247₁ → 245₁ → 246₁ → 10₂ → 11₂ → 12₂ → 13₂ →
 → 247₁ → 245₁ → 246₁ → 247₁ . . .

Hay conmutación o cambio de contexto entre 246₁ y 10₂ y entre 13₂ y 247₁.

Para el procesador los threads de ejecución están intercalados pero para los procesos individuales, son secuencias continuas.

Una extensión natural del modelo de proceso es permitir la ejecución de varios threads dentro del mismo proceso. Esta extensión proporciona un mecanismo eficiente para controlar los threads de ejecución que comparten tanto código como datos, con lo que se evita cambios de contexto. Esta estrategia también mejora el rendimiento porque las máquinas con varios procesadores pueden ejecutar múltiples threads simultáneamente.

Cada hilo de ejecución se asocia con un "Thread", un TDA que representa el flujo de control dentro de un proceso. Cada hilo tiene su propia pila de ejecución, valor de contador de programa, conjunto de registros y estado.

Al declarar muchos hilos dentro de los confines de un solo proceso, el programador puede lograr paralelismo a bajo costo, no obstante los hilos también pueden ofrecer ciertas complicaciones en cuanto a la necesidad de sincronizarlos. Los hilos o threads son llamados *procesos ligeros* y son primos de los procesos UNIX, los cuales se forman de un solo Thread de ejecución simple que inicia en el *main()*. Cada línea de código es ejecutada en turno, una línea cada vez. En UNIX, cualquier proceso tiene:

- a) Un espacio de direcciones (pila, datos y segmentos de código)
- b) Una tabla de descriptores de archivos (archivos abiertos)
- c) Un hilo o programa de ejecución

III.1.1. Características de los Threads

- Los hilos son más pequeños comparados con los procesos
- La creación de un hilo es menos costosa que la de un proceso
- Los hilos comparten los recursos mientras que los procesos requieren su propio conjunto de recursos
- Los hilos ocupan menos memoria que los procesos (son más económicos)
- Los hilos proporcionan a los programadores la posibilidad de escribir aplicaciones concurrentes que se pueden ejecutar tanto en sistemas monoprocesador como multiprocesador de forma transparente.
- Los hilos pueden incrementar el rendimiento en entornos monoprocesador

III.2. INTRODUCCION A LA PROGRAMACION CON THREADS

Una librería o paquete de threads permite escribir programas con varios puntos simultáneos de ejecución, sincronizados a través de memoria compartida. Sin embargo la programación con hilos introduce nuevas dificultades. La programación concurrente tiene técnicas y problemas que no ocurren en la programación secuencial. Algunos problemas son sencillos (por ejemplo el ínter bloqueo) pero algunos otros aparecen como penalizaciones al rendimiento de la aplicación.

Un thread es un concepto sencillo: *un simple flujo de control secuencial*. Con un único thread existe en cualquier instante un único punto de ejecución. El programador no necesita aprender nada nuevo para programar un único thread.

Sin embargo, cuando se tienen múltiples hilos en un programa significa que en cualquier instante el programa tiene múltiples puntos de ejecución, uno en cada uno de sus threads. El programador decide cuando y donde crear múltiples threads, ayudándose de una librería o paquete en tiempo de ejecución.

En un lenguaje de alto nivel, las variables globales son compartidas por todos los threads del programa, esto es, los hilos leen y escriben en las mismas posiciones de memoria. El programador es el responsable de emplear los mecanismos de sincronización de la librería de hilos proporcionada por el lenguaje para garantizar que la memoria compartida se acceda de forma correcta por los hilos. Las facilidades proporcionadas por la librería de hilos que se utilice son conocidas como *primitivas ligeras*, lo que significa que las primitivas de:

- Creación
- Mantenimiento

- Sincronización y
- Destrucción

Son suficientemente económicas en esfuerzo para las necesidades del programador.

III.3. EMPLEO DE THREADS EN LA CONCURRENCIA

- *Uso de un sistema multiprocesador:* Los hilos son una herramienta atractiva para permitir a un programa aprovecharse de este tipo de Hardware. Empleando una librería de threads, el programador puede utilizar los procesadores de forma económica. Esto parece funcionar bien en sistemas que van de 10 a 1000 procesadores.
- *Gestión de dispositivos de entrada/salida:* Discos, redes, terminales e impresoras pueden ser fácilmente programables mediante hilos, de tal forma que las peticiones a un dispositivo sean secuenciales y el hilo que realiza la petición se suspenda hasta que la solicitud sea completada; mientras el programa principal realiza otro trabajo con otros hilos solapando la ejecución de varios threads.
- *Los usuarios como fuente de concurrencia:* A veces un usuario necesita realizar dos o más tareas de forma simultánea. Los threads son una buena forma de cubrir esta necesidad.
- *Constitución de un sistema distribuido:* Servidores de red compartidos, donde el servidor se encarga de recibir peticiones de múltiples clientes. El uso de múltiples hilos permite al servidor gestionar las solicitudes de los clientes en paralelo, en vez de procesarlas en serie o crear un proceso de servicio para cada cliente, lo cual supone un enorme gasto.
- *Reducción de la latencia de las operaciones de un programa:* La latencia es el tiempo empleado entre la llamada a un procedimiento y la finalización de éste. Emplear hilos para diferir el trabajo es una técnica potente. La reducción de la latencia puede mejorar los tiempos de respuesta de un programa.

Uno de los problemas que se tenían al utilizar múltiples threads de ejecución es que hasta hace poco no existía un estándar para ello. La extensión POSIX.1c se aprobó en Junio de 1995. Con la adopción de un estándar POSIX para los hilos, se están haciendo más comunes las aplicaciones con Threads.

III.4. EL ESTANDAR POSIX THREADS

El estándar POSIX Threads significa técnicamente el API Thread especificado por el estándar formal internacional POSIX 1003.1c-1995. POSIX significa: **P**ortable **O**perating **S**ystem **I**nterface. Todas las fuentes que empleen POSIX 1003.1c, POSIX.1c o simplemente **Pthreads**, deben incluir el archivo de encabezado *pthread.h* con la directiva:

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

Ya que pthread es una librería POSIX, se podrán portar los programas hechos con ella a cualquier sistema operativo POSIX que soporte threads. Por tanto, para crear programas que

hagan uso de la librería *pthread.h* necesitamos en primer lugar la librería en sí. Esta viene en la mayoría de las distribuciones de LINUX y si no es así, se puede bajar de la red.

Una vez que tenemos la librería instalada, deberemos compilar el programa y ligarlo con dicha librería en base al compilador que se utilice. La librería de hilos POSIX.1c debe ser la última librería especificada en la línea de comandos del compilador:

```
CC . . . -lpthread
```

Por ejemplo, la forma más usual de hacer lo anterior, si estamos usando un compilador GNU como *gcc*, es con el comando:

```
gcc prog_con_hilos.c -o prog_con_hilos_ejecutable -lpthread
```

En POSIX.1c todos los hilos de un proceso comparten las siguientes características:

- El espacio de direcciones
- El ID del proceso
- EL ID del proceso padre
- El ID del proceso líder del grupo
- Los identificadores de usuario
- Los identificadores de grupo
- El directorio de trabajo raíz y actual
- La máscara de creación de archivos
- La tabla de descriptores de archivos
- El timer del proceso

Por otro lado, cada hilo tiene la siguiente información específica:

- Un identificador de hilo único
- La política de planificación y la prioridad
- Una variable errno por hilo
- Datos específicos por hilo
- Gestores de cancelación por hilo
- Máscara de señales por hilo

Las operaciones llevadas a cabo sobre un hilo son:

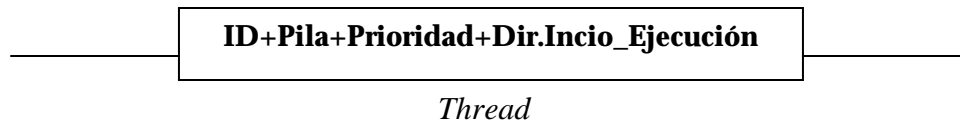
- Creación y destrucción
- Sincronización entre hilos
- Posibilidad de disponer para cada thread memoria local propia
- Gestión de prioridades entre hilos
- Gestión de señales

Como ya se ha mencionado, un proceso puede tener sus propios hilos, es decir, los threads viven dentro de un proceso pesado, por tanto, si un thread cambia alguna de las entidades de un proceso, el cambio será visto por todos los threads. La función `main()` esta asociada con el hilo principal e inicial del proceso, en otras palabras, la función `main()` de cualquier

programa que compilemos es un hilo de ejecución. Cualquier otro hilo de ejecución que queramos se ha de crear explícitamente.

III.4.1. Gestión Básica de Threads

Un thread tiene un identificador (ID), una pila, una prioridad de ejecución y una dirección de inicio de la ejecución:



Hacemos referencia a los hilos POSIX mediante un ID de tipo:

`pthread_t`

Un hilo puede averiguar su ID llamando a:

`pthread_self`

III.4.2. Espacio de Nombres

A continuación se define la estructura de los tipos de datos del estandar POSIX.1c. Cada tipo de dato es de la forma: *pthread[_object]_t*

Existen 8 tipos de datos en el POSIX.1c:

TIPO DE DATO	DESCRIPCION
<i>pthread_attr_t</i>	Atributo de hilo
<i>pthread_mutexattr_t</i>	Atributo de mutex
<i>pthread_condattr_t</i>	Atributo de variable de condición
<i>pthread_mutex_t</i>	Mutex (bloqueo con exclusión mutua)
<i>pthread_cond_t</i>	Variable de condición
<i>pthread_t</i>	Hilo (Identificador de hilo o ID)
<i>pthread_once_t</i>	Ejecución una sola vez
<i>pthread_key_t</i>	Clave sobre datos específicos de hilo

En cuanto a las funciones estándar del POSIX.1c, éstas tienen la forma:

pthread[_object]_operation[_np | _NP]

donde:

_object: es un tipo (no requerido si es thread -hilo-)

_operation: es un tipo específico de operación

_np | _NP: Es usado para identificar funciones específicas implementadas no portables

Las funciones de threads más comunes en POSIX son:

<i>CLASIFICACION</i>	<i>FUNCION</i>	<i>DESCRIPCION</i>
GESTION DE HILOS (Existen una serie de funciones básicas en la gestión de hilos que realizan una aplicación multihilo)	pthread_create	Crea un nuevo hilo de ejecución
	pthread_equal	Compara si dos identificadores de hilo son el mismo hilo
	pthread_exit	Finaliza el hilo que realiza la llamada
	pthread_join	Sincroniza el hilo actual con la finalización del hilo específico, devolviendo el estado del hilo por el que se espera
	pthread_self	Devuelve el identificador del hilo que realiza la llamada
	pthread_detach	Convierte el hilo especificado en independiente
	pthread_getschedparam	Obtiene la política de planificación y los parámetros del hilo especificado
	pthread_setschedparam	Establece la política de planificación y los parámetros del hilo especificado
	pthread_kill	Envía la señal especificada al hilo especificado

<i>CLASIFICACION</i>	<i>FUNCION</i>	<i>DESCRIPCION</i>
EXCLUSION MUTUA (Un mutex es empleado para proteger el acceso compartido a los recursos)	pthread_mutex_init	Inicializa un mutex con los atributos especificados
	pthread_mutex_destroy	Destruye el mutex especificado
	pthread_mutex_lock	Adquiere el mutex indicado (bloquea el mutex)
	pthread_mutex_trylock	Intenta adquirir el mutex indicado
	pthread_mutex_unlock	Libera el mutex previamente adquirido (bloqueado) que se especifica
	pthread_mutex_getprioceiling	Obtiene el valor de prioridad límite (prioceiling) del mutex especificado

<i>CLASIFICACION</i>	<i>FUNCION</i>	<i>DESCRIPCION</i>
DATOS ESPECIFICOS DE HILO (Si un hilo desea crear datos que no sean globales a todos los hilos pero si a sus procedimientos, puede crear datos especiales de este tipo)	pthread_key_create	Crea una clave para datos específicos de hilo, que puede ser empleada por todos los hilos del proceso
	pthread_key_delete	Destruye una clave de datos específicos del hilo
	pthread_setspecific	Establece el valor para la clave dada en el hilo que realiza la llamada
	pthread_getspecific	Devuelve el último valor para la clave dada del hilo que realiza la llamada

<i>CLASIFICACION</i>	<i>FUNCION</i>	<i>DESCRIPCION</i>
CANCELACION (La cancelación de es gestionada como una petición de envío de una señal interna especial SIGCANCEL a un hilo. La acción a tomar depende del estado de cancelación del hilo receptor)	pthread_cleanup_push	Introduce (push) una función en la cima de la pila de cancelación, que será ejecutada cuando se produzca la cancelación del hilo
	pthread_cleanup_pop	Extrae (pop) una función de la cima de la pila de cancelación y opcionalmente la ejecuta
	pthread_setcancelstate	Establece el estado de cancelación y devuelve el estado anterior para el hilo que realiza la llamada
	pthread_setcanceltype	Establece el tipo de cancelación y devuelve el tipo anterior para el hilo que realiza la llamada
	pthread_cancel	Cancela el hilo especificado
	pthread_testcancel	Introduce un punto de cancelación que es un punto en el que un hilo puede ser finalizado si se le notificó una señal de cancelación

CLASIFICACION	FUNCION	DESCRIPCION
ATRIBUTOS DE HILO (Los atributos permiten crear patrones de hilo para crear hilos con las mismas características)	<code>pthread_attr_init</code>	Inicializa un objeto de atributos de hilo
	<code>pthread_attr_destroy</code>	Destruye un objeto de atributos de hilo
ATRIBUTOS DE MUTEX (Los atributos permiten crear patrones de mutex para crear mutex con las mismas características)	<code>pthread_mutexattr_init</code>	Inicializa un objeto de atributos de mutex
	<code>pthread_mutexattr_destroy</code>	Destruye un objeto de atributos de mutex
ATRIBUTOS DE VARIABLE DE CONDICION (Los atributos permiten crear patrones de variables de condición para crear variables de condición con las mismas características)	<code>pthread_condattr_init</code>	Inicializa un objeto de atributos de variable de condición
	<code>pthread_condattr_destroy</code>	Destruye un objeto de atributos de variable de condición

CLASIFICACION	FUNCION	DESCRIPCION
GESTION DE SEÑALES (Cada hilo tiene su propia máscara de señales. Los gestores de señales son compartidos por todos los hilos del proceso pues se instalan a nivel proceso)	<code>pthread_sigmask</code>	Examina o modifica la máscara de señales del hilo que realiza la llamada
	<code>pthread_kill</code>	Envía la señal especificada al hilo indicado
	<code>sigwait</code>	Acepta de forma síncrona una señal (suspende el hilo hasta la llegada de la señal indicada).

III.4.3. Creación y Destrucción (manipulación) de Threads

Se dice que un hilo es dinámico si se puede crear en cualquier instante durante la ejecución de un proceso y si no es necesario especificar por adelantado el número de hilos. En POSIX los hilos se crean dinámicamente con la función `pthread_create()`, la cual crea un hilo y lo coloca en una cola de hilos preparados.

Para crear un hilo nos valdremos de la función `pthread_create()` de la librería de hilos POSIX y de la estructura de datos `pthread_t` que identifica cada thread diferenciándolo de los demás y que contiene todos sus datos:

Sintaxis:

```
int pthread_create(pthread_t *hilo, const pthread_attr_t *atributo,
void * (*rutina)(void *), void *arg);
```

Donde:

***hilo:** Es una variable de tipo *pthread_t* que contendrá los datos del thread y que nos servirá para identificar el thread en concreto cuando nos interese hacer llamadas a la librería para llevar a cabo una acción sobre él.

***atributo:** Es un parámetro del tipo *pthread_attr_t* que se debe inicializar previamente con los atributos que queramos que tenga el thread. Entre los atributos están la prioridad, el quantum, el algoritmo de planificación a utilizar, etc. Lo normal es pasar como parámetro de este argumento el valor NULL para que la librería le asigne al thread los atributos por defecto.

***rutina:** Es la dirección de la función que se quiere que ejecute el thread. La función debe devolver un apuntador genérico (void *) como resultado y debe tener como único parámetro otro apuntador genérico. La ventaja de tener apuntadores genéricos (void *) es que se puede devolver cualquier cosa que se nos ocurra mediante los castings de tipos necesarios. Si necesitamos pasar o devolver más de un parámetro a la vez, se puede crear una estructura y meter allí dentro todo lo que necesitemos. Luego pasaremos o devolveremos la dirección de ésta estructura como único parámetro.

***arg:** Es un apuntador al parámetro que se le pasará a la función. Puede ser NULL si no se le quiere pasar nada a la función.

La función devuelve un cero si todo ha ido bien, o un valor distinto de cero en caso de que haya ocurrido algún error. Una vez que se ha llamado a la función *pthread_create()*, se tienen entonces los hilos funcionando con dos opciones:

- a) Esperar a que terminen los threads, en el caso de que sea de interés recoger algún resultado, o bien
- b) Simplemente decirle a la librería de pthreads que cuando termine la ejecución de la función del thread en cuestión elimine todos sus datos de sus tablas internas.

Para ello se disponen de dos funciones más: *pthread_join()* y *pthread_detach()*

Sintaxis:

```
int pthread_join(pthread_t *hilo, void **thread_return);
```

Esta función suspende el thread llamante hasta que no termine su ejecución el thread indicado por el parámetro ***hilo**. Además, una vez que éste último termina, pone en ****thread_return** el resultado devuelto por el thread que se estaba ejecutando. Este último parámetro puede ser NULL en cuyo caso le estaremos indicando a la librería que no nos interesa el resultado.

La función devuelve cero en caso de éxito o un valor diferente de cero en caso de producirse un error.

Sintaxis:

```
int pthread_detach(pthread_t *hilo)
```

Esta función indica a la librería que no queremos que nos guarde el resultado de la ejecución del thread indicado por el parámetro ***hilo**. Por defecto la librería guarda el resultado de la ejecución de todos los threads hasta que hacemos un *pthread_join()* para recoger el resultado. Así que si no nos interesa el resultado de los threads, tenemos que indicarlo con esta función. Una vez que el hilo haya terminado, la librería eliminará los datos del thread de sus tablas internas y se tendrá más espacio disponible para crear otros hilos.

La función devuelve cero en caso de éxito o un valor diferente de cero en caso contrario.

Finalmente, para devolver valores cuando un thread finaliza, se utiliza la función *pthread_exit()*:

Sintaxis:
`void pthread_exit(void *thread_return);`

Esta función termina la ejecución del thread que la llama. El valor del parámetro ***thread_return** queda disponible para *pthread_join()* si ésta tuvo éxito. El parámetro de ésta función es un apuntador genérico a los datos que se quieren devolver como resultado. Estos datos serán recogidos más tarde cuando se haga un *pthread_join()* a través del identificador de nuestro hilo. La función no devuelve ningún valor.

PROGRAMA 1: El siguiente programa creará `MAX_THREADS` threads que ejecutarán la función *funcion_thr()*. Esta función sacará un mensaje por pantalla del tipo: *“Hola, soy el thread número y”*, donde *y* será un número diferente para cada hilo.

Para pasar esos parámetros a la función, utilizaremos un *struct de C* donde se introducirá la cadena que debe imprimir cada hilo más su identificador (la cadena la podríamos haber metido directamente dentro de la función, pero así veremos como se pasa más de un parámetro al hilo).

Una vez terminada su ejecución, el thread devolverá como resultado su identificados (codificado en un entero), que será impreso en pantalla por el thread padre que esperará a que todos los threads terminen:

```
/* Creamos MAX_THREAD threads que sacan por pantalla una cadena y su
   identificador. Una vez que terminan su ejecucion devuelven como
   resultado su identificador */

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

#define MAX_THREADS      10

pthread_t tabla_thr[MAX_THREADS]; // tabla con los identificadores de los
                                  // threads

typedef struct              // tipo de datos y tabla con los parametros
{
    int id;
    char *cadena;
} thr_param_t;
```

```

thr_param_t param[MAX_THREADS];

// tenemos que crear una tabla para los parametros porque los pasamos por
// referencia. Asi, si solo tuvieramos una variable para los parametros
// al modificar esta modificariamos todas las que habiamos pasado
// anteriormente porque los threads no se quedan con el valor sino con la
// direccion

// Esta es la funcion que ejecutan los threads

void * funcion_thr(void *p)
{
    thr_param_t *datos;
    datos= (thr_param_t *)p;
    printf("%s %d\n", datos->cadena, datos->id);
    pthread_exit(&(datos->id)); // una vez terminamos, devolvemos el valor
}

int main(void)
{
    int i, *res;
    void * r;

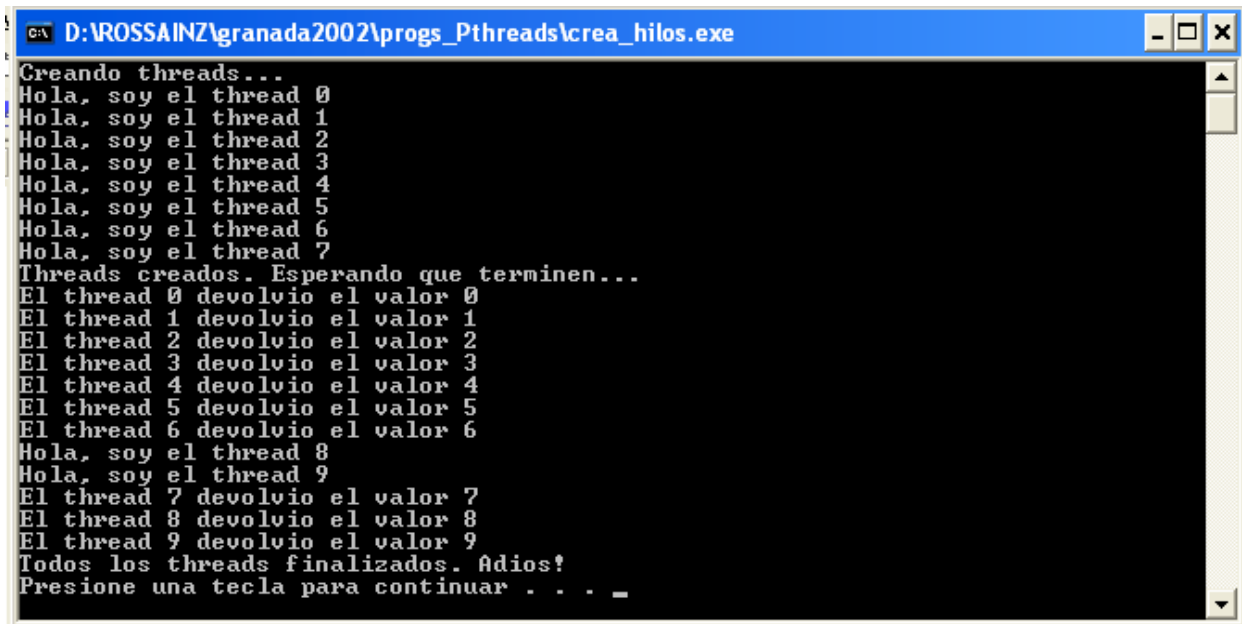
    printf("Creando threads...\n");
    for (i=0; i<MAX_THREADS; i++)
    {
        param[i].cadena="Hola, soy el thread";
        param[i].id= i;
        pthread_create(&tabla_thr[i],NULL,funcion_thr,(void *)&param[i]);
    }

    printf("Threads creados. Esperando que terminen...\n");
    for (i=0; i<MAX_THREADS; i++)
    {
        pthread_join(tabla_thr[i],&r);
        res=(int *)r;
        printf("El thread %d devolvio el valor %d\n", i, *res);
    }

    printf("Todos los threads finalizados. Adios!\n");
    system("PAUSE");
    return 0;
}

```

Una posible salida:

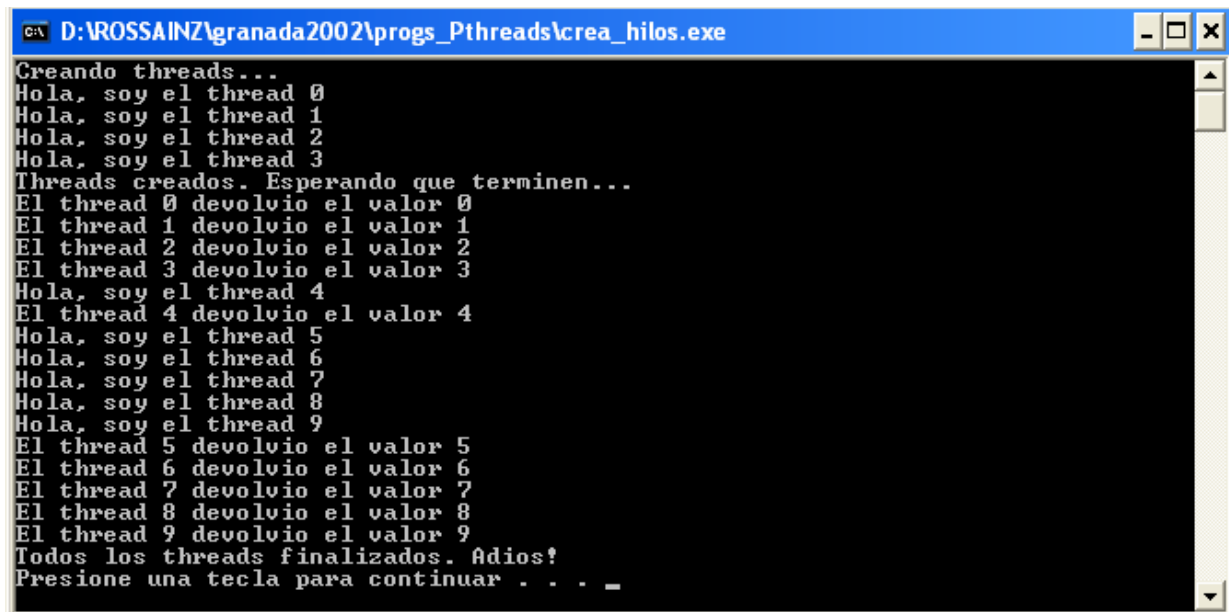


```

D:\ROSSAINZ\granada2002\progs_Pthreads\crea_hilos.exe
Creando threads...
Hola, soy el thread 0
Hola, soy el thread 1
Hola, soy el thread 2
Hola, soy el thread 3
Hola, soy el thread 4
Hola, soy el thread 5
Hola, soy el thread 6
Hola, soy el thread 7
Threads creados. Esperando que terminen...
El thread 0 devolvio el valor 0
El thread 1 devolvio el valor 1
El thread 2 devolvio el valor 2
El thread 3 devolvio el valor 3
El thread 4 devolvio el valor 4
El thread 5 devolvio el valor 5
El thread 6 devolvio el valor 6
Hola, soy el thread 8
Hola, soy el thread 9
El thread 7 devolvio el valor 7
El thread 8 devolvio el valor 8
El thread 9 devolvio el valor 9
Todos los threads finalizados. Adios!
Presione una tecla para continuar . . . _

```

Otra posible salida:



```

D:\ROSSAINZ\granada2002\progs_Pthreads\crea_hilos.exe
Creando threads...
Hola, soy el thread 0
Hola, soy el thread 1
Hola, soy el thread 2
Hola, soy el thread 3
Threads creados. Esperando que terminen...
El thread 0 devolvio el valor 0
El thread 1 devolvio el valor 1
El thread 2 devolvio el valor 2
El thread 3 devolvio el valor 3
Hola, soy el thread 4
El thread 4 devolvio el valor 4
Hola, soy el thread 5
Hola, soy el thread 6
Hola, soy el thread 7
Hola, soy el thread 8
Hola, soy el thread 9
El thread 5 devolvio el valor 5
El thread 6 devolvio el valor 6
El thread 7 devolvio el valor 7
El thread 8 devolvio el valor 8
El thread 9 devolvio el valor 9
Todos los threads finalizados. Adios!
Presione una tecla para continuar . . . _

```

Este primer programa ilustra el esquema básico que siguen todas las aplicaciones que lanzan threads para realizar un cálculo y esperar su resultado:

1. Crear el (los) thread(s)
2. Esperar a que terminen
3. Recoger y procesar el (los) resultado(s)

A esto se le llama PARALELISMO ESTRUCTURADO.

PROGRAMA 2. Realizaremos la aplicación más típica de todo lenguaje de programación, el programa HOLA MUNDO en su versión multi-hilo. La aplicación imprimirá el mensaje “HOLA MUNDO” en la salida estándar (stdout). El código para la versión secuencial (no multi-hilo) de *Hola Mundo* es:

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

void imprimir_mensaje( void *puntero );

int main()
{
    char *mensaje1 = "Hola";
    char *mensaje2 = "Mundo";

    imprimir_mensaje((void *)mensaje1);
    imprimir_mensaje((void *)mensaje2);
    printf("\n");
    system("PAUSE");
    exit(0);
    return 1;
}

void imprimir_mensaje( void *puntero )
{
    char *mensaje;
    mensaje = (char *) puntero;
    printf("%s ", mensaje);
}
```

Para la versión multihilo necesitamos dos variables de hilo y una función de comienzo para los nuevos hilos que será llamada cuando comiencen su ejecución. También necesitamos alguna forma de especificar que cada hilo debe imprimir un mensaje diferente. Una aproximación es dividir el trabajo en cadenas de caracteres distintas y proporcionar a cada hilo una cadena diferente como parámetro. Examinemos el programa 3:

PROGRAMA 3. El programa crea el primer hilo llamando a *pthread_create()* y pasando “*Hola*” como argumento inicial. El segundo hilo es creado con “*Mundo*” como argumento. Cuando el primer hilo inicia la ejecución, comienza en la función *imprimir_mensaje()* con el argumento “*Hola*”. Imprime “*Hola*” y finaliza la función. Un hilo termina su ejecución cuando finaliza su función inicial. Así pues, el primer hilo termina después de imprimir el mensaje “*Hola*”. Cuando el 2do. Hilo se ejecuta, imprime “*Mundo*” y al igual que el anterior hilo, finaliza.

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

void * imprimir_mensaje( void *puntero );

int main()
{
    pthread_t hilo1, hilo2;
    char *mensaje1 = "Hola";
    char *mensaje2 = "Mundo";

    pthread_create(&hilo1, NULL, imprimir_mensaje, (void*) mensaje1);
    pthread_create(&hilo2, NULL, imprimir_mensaje, (void*) mensaje2);
    printf("\n\n");
    system("PAUSE");
    exit(0);
    return 1;
}

void * imprimir_mensaje( void *puntero )
{
    char *mensaje;
    mensaje = (char *) puntero;
    printf("%s ", mensaje);
    return puntero;
}
```

Aunque este programa parece correcto, posee errores que debemos considerar como defectos considerables:

1. El defecto más importante es que los hilos pueden ejecutarse concurrentemente. No existe entonces garantía de que el primer hilo ejecute la función *printf()* antes que el segundo hilo. Por tanto, podríamos ver el mensaje *"Mundo Hola"* en vez de *"Hola Mundo"*.
2. Por otro lado existe una llamada a *exit()* realizada por el hilo padre en la función principal *main()*. Si el hilo padre ejecuta la llamada a *exit()* antes de que alguno de los hilos ejecute *printf()*, la salida no se generará completamente. Esto es porque la función *exit()* finaliza el proceso (libera la tarea) y termina todos los hilos. Cualquier hilo padre o hijo que realice la llamada a *exit()* puede terminar con el resto de los hilos del proceso. Los hilos que deseen finalizar explícitamente, deben utilizar la función *pthread_exit()*.

Así nuestro programa *"Hola Mundo"* tiene 2 condiciones de ejecución (race conditions) o condiciones problemáticas:

- La posibilidad de ejecución de la llamada a *exit()*
- La posibilidad de qué hilo ejecutará la función *printf()* primero.

Una forma de arreglar estas dos condiciones de manera artesanal es la siguiente, mostrada en el programa 4.

PROGRAMA 4. Como queremos que cada hilo finalice antes que el hilo padre, podemos insertar un retraso en el hilo padre para dar tiempo a los hilos hijos a que ejecuten *printf()*. Para asegurar que el primer hilo ejecute *printf()* antes que el segundo hilo, podríamos insertar un retraso antes de crear el segundo hilo. El código resultante es entonces:

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

void * imprimir_mensaje( void *puntero );

int main()
{
    pthread_t hilo1, hilo2;
    char *mensaje1 = "Hola";
    char *mensaje2 = "Mundo";

    pthread_create(&hilo1, NULL, imprimir_mensaje, (void*) mensaje1);
    sleep(100);
    pthread_create(&hilo2, NULL, imprimir_mensaje, (void*) mensaje2);
    sleep(100);
    system("PAUSE");
    exit(0);
    return 1;
}

void * imprimir_mensaje( void *puntero )
{
    char *mensaje;
    mensaje = (char *) puntero;
    printf("%s ", mensaje);
    pthread_exit(0);
}
```

Este código parece funcionar bien, sin embargo no es seguro. Nunca es seguro confiar en los retrasos de tiempo. La condición problemática aquí es exactamente la misma que se tiene con una aplicación distribuida y un recurso compartido. El recurso es la salida estándar y los elementos de la computación distribuida son los tres hilos. El hilo 1 debe usar la salida estándar antes que el hilo 2 y ambos deben realizar sus acciones antes de que el hilo padre realice la llamada a *exit()*.

Además de nuestros intentos por sincronizar los hilos mediante retrasos, hemos cometido otro error. La función *sleep()*, al igual que la función *exit()* es relativa a procesos. Cuando un hilo ejecuta *sleep()* el proceso entero duerme, es decir, todos los hilos duermen cuando el proceso duerme. Nos encontramos por tanto en la misma situación que teníamos sin las llamadas a *sleep()* y encima el programa tarda 200 milisegundos más en su ejecución.

PROGRAMA 5. La versión correcta de “*HOLA MUNDO*” empleando primitivas de sincronización que evitan las condiciones de ejecución no deseadas, se presenta en el siguiente código:

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

```

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

typedef struct param
{
    char *mensaje;
    pthread_t hilo;
} param_t;

void * imprimir_mensaje( void *puntero )
{
    param_t *datos;

    datos = (param_t *) puntero;

    if (datos->hilo != 0)
        pthread_join( datos->hilo, NULL );

    printf("%s ", datos->mensaje);
    pthread_exit( 0 );
}

int main()
{
    pthread_t hilo1, hilo2;

    param_t mensaje1 = {"Hola", 0};
    param_t mensaje2 = {"Mundo", 0};

    pthread_create(&hilo1, NULL, imprimir_mensaje, (void*)&mensaje1);
    mensaje2.hilo = hilo1;

    pthread_create(&hilo2, NULL, imprimir_mensaje, (void*)&mensaje2);
    pthread_join( hilo2, NULL );

    system("PAUSE");
    exit(0);
    return 1;
}

```

En esta versión, el parámetro de la función *imprimir_mensaje()* es del tipo definido por el usuario *param_t* que es una estructura que nos permite pasar el mensaje y el identificador de un hilo.

El hilo padre tras crear el hilo 1, obtiene su identificador y lo almacena como parámetro para el hilo 2. Ambos hilos ejecutan la función inicial *imprimir_mensaje()*, pero la ejecución es distinta para cada hilo.

El primer hilo contendrá un 0 (cero) en el campo *hilo* de la estructura de datos con lo que no ejecutará la función *pthread_join()*; ejecutará *printf()* y la función *pthread_exit()*. El segundo hilo en cambio, contendrá el identificador del primer hilo en el campo *hilo* de la estructura de datos y ejecutará la función *pthread_join()*, que lo que hace es esperar a la finalización del primer hilo. Después ejecuta la impresión con *printf()* y después realiza la finalización con

pthread_exit(). El hilo padre, tras la creación de los 2 hilos se sincroniza con la finalización del hilo 2 mediante *pthread_join()*.

III.4.4. Problemas de Concurrency con Pthreads

Cuando decidimos trabajar con programas concurrentes uno de los mayores problemas que surgen (inherente a la concurrencia) es el acceso a variables y/o estructuras compartidas o globales, por ejemplo:

```
Hilo 1: void * fncion_hilo_1(void *arg)
        {
            int resultado;
            .
            .
            .
            if (i == valor_cualquiera)
                {
                    .
                    .
                    resultado = i * (int) *arg;
                    .
                    .
                }
            pthread_exit(&resultado);
        }
```

```
Hilo 2: void * fncion_hilo_2(void *arg)
        {
            int otro_resultado;
            .
            .
            .
            if (funcion_sobre_arg(*arg) == 0)
                {
                    .
                    .
                    i = *arg;
                    .
                    .
                }
            pthread_exit(&otro_resultado);
        }
```

Este código que tiene a la variable *i* como global puede ser problemático si se ejecuta en paralelo y se dan ciertas condiciones.

Supongamos que el *hilo 1* empieza a ejecutarse antes que el *hilo 2* y que casualmente se produce un cambio de contexto (el S.O. suspende la tarea actual y pasa a ejecutar la siguiente) justo después de la línea que dice `if (i == valor_cualquiera)`.

Supondremos además que la condición se cumple por lo que la entrada en ese `if` se producirá. Pero justo en ese momento se realiza el cambio de contexto por el sistema y pone a ejecutar el *hilo 2*, que se ejecuta el tiempo suficiente como para ejecutar la línea: `i = *arg;` Al poco rato, el *hilo 2* deja de ejecutarse volviendo a la ejecución del *hilo 1*.

¿Qué valor tiene ahora i ? ¿El que el hilo 1 esta suponiendo que tiene o el que le a asignado el hilo 2 ?: *i* ha tomado el valor asignado por el *hilo 2*, con lo que el resultado que devolverá el *hilo 1* después de sus cálculos será inválido o inesperado.

Estos son los errores más difíciles de detectar, ya que puede que a veces vaya todo a la perfección cuando se ejecuta el código, pero puede ser que en otras ocasiones dalga todo mal. A esto se le conoce como: **Race Conditions (Condiciones de carrera)** porque según como vaya la cosa puede funcionar o no.

III.4.5. Mecanismos de Pthreads para prevenir los problemas de concurrencia

La librería de Pthreads ofrece mecanismos básicos para prevenir los problemas descritos anteriormente. Estos mecanismos son llamados **SEMAFOROS BINARIOS** y se utilizan para implementar las llamadas **REGIONES CRITICAS (RC)** o **ZONAS DE EXCLUSION MUTUA (ZE)**.

Región Crítica (RC): Es una parte de nuestro código susceptible de verse afectada por cosas como la del ejemplo anterior.

Regla General: Siempre que haya variables o estructuras globales que vayan a ser accedidas por más de un thread a la vez, el acceso a éstas deberá ser considerado una región crítica y proteogido con los medios que proporciona la librería de los Pthreads (que se explican más adelante).

Pthreads ofrece los semáforos binarios llamados también *Semáforos Mutex* o simplemente *mutex*.

Semáforo Binario: Es una estructura de datos que actúa como un semáforo porque puede tener dos estados: abierto o cerrado. Cuando el semáforo esta abierto al primer thread que pide un bloqueo se le asigna ese bloqueo y no deja pasar a nadie más por el semáforo. Mientras que si el semáforo esta cerrado, porque algún thread tenga ya el bloqueo, el thread que lo pidió parará su ejecución hasta que no sea liberado el susodicho bloqueo.

Sólo puede haber un solo thread poseyendo el bloqueo del semáforo, mientras que puede haber más de un thread esperando para entrar en la región crítica, enconlados en la cola de espera del semáforo. Los threads por tanto se excluyen mutuamente (mutex) el uno al otro para entrar.

Forma de implementar las RC: Se pide el bloqueo del semáforo antes de entrar, éste es otorgado al primero que llega mientras que los demás se quedan bloqueado, esperando a que el que entró primero libere el bloqueo o exclusión. Una vez que el que entró sale de la región crítica, éste debe notificarlo a la librería de los Pthreads para que mire si hay algún

otro thread esperando a entrar en la cola. Si es así, le da el bloqueo al primero y deja que siga ejecutándose.

III.4.5.1. Sincronización de Hilos

Como ya sabemos, los hilos se crean dentro del espacio de direcciones de un proceso y comparten recursos como las variables estáticas y los descriptores de archivos. Cuando los hilos utilizan recursos compartidos como estos; deben sincronizar su interacción a fin de obtener resultados consistentes. Hay dos tipos distintos de sincronización:

- **Uso de candados (locking):** Se refiere por lo regular a la tenencia a corto plazo de recursos.
- **La espera:** Se refiere al bloqueo hasta que ocurre cierto suceso. Puede tener duración ilimitada.

La librería Pthreads proporciona *mutex* y *variables de condición* para apoyar estos dos tipos de sincronización en programas multihilos. Además permite el empleo de semáforos en este tipo de programas.

III.4.5.2. Exclusión Mutua o MUTEX

El *mutex* o *candado mutex* (mutex lock) es el mecanismo de sincronización de hilos más sencillo y eficiente. Los programas que emplean mutex preservan regiones críticas y obtienen acceso exclusivo a los recursos. En otras palabras, con mutex se trata de indicar que una región particular de código (RC) sólo puede ser ejecutada por un determinado hilo al mismo tiempo. Primeramente un programa debe declarar una variable de tipo:

```
pthread_mutex_t
```

e inicializarla antes de utilizarla para su sincronización. Por lo general, las variables de mutex son variables estáticas accesibles para todos los hilos del proceso. Aunque un programa puede inicializar un mutex empleando el inicializador estático:

```
PTHREAD_MUTEX_INITIALIZER
```

es más común invocar a la función de inicialización:

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

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

***mutex:** es un apuntador a un parámetro del tipo *pthread_mutex_t* que es el tipo de datos que utiliza la librería Pthreads para controlar los mutex.

***attr:** Es un apuntador a una estructura del tipo *pthread_mutexattr_t* y sirve para definir que tipo de mutex queremos: normal, recursivo o errorcheck. Si este valor es NULL (recomendado), la librería le asignará un valor por defecto.

La función devuelve 0 (cero) si se pudo crear el mutex o -1 si hubo algún error.

Ejemplo 1. El siguiente segmento de código inicializa el mutex *my_lock* con los atributos por defecto. La variable *my_lock* debe ser accesible para todos los hilos que la usan:

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>

pthread_mutex_t my_lock;

if (! Pthread_mutex_init(&my_lock, NULL))
    perror("No puede inicializarse my_lock");
```

El método del inicializador estático tiene 2 ventajas respecto de *pthread_mutex_init()*, al inicializar candados mutex: suele ser más eficiente y se garantiza que se ejecutará una y sólo una vez antes de que se inicie la ejecución de cualquier hilo.

Ejemplo 2. El segmento fragmento de código inicializa el mutex *my_lock* con los atributos por omisión empleando el inicializador estático:

```
#include <pthread.h>

pthread_mutex_t my_lock= PTHREAD_MUTEX_INITIALIZER;
```

Un mutex tiene dos estados: *bloqueado* y *desbloqueado*. Inicialmente esta desbloqueado. La función *pthread_mutex_lock()* bloquea el mutex y continúa con la ejecución:

SINTAXIS:

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

La función pide el bloqueo para entrar en una región crítica (RC). Si se quiere implementar una RC todos los hilos tendrán que pedir el bloqueo sobre el mismo candado. Esta función devuelve 0 (cero) si no hubo error, o un valor diferente de cero si lo hubo.

Al finalizar el bloqueo, se debe desbloquear el mutex mediante la función *pthread_mutex_unlock()*.

SINTAXIS:

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

Ésta es una función contraria a la anterior y libera el bloqueo que se tuviera sobre un candado o mutex. La función retorna 0 (cero) como resultado si no hubo error, o un valor diferente de cero si lo hubo.

SINTAXIS:

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

Esta función le dice a la librería que el mutex que le estamos indicando no lo vamos a usar más y que podemos liberar toda la memoria ocupada por ese mutex en sus estructuras internas. La función devuelve 0 (cero) si no hubo error o algo diferente de cero si lo hubo.

En general, un hilo puede utilizar un *pthread_mutex_lock()* (candado mutex) para proteger sus regiones críticas (RC). A fin de preservar la semántica de la región crítica, el mutex debe ser liberado por el mismo hilo que lo adquirió, es decir, no debemos hacer que un hilo adquiera un candado y otro lo libere. Una vez que un hilo adquiera un mutex deberá conservarlo durante un tiempo corto, en otras palabras, cuanto más pequeñas son las regiones críticas más concurrentes son los programas ya que tendrán que esperar menos tiempo en el caso de que hayan bloqueos.

Los hilos que estén esperando sucesos de duración impredecible deberán utilizar otros mecanismos de sincronización como los semáforos y las variables de condición.

Ejemplo 3. El siguiente código utiliza un mutex para proteger una región crítica (RC):

```
#include <pthread.h>

pthread_mutex_t my_lock= PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&my_lock);
    /*
        REGION CRITICA
    */
pthread_mutex_unlock(&my_lock);
```

Ejemplo 4. Si reescribimos el pseudocódigo del ejemplo anterior al ejemplo 1 con lo que hemos visto hasta ahora, la modificación sería la siguiente:

Variables globales: *pthread_mutex_t mutex_acceso;*
int i;

```
Hilo 1: void * fncion_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: void * fncion_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);
        .
        .
        .
    }

```

Como puede observarse lo único que hay que hacer es: inicializar el candado o mutex, pedir el bloqueo antes de la RC y liberarlo después de salir de la RC.

Ejemplo 5. muestra un fragmento de código donde el mutex *m* esta asociado con la variable global *cabeza*. Las funciones *pthread_mutex_lock()* y *pthread_mutex_unlock()* proporcionan exclusión mutua, lo que permite añadir sin problemas de interferencias, la variable local *nuevo_elemento* a la lista enlazada cuya cabeza es “*cabeza*”.

```

typedef struct lista
    {
        char car;
        lista_t *next;
    } lista_t;

```



```

pthread_mutex_t m;
lista_t *cabeza;

void insertar_elemento(lista_t *nuevoElemento)
{
    pthread_mutex_lock(&m)
        nuevoElemento → next = cabeza;
        cabeza= nuevoElemento;
    pthread_mutex_unlock(&m);
}

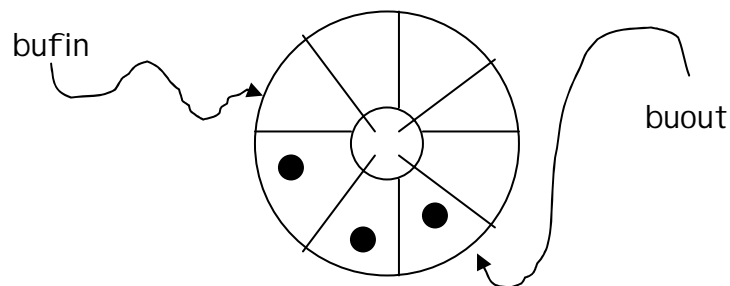
```

Un mutex normalmente esta relacionado con una variable global o estructura de datos global, como ocurre en el ejemplo anterior. En este caso como mucho un hilo se encuentra ejecutando el código de la RC en un instante de tiempo dado.

IMPORTANTE: La adquisición y liberación de candados son voluntarias en el sentido de que sólo se logra la exclusión mutua cuando todos los hilos adquieren correctamente el mutex antes de regresar de sus RC.

No hay nada que impida a un hilo no cooperativo ingresar en su RC sin adquirir un candado. Una forma de asegurar el acceso exclusivo a objetos es permitir el acceso sólo a través de funciones bien definidas e incluir las llamadas de adquisición de candados en esas funciones. Así el mecanismo de candados será transparente para los hilos invocadores.

PROGRAMA 6. La figura siguiente muestra una implementación de una cola como un buffer circular con 8 ranuras o slots y 3 elementos de datos:



El valor *buout* indica el número de ranura del siguiente elemento de datos que se va a sacar y el valor *bufin* indica la siguiente ranura que se llenará. Si los hilos productores y consumidores no actualizan *buout* y *bufin* de una forma mutuamente exclusiva, un productor podría sobrescribir un elemento que no ha sido sacado o un consumidor podría sacar un elemento que ya se usó.

El código siguiente muestra un programa que implementa un buffer circular como un objeto compartido. El código esta en un archivo aparte para que el programa sólo pueda acceder a buffer a través de *get_item* (obtener elemento) y *put_item* (insertar elemento).

```

#include <pthread.h>
#define BUFSIZE 8

static int buffer[BUFSIZE];
static int bufin=0;

```

```

static int bufout=0;

static pthread_mutex_t buffer_lock= PTHREAD_MUTEX_INITIALIZER;

/* Obtener el siguiente elemento del buffer y colocarlo en *itemp */
void get_item(int *itemp)
{
    pthread_mutex_lock(&buffer_lock);
    *itemp=buffer[bufout];
    bufout=(bufout+1) % BUFSIZE;
    // printf("Tamano de bufout: %d\n",bufin);
    pthread_mutex_unlock(&buffer_lock);
    return;
}

/*colocar un elemento en el buffer en la posición bufin y actualizar
bufin */
void put_item(int item)
{
    pthread_mutex_lock(&buffer_lock);
    buffer[bufin]=item;
    bufin=(bufin+1) % BUFSIZE;
    // printf("Tamano de bufin: %d\n",bufin);
    pthread_mutex_unlock(&buffer_lock);
    return;
}

```

El programa siguiente es un ejemplo sencillo en el que un hilo productor escribe los cuadrados de los enteros del 1 al 100 en el buffer circular y un hilo consumidor saca los valores y los suma.

Aunque el buffer está protegido con candados mutex, la sincronización productor-consumidor no es correcta. El consumidor puede sacar elementos de ranuras vacías y el productor puede sobrescribir ranuras llenas:

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "buffer_circ.h"

#define SUMSIZE 100

int sum=0;

void put_item(int);
void get_item(int *);

void producer(void *arg1)
{
    int i;
    for(i=1;i<=SUMSIZE;i++)

```

```

    {
        printf("Se ejecuta PRODUCER: %d\n",i);
        put_item(i*i);
    }
}

void consumer(void *arg2)
{
    int i, myitem;

    for(i=1;i<=SUMSIZE;i++)
    {
        printf("Se ejecuta CONSUMER: %d\n",i);
        get_item(&myitem);
        sum+=myitem;
    }
}

int main(void)
{
    pthread_t prodtid;
    pthread_t constid;
    int i, total;

    total=0;
    for(i=1;i<=SUMSIZE;i++)
        total+=i*i;
    printf("La suma actual debe ser %d\n",total);

    /* CREAR HILOS */

    if(pthread_create(&constid,NULL,(void*)(*)(void*)&consumer,NULL))
        perror("No se pudo crear el hilo consumer...");
    else
        if(pthread_create(&prodtid,NULL,(void*)(*)(void*)&producer,NULL))
            perror("No se pudo crear el hilo producer...");

    /* ESPERAR A QUE LOS HILOS TERMINEN */
    pthread_join(prodtid,NULL);
    pthread_join(constid,NULL);
    printf("Los threads produjeron la suma: %d\n",sum);
    system("PAUSE");
    exit(0);
    return 1;
}

```

Un posible resultado de la ejecución de este programa sería:

```

La suma actual debe ser 338350
Se ejecuta CONSUMER: 1
Se ejecuta CONSUMER: 2
Se ejecuta CONSUMER: 3
Se ejecuta CONSUMER: 4
Se ejecuta CONSUMER: 5
Se ejecuta CONSUMER: 6
Se ejecuta CONSUMER: 7

```

```
Se ejecuta CONSUMER: 8
Se ejecuta CONSUMER: 9
Se ejecuta CONSUMER: 10
Se ejecuta CONSUMER: 11
Se ejecuta CONSUMER: 12
Se ejecuta CONSUMER: 13
Se ejecuta CONSUMER: 14
Se ejecuta CONSUMER: 15
Se ejecuta CONSUMER: 16
Se ejecuta PRODUCER: 1
Se ejecuta PRODUCER: 2
Se ejecuta PRODUCER: 3
Se ejecuta PRODUCER: 4
Se ejecuta PRODUCER: 5
Se ejecuta PRODUCER: 6
Se ejecuta PRODUCER: 7
Se ejecuta PRODUCER: 8
.
.
.
Se ejecuta PRODUCER: 94
Se ejecuta PRODUCER: 95
Se ejecuta PRODUCER: 96
Se ejecuta PRODUCER: 97
Se ejecuta PRODUCER: 98
Se ejecuta PRODUCER: 99
Se ejecuta PRODUCER: 100
Se ejecuta CONSUMER: 85
Se ejecuta CONSUMER: 86
Se ejecuta CONSUMER: 87
Se ejecuta CONSUMER: 88
Se ejecuta CONSUMER: 89
Se ejecuta CONSUMER: 90
Se ejecuta CONSUMER: 91
Se ejecuta CONSUMER: 92
Se ejecuta CONSUMER: 93
Se ejecuta CONSUMER: 94
Se ejecuta CONSUMER: 95
Se ejecuta CONSUMER: 96
Se ejecuta CONSUMER: 97
Se ejecuta CONSUMER: 98
Se ejecuta CONSUMER: 99
Se ejecuta CONSUMER: 100
Los threads produjeron la suma: 315430
Presione una tecla para continuar . . .
```

```
D:\ROSSAINZ\granada2002\progs_Pthreads\mutex\prod_cons_1.exe
La suma actual debe ser 338350
Se ejecuta CONSUMER: 1
Se ejecuta CONSUMER: 2
Se ejecuta CONSUMER: 3
Se ejecuta CONSUMER: 4
Se ejecuta CONSUMER: 5
Se ejecuta CONSUMER: 6
Se ejecuta CONSUMER: 7
Se ejecuta CONSUMER: 8
Se ejecuta CONSUMER: 9
Se ejecuta CONSUMER: 10
Se ejecuta CONSUMER: 11
Se ejecuta CONSUMER: 12
Se ejecuta CONSUMER: 13
Se ejecuta CONSUMER: 14
Se ejecuta CONSUMER: 15
Se ejecuta CONSUMER: 16
Se ejecuta PRODUCER: 1
Se ejecuta PRODUCER: 2
Se ejecuta PRODUCER: 3
Se ejecuta PRODUCER: 4
Se ejecuta PRODUCER: 5
Se ejecuta PRODUCER: 6
Se ejecuta PRODUCER: 7
Se ejecuta PRODUCER: 8
```

PROGRAMA 7. Si se incluye la instrucción *sched_yield* en el programa anterior, el productor y el consumidor operan en alternancia estricta siempre que solo un hilo se ejecute a la vez. Para que el programa funcione a la perfección, tiene que iniciar el hilo productor y el hilo consumidor debe estar en activo en el momento en que el productor ceda su lugar por primera vez.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "buffer_circ.h"

#define SUMSIZE 100

int sum=0;

void put_item(int);
void get_item(int *);

void producer(void *arg1)
{
    int i;

    for(i=1;i<=SUMSIZE;i++)
    {
        printf("Se ejecuta PRODUCER...%d\n",i);
        put_item(i*i);
        // pthread_yield();
        sched_yield(); // Obliga al hilo productor ceder su lugar en
                       // cada iteración
    }
}
```

```

void consumer(void *arg2)
{
    int i, myitem;

    for(i=1;i<=SUMSIZE;i++)
    {
        printf("Se ejecuta CONSUMER...%d\n",i);
        get_item(&myitem);
        sum+=myitem;
        // pthread_yield();
        sched_yield(); // Obliga al hilo consumidor ceder su lugar
                       // en cada iteración
    }
}

int main(void)
{
    pthread_t prodtid;
    pthread_t constid;
    int i, total;

    total=0;
    for(i=1;i<=SUMSIZE;i++)
        total+=i*i;
    printf("La suma actual debe ser %d\n",total);

    /* CREAR HILOS */

    if(pthread_create(&constid,NULL,(void*)(*)(void*)&producer,NULL))
        perror("No se pudo crear el hilo consumer...");
    else
    if(pthread_create(&prodtid,NULL,(void*)(*)(void*)&consumer,NULL))
        perror("No se pudo crear el hilo producer...");

    /* ESPERAR A QUE LOS HILOS TERMINEN */
    pthread_join(prodtid,NULL);
    pthread_join(constid,NULL);
    printf("Los threads produjeron la suma: %d\n",sum);
    system("PAUSE");
    exit(0);
    return 1;
}

```

La instrucción *pthread_yield()* para el POSIX.1c o *sched_yield()* para el POSIX.1b cede su lugar a un hilo en alternancia estricta. La salida del programa es como la que se muestra a continuación:

```
D:\ROSSAINZ\granada2002\progs_Pthreads\mutex\prod_cons_2.exe
La suma actual debe ser 338358
Se ejecuta PRODUCER...1
Se ejecuta CONSUMER...1
Se ejecuta PRODUCER...2
Se ejecuta CONSUMER...2
Se ejecuta PRODUCER...3
Se ejecuta CONSUMER...3
Se ejecuta PRODUCER...4
Se ejecuta CONSUMER...4
Se ejecuta PRODUCER...5
Se ejecuta CONSUMER...5
Se ejecuta PRODUCER...6
Se ejecuta CONSUMER...6
Se ejecuta PRODUCER...7
Se ejecuta CONSUMER...7
Se ejecuta PRODUCER...8
Se ejecuta CONSUMER...8
Se ejecuta PRODUCER...9
Se ejecuta CONSUMER...9
Se ejecuta PRODUCER...10
Se ejecuta CONSUMER...10
Se ejecuta PRODUCER...11
Se ejecuta CONSUMER...11
Se ejecuta PRODUCER...12
Se ejecuta CONSUMER...12
```

```
D:\ROSSAINZ\granada2002\progs_Pthreads\mutex\prod_cons_2.exe
Se ejecuta CONSUMER...89
Se ejecuta PRODUCER...90
Se ejecuta CONSUMER...90
Se ejecuta PRODUCER...91
Se ejecuta CONSUMER...91
Se ejecuta PRODUCER...92
Se ejecuta CONSUMER...92
Se ejecuta PRODUCER...93
Se ejecuta CONSUMER...93
Se ejecuta PRODUCER...94
Se ejecuta CONSUMER...94
Se ejecuta PRODUCER...95
Se ejecuta CONSUMER...95
Se ejecuta PRODUCER...96
Se ejecuta CONSUMER...96
Se ejecuta PRODUCER...97
Se ejecuta CONSUMER...97
Se ejecuta PRODUCER...98
Se ejecuta CONSUMER...98
Se ejecuta PRODUCER...99
Se ejecuta CONSUMER...99
Se ejecuta PRODUCER...100
Se ejecuta CONSUMER...100
Los threads produjeron la suma: 338358
Presione una tecla para continuar . . .
```

La alternancia estricta no resuelve el problema cuando hay números arbitrarios de productores y consumidores. El comportamiento del programa depende del número exacto de hilos que pueden ejecutarse de forma concurrente y del orden en que inician su ejecución.

Un programa multihilo debe funcionar correctamente, sea cual fuere el orden de ejecución y el nivel de paralelismo. Los elementos y ranuras del problema de productores y consumidores deben sincronizarse de modo que el programa sea independiente del orden de ejecución de los hilos. Un método tradicional de sincronización es aquel que emplea *semáforos*.

III.4.5.3. Mas Problemas???

El problema de los DEADLOCKS (o ABRAZOS MORTALES): Los DEADLOCKS se producen cuando un hilo se bloquea esperando un recurso que tiene bloqueado otro hilo que esta esperando un recurso. Si el recurso para el segundo thread no llega nunca, no se desbloqueará con lo cual tampoco se desbloqueará el primer thread. El resultado será nuestro programa bloqueado. Algunas posibles soluciones son:

3. Usar candados recursivos (mutex recursivos)
4. Probar antes de entrar con el uso de `int pthread_mutex_trylock()`.

III.4.5.4. Las variables de Condición

Se puede ver a un mutex como un tipo simple de planificación de recursos. El recurso a planificar es la memoria compartida accedida dentro de las funciones de bloqueo y desbloqueo del mutex y la política de planificación es un único hilo al mismo tiempo.

Sin embargo a veces el programador necesita expresar políticas de planificación más complicadas. Esto requiere el uso de un mecanismo que permite a un hilo suspenderse hasta que suceda algún determinado evento. Este mecanismo de espera por un evento se consigue mediante una *variable de condición*.

Una *variable de condición* esta siempre asociada con un mutex particular y con los datos protegidos por el mutex.

Suponga que un hilo necesita esperar hasta que se cumpla cierto predicado en el que interviene un conjunto de variables compartidas (por ejemplo, que dos de esas variables sean iguales). Estas variables compartidas se protegerán con un candado mutex, pues cualquier segmento de código que las utilice forma parte de una RC. Una variable de condición adicional ofrece un mecanismo para que los hilos esperen el cumplimiento de predicados en los que intervienen esas variables. Cada vez que un hilo modifica una de estas variables compartidas, señala mediante la variable de condición que se ha realizado un cambio. Esta señal activa un hilo en espera, que entonces determina si ahora su predicado se cumple.

Cuando se envía una señal a un hilo en espera, éste debe cerrar el mutex antes de probar su predicado. Si el predicado es *falso*, el hilo deberá liberar el candado y bloquearse de nuevo. El mutex debe liberarse antes de que el hilo se bloquee para que otro hilo pueda tener acceso al mutex y modificar las variables protegidas. La liberación del mutex y el bloqueo deben ser atómicos para evitar que otro hilo modifique las variables entre estas dos operaciones. La señal sólo indica que las variables pueden haber cambiado, no que se ha cumplido el predicado y el hilo bloqueado debe entonces volver a probar el predicado cada vez que reciba una señal.

Hay que seguir los pasos siguientes al utilizar una variable de condición para sincronizar con base en un predicado arbitrario:

1. Adquiera el mutex
2. Pruebe el predicado

-
3. Si el predicado se cumple, realice el trabajo y libere el mutex
 4. Si el predicado no se cumple, llame a *cond_wait* y pase al paso 2 cuando retorne.

Así, una variable de condición se inicializa ya sea empleando un inicializador estático:

```
pthread_cond_t v = PTHREAD_COND_INITIALIZER;
```

o llamando a la función *pthread_cond_init()*:

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t  
*attr);
```

Si **attr* es NULL (recomendado) *pthread_cond_init* utiliza los atributos de variable de condición por omisión.

La función *pthread_cond_wait()* desbloquea el mutex y suspende el hilo (lo encola a la espera de la variable de condición) de forma atómica, en una sola acción.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Un hilo prueba entonces un predicado y llama a *pthread_cond_wait()* si el predicado es falso. Cuando otro hilo modifica variables que podrían hacer que se cumpliera el predicado, activa al hilo bloqueado ejecutando *pthread_cond_signal()*.

```
int pthread_cond_signal(pthread_cond_t *cond);
```

La función *pthread_cond_signal()* no hace nada a menos que exista un hilo bloqueado en la variable de condición, en cuyo caso despierta uno de los hilos bloqueados a la espera.

La función *pthread_cond_broadcast()* es similar a la anterior excepto que despierta a todos los hilos suspendidos a la espera de la variable de condición.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

A fin de garantizar que la prueba del predicado y la espera sean atómicas, el hilo invocador debe obtener un mutex antes de probar el predicado. La implementación garantiza que si el hilo se bloquea en una variable de condición, *pthread_cond_wait()* libera automáticamente el mutex y se bloqueará. Otro hilo no podrá emitir una señal antes de que este hilo se bloquee.

Cuando un hilo es despertado después de ser suspendido en la función *pthread_cond_wait()*, vuelve a intentar adquirir el mutex y después continúa su ejecución. Si el mutex no está disponible, el hilo se suspende a la espera de que el mutex esté disponible. El mutex asociado con una variable de condición protege los datos compartidos. Si un hilo quiere un recurso bloquea el mutex y examina los datos compartidos. Si el recurso está disponible el hilo continúa, sino, el hilo libera el mutex y se suspende llamando a *pthread_cond_wait()*.

Después, cuando algún hilo termina con el recurso, despierta al primer hilo mediante `pthread_cond_signal()` o `pthread_cond_broadcast()`.

El siguiente fragmento de código permite que un hilo se suspenda hasta que una lista enlazada, cuya cabeza es “*cabeza*”, no es vacía y entonces elimina el elemento de la cabeza de la lista.

Ejemplo 6.

```
pthread_mutex_t m;
pthread_cond_t condNoVacio;

Lista_t *cabeza;
Lista_t * extraer_elemento(void)
{
    pthread_mutex_lock(&m);
    while(cabeza==NULL)/* condición de espera */
        pthread_cond_wait(&condNoVacio, &m);
    elemento=cabeza;
    cabeza=cabeza →next;
    pthread_mutex_unlock(&m);
    return elemento;
}
```

y el siguiente fragmento puede ser empleado para un hilo que añade un elemento a la cabeza de la lista:

```
void insertar_elemento(lista_t *nuevoelemento)
{
    pthread_mutex_lock(&m);
    nuevoelemento → next = cabeza;
    cabeza = nuevoelemento;
    pthread_cond_signal(&condNoVacio);
    pthread_mutex_unlock(&m);
}
```

En el ejemplo, dentro de la función `extraer_elemento()` el hilo debe adquirir el mutex *m* especificado en la función `pthread_mutex_lock()` antes de invocar a `pthread_cond_wait()`. Si la condición (`cabeza == NULL`) es verdadera, el hilo ejecutará `pthread_cond_wait()`, liberará el mutex *m* y se bloqueará en la variable de condición `condNoVacio`.

Cuando un hilo ejecuta la función `pthread_cond_wait()` en el ejemplo anterior, esta en posesión del mutex *m*. El hilo se bloquea atómicamente y libera el mutex, permitiendo que otro hilo adquiera el mutex y modifique las variables del predicado.

Cuando un hilo regresa con éxito de una función `pthread_cond_wait()`, ha adquirido el mutex y puede volver a probar el predicado sin volver a adquirir explícitamente el mutex.

La función `pthread_cond_wait()` activa un hilo que esta esperando una variable de condición. Si hay más hilos esperando, se escoge uno siguiendo un criterio congruente con el algoritmo de programación utilizado. La función `pthread_cond_broadcast()` activa todos los hilos que esperan una variable de condición. Estos hilos activados entran en contención por el candado mutex antes de regresar de `pthread_cond_wait()`.

He aquí algunas reglas para el uso de las variables de condición:

- Adquiera el mutex antes de probar el predicado
- Vuelva a probar el predicado después de regresar de un `pthread_cond_wait()` pues el retorno podría deberse a algún suceso no relacionado o bien a un `pthread_cond_signal()` que no implica que el predicado ya se cumpla.
- Adquiera el mutex antes de modificar alguna de las variables que aparecen en el predicado
- Adquiera el mutex antes de llamar a las funciones `pthread_cond_signal()` o `pthread_cond_broadcast()`.
- Retenga el mutex durante un periodo corto, por lo regular sólo mientras prueba el predicado. Libere el mutex lo antes posible, ya sea de forma explícita con `pthread_mutex_unlock()` o de forma implícita con `pthread_cond_wait()`.

PROGRAMA 8. Una solución con variables de condición al Problema del buffer acotado controlado por el productor. El productor termina después de producir una cantidad fija de elementos. El consumidor continúa hasta que procesa todos los elementos y detecta que el productor ya terminó.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "buffer_circ.h"

#define SUMSIZE 100
#define BUFSIZE 8
int sum=0;

pthread_cond_t slots= PTHREAD_COND_INITIALIZER;
pthread_cond_t items= PTHREAD_COND_INITIALIZER;

pthread_mutex_t slot_lock= PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t item_lock= PTHREAD_MUTEX_INITIALIZER;

int nslots=BUFSIZE;
int producer_done=0;
int nitems=0;

void get_item(int *itemp);
void put_item(int item);

void producer(void *arg1)
{
    int i;
    for(i=1;i<=SUMSIZE;i++)
    {
        pthread_mutex_lock(&slot_lock); //Adquirir derecho a una

//ranura...
```

```

        while (nslots<=0)
            pthread_cond_wait(&nslots,&slot_lock);
        nslots--;
    pthread_mutex_unlock(&slot_lock);

    printf("Se ejecuta PRODUCER...%d\n",i);
    put_item(i*i);

    pthread_mutex_lock(&item_lock); //renunciar derecho a un
                                    //elemento...
        nitems++;
        pthread_cond_signal(&nitems);
        pthread_mutex_unlock(&item_lock);
    }
    pthread_mutex_lock(&item_lock);
    producer_done=1;
    pthread_cond_broadcast(&nitems);
    pthread_mutex_unlock(&item_lock);
}

void consumer(void *arg2)
{
    int i=0, myitem;

    for( ; ; )
    {
        pthread_mutex_lock(&item_lock); /* Adquirir derecho a un
                                          elemento */
        while((nitems<=0)&& !producer_done)
            pthread_cond_wait(&nitems,&item_lock);
        if((nitems<=0) && producer_done)
        {
            pthread_mutex_unlock(&item_lock);
            break;
        }
        nitems--;
        pthread_mutex_unlock(&item_lock);

        i++;
        printf("Se ejecuta CONSUMER...%d\n",i);
        get_item(&myitem);
        sum+=myitem;

        pthread_mutex_lock(&slot_lock); /* renunciar derecho a una
                                          ranura */
        nslots++;
        pthread_cond_signal(&nslots);
        pthread_mutex_unlock(&slot_lock);
    }
}

```

```

int main(void)
{
    pthread_t prodtid;
    pthread_t constid;
    int i, total;

    total=0;
    for(i=1;i<=SUMSIZE;i++)
        total+=i*i;
    printf("La suma actual debe ser %d\n",total);

    /* CREAR HILOS */
    pthread_create(&constid,NULL,(void* (*)(void*))&consumer,NULL);
    pthread_create(&prodtid,NULL,(void* (*)(void*))&producer,NULL);

    /* ESPERAR A QUE LOS HILOS TERMINEN */
    pthread_join(prodtid,NULL);
    pthread_join(constid,NULL);
    printf("Los threads produjeron la suma: %d\n",sum);
    system("PAUSE");
    exit(0);
    return 1;
}

```

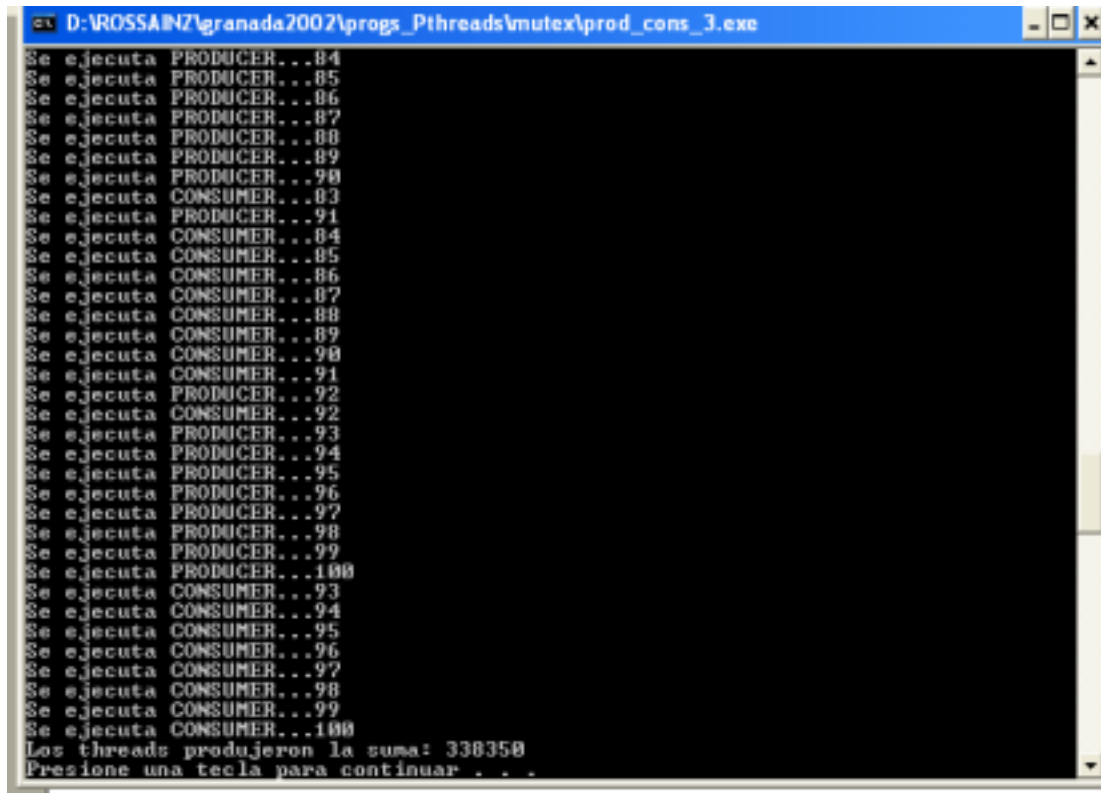
El productor de este programa se bloquea hasta que se cumple el predicado *nslots>0*. Su prueba puede escribirse: *while(!(nslots>0))*, o bien *((nitems>0) || producer_done)*, es decir, el consumidor debe actuar si hay un elemento disponible o si el productor ya terminó. Por tanto la prueba aquí es si el *while ((nitems<=0)&& !(producer_done))*. El consumidor terminará solo si el productor ya acabó y no quedan más elementos. Un posible resultado de ejecución sería el siguiente:

```

D:\ROSSAINZ\granada2002\progs_Pthreads\mutex\prod_cons_3.exe
La suma actual debe ser 338350
Se ejecuta PRODUCER...1
Se ejecuta CONSUMER...1
Se ejecuta PRODUCER...2
Se ejecuta CONSUMER...2
Se ejecuta PRODUCER...3
Se ejecuta CONSUMER...3
Se ejecuta PRODUCER...4
Se ejecuta PRODUCER...5
Se ejecuta PRODUCER...6
Se ejecuta PRODUCER...7
Se ejecuta PRODUCER...8
Se ejecuta PRODUCER...9
Se ejecuta PRODUCER...10
Se ejecuta PRODUCER...11
Se ejecuta CONSUMER...4
Se ejecuta CONSUMER...5
Se ejecuta CONSUMER...6
Se ejecuta CONSUMER...7
Se ejecuta CONSUMER...8
Se ejecuta CONSUMER...9
Se ejecuta CONSUMER...10
Se ejecuta CONSUMER...11
Se ejecuta PRODUCER...12
Se ejecuta CONSUMER...12

```

solo si el productor ya acabó y no quedan más elementos. Un posible resultado de ejecución sería el siguiente:



```
D:\ROSSAINZ\granada2007\progs_Pthreads\mutex\prod_cons_3.exe
Se ejecuta PRODUCER...84
Se ejecuta PRODUCER...85
Se ejecuta PRODUCER...86
Se ejecuta PRODUCER...87
Se ejecuta PRODUCER...88
Se ejecuta PRODUCER...89
Se ejecuta PRODUCER...90
Se ejecuta CONSUMER...83
Se ejecuta PRODUCER...91
Se ejecuta CONSUMER...84
Se ejecuta CONSUMER...85
Se ejecuta CONSUMER...86
Se ejecuta CONSUMER...87
Se ejecuta CONSUMER...88
Se ejecuta CONSUMER...89
Se ejecuta CONSUMER...90
Se ejecuta CONSUMER...91
Se ejecuta PRODUCER...92
Se ejecuta CONSUMER...92
Se ejecuta PRODUCER...93
Se ejecuta PRODUCER...94
Se ejecuta PRODUCER...95
Se ejecuta PRODUCER...96
Se ejecuta PRODUCER...97
Se ejecuta PRODUCER...98
Se ejecuta PRODUCER...99
Se ejecuta PRODUCER...100
Se ejecuta CONSUMER...93
Se ejecuta CONSUMER...94
Se ejecuta CONSUMER...95
Se ejecuta CONSUMER...96
Se ejecuta CONSUMER...97
Se ejecuta CONSUMER...98
Se ejecuta CONSUMER...99
Se ejecuta CONSUMER...100
Los threads produjeron la suma: 338358
Presione una tecla para continuar . . .
```