

Taller de programación I (7542)

Hilos de ejecución (Threads)- Parte I

Leandro H. Fernández
Revisión 3

Objetivo y alcance

Este apunte tiene como objetivo darle al alumno las herramientas necesarias para programar utilizando concurrencia. Las explicaciones sobre el funcionamiento de los sistemas operativos fueron simplificadas para agilizar la lectura en algunos casos, y para dar explicaciones generales sin entrar en las diferencias entre arquitecturas, en otros.

¿Qué son los hilos de ejecución?

Los programas de computadora más simples consisten en un algoritmo cuyos pasos se ejecutan uno a uno, dentro de un proceso en el sistema operativo. El archivo resultante de la compilación o la interpretación del código fuente tiene un punto de inicio y un final. Y en cada instante de tiempo el "cabezal" de ejecución está posicionado sobre una línea de código exclusivamente.

Los hilos de ejecución son el resultado de dividir la ejecución del proceso en dos o más partes. De forma que en un mismo instante de tiempo existan dos o más cabezales de ejecución independientes. Cada uno posicionado sobre una línea de código arbitraria. Permitiendo que un proceso realice varias tareas al mismo tiempo, con cierto nivel de independencia.

Implementación

Cada sistema operativo implementa el funcionamiento de los procesos y los hilos de ejecución a su manera. Pero en la mayoría de los casos se respeta el concepto de que dentro de un proceso existen uno o más hilos de ejecución (*threads*). De esta forma llamamos *monothread* (o *singlethread*) a una aplicación con un solo hilo y *multithread* a una con varios.

Por otro lado, la posibilidad de que dos hilos de ejecución realmente tengan lugar en forma simultánea depende de la cantidad de procesadores del equipo en que corre la aplicación. Cuando se trata de un equipo de un sólo procesador, los distintos threads de los procesos existentes tienen a su disposición la CPU por turnos. Es decir, por división del tiempo. En esta situación es común que el sistema operativo regule el uso de la CPU mediante un sistema de agenda conocido como **scheduler** [[http://en.wikipedia.org/wiki/Scheduling_\(computing\)](http://en.wikipedia.org/wiki/Scheduling_(computing))].

Sin ahondar en detalles sobre el funcionamiento de *scheduler* podemos mencionar que: en la mayoría de las plataformas y sistemas operativos con que nos encontramos en la

actualidad, el sistema operativo distribuye el tiempo de CPU guardando un registro del estado de cada hilo que no se ejecuta. Cuando un hilo debe prepararse para su ejecución, el sistema operativo restablece el estado de la CPU reproduciendo la situación exacta al momento en que el hilo detuvo su ejecución anterior. Lo deja correr un tiempo arbitrario que depende de varios factores. Cuando el tiempo se termina nuevamente se almacena el estado de CPU en memoria y se prepara otro hilo para su ejecución. Al estado del procesador almacenado se le llama contexto y al cambio entre el contexto de un hilo y otro se lo llama cambio de contexto (**context switch**).

De esta forma, desde el punto de vista de un hilo, la mecánica de compartición de CPU es transparente. El nombre genérico que se le da a la práctica de compartir los recursos entre hilos es **multitasking** [http://en.wikipedia.org/wiki/Computer_multitasking]. El diseño de *multitasking* donde el sistema operativo decide cuándo suspender una tarea para darle lugar a otra se llama **preemptive multitasking** y se utiliza [AmigaOS](#), la familia [Windows NT](#) (incluyendo [XP](#) y [Vista](#)), [Linux](#), [*BSD](#), OS/2 2.X - OS/2 Warp 3 - 4.5, [Mac OS X](#) y Windows 95/98/ME (sólo para aplicaciones de 32-bits).

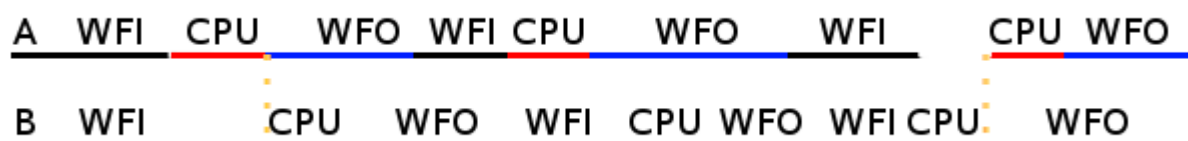
Utilidad

No está de más preguntarse hasta qué punto sirve lanzar dos tareas cuando tengo sólo una CPU en el equipo donde se ejecuta el proceso. Dado que la CPU será utilizada alternativamente por una y otra a través del tiempo. ¿Cómo puede resultar esta división en una ganancia?

Lo cierto es que las ventajas de utilizar programación concurrente dependerán de la implementación del sistema operativo y de la cantidad de procesadores (o núcleos) del equipo. Es fácil ver las ventajas si se trata de un sistema con varios procesadores, ya que disponer de tres CPUs me permitirá ejecutar hasta tres hilos en forma simultánea. Sin embargo, la ejecución paralela puede ser más eficiente en un sistema de un sólo procesador. Y esto se debe a que durante el ciclo de vida de un hilo de ejecución existen tiempos muertos de espera en interfaces de entrada/salida.

Imaginemos un hilo de ejecución que recibe dos valores por entrada estándar y retorna la suma, y esto lo repite en forma indefinida. Supongamos que el usuario tarda tres segundos en ingresar el par de valores y que la suma requiere el uso de la CPU por medio segundo. Y finalmente se requiere un segundo más para imprimir el resultado en pantalla. Con estos valores —completamente arbitrarios y tomados como ejemplo sólo a los fines pedagógicos— se hace evidente que por cada ciclo de cuatro segundos y medio hay tan sólo medio segundo de uso de CPU. Es aquí donde el *multitasking* puede aprovecharse a pesar de tener un sólo procesador.

En un sistema operativo que implementa **preemptive multitasking**, dos tareas con un comportamiento como el indicado anteriormente pueden ejecutarse en forma paralela. Asumamos que la entrada y la salida de cada tarea es independiente. Simplifiquemos el funcionamiento de cada ciclo en tres macro etapas: espera por entrada, cálculo y espera por salida. Dado que sólo la etapa de cálculo podrá tener lugar en un sólo hilo al mismo tiempo, la ejecución repetida podría generar la siguiente línea de tiempo:



En el gráfico hemos dibujado dos líneas paralelas representando la etapa en que se encuentran los hilos A y B. Etiquetando el estado de espera de entrada con la sigla WFI, el estado de cálculo con CPU y finalmente WFO para la etapa de espera de salida.

La situación esquematizada es completamente arbitraria. En el inicio ambos hilos están esperando que el usuario ingrese los valores. Evento que termina primero para el hilo A quien inmediatamente toma la CPU. Y cuando lo mismo ocurre con B, éste debe esperar que A libere el procesador para iniciar su etapa de cálculo. La espera se representó con un corte en la línea de tiempo.

El segundo evento de cálculo tiene lugar en tiempo distintos para cada hilo, por lo que no hay tiempo de espera de CPU. Y en el tercer caso es el hilo B quien toma primero la CPU y A quien debe esperar que esta se libere.

Es importante notar que en la práctica la operación de cálculo de cualquier hilo podría ser interrumpida para darle lugar a otra. Pero dejaremos ese caso particular para tratarlo en la sección titulada ¿Por qué se deben proteger los recursos compartidos?

Nivel de independencia

Los hilos de ejecución nos permiten realizar tareas simultáneas e independientes. Pero esta independencia es limitada ya que por lo general los hilos de un mismo proceso están relacionados de alguna manera; comparten un objetivo final.

Cuando una aplicación tiene que realizar dos tareas que no dependen una de otra es plausible utilizar un diseño de ejecución en paralelo. Al mismo tiempo, por tratarse de tareas de un proceso particular existirá cierta información que será compartida por ambas. Si así no fuera, lo mismo daría que se tratase de dos procesos distintos.

Esto quiere decir que parte de la tarea diseño de un sistema, cuando se utilice concurrencia, implicará decidir qué trabajos se realizan dentro de un proceso y cuales se realizan en procesos separados. Decisión que se toma en función de los recursos que comparten los procesos y los que comparten los hilos de ejecución.

Recursos compartidos

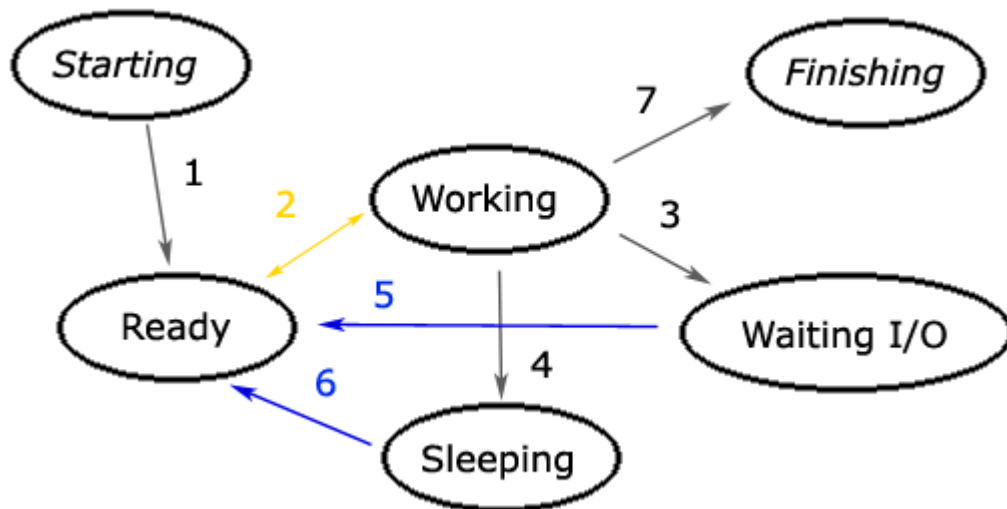
	Entre procesos	Entre hilos
Variables locales no <i>static</i> y automáticas	No	No
Variables globales y locales <i>static</i>	No	Sí
<i>Code segment</i>	No *	Sí
<i>Heap</i>	No	Sí
<i>File descriptors</i> (Archivos, sockets, etcétera)	No **	SI
Colas I/O	No	No ***
Registros de procesador	No	No

* En algunos sistemas operativos el *code segment* es el mismo para distintos procesos.

** Cuando un proceso lanza a otro, dependiendo del SO y de los parámetros de lanzamiento, puede heredar sus archivos abiertos.

*** Depende del sistema operativo.

Estados posibles de los hilos



Starting (Iniciando): el hilo es creado por el sistema operativo.

Ready (Listo): el hilo está listo para ejecutarse; hacer uso de un procesador.

Working (Trabajando): el hilo está haciendo uso del procesador.

Waiting I/O (Esperando E/S): el hilo está bloqueado esperando datos en la entrada o la salida.

Sleeping (Durmiendo): el hilo hizo una pausa de un tiempo determinado.

Finishing (Finalizando): el hilo terminó su ejecución.

1. Cuando el hilo termina de crearse se pasa de inmediato al estado Ready en el que permanecerá hasta que el scheduler del sistema operativo decida darle tiempo de uso de CPU.
2. El hilo puede pasar de Ready a Working sólo cuando el scheduler lo determina. En cambio, puede pasar de Working a Ready usualmente de dos formas: el scheduler determina que se acabó su tiempo de CPU o bien el propio hilo ofrece tiempo de CPU.
3. En todo momento que un hilo realiza una operación de lectura o escritura bloqueante pasa al estado Waiting I/O. Ver punto cinco.
4. El hilo puede necesitar esperar un tiempo arbitrario sin realizar tarea alguna. Cuando esta suspensión es solicitada no hay garantías de que el tiempo se cumpla con precisión. Ver punto seis.
5. Cuando la operación de lectura o escritura termina, el hilo pasa a estado Ready hasta que el scheduler lo disponga.
6. Cuando la suspensión solicitada por un hilo finaliza, éste pasa al estado Ready hasta que el scheduler lo disponga. Notar que de esta manera el tiempo real de la suspensión será siempre mayor o igual al solicitado.

¿Por qué se deben proteger los recursos compartidos?

Cuando dos o más hilos de ejecución deben interactuar con un recurso común se necesita un mecanismo de protección o exclusión. Ya que determinadas operaciones de lectura o escritura sobre el recurso darían resultados inesperados en caso de realizarse

en forma simultánea. Por ejemplo, si dos hilos de ejecución incrementan el valor de una misma variable. Como podemos interpretar en el siguiente código.

Código en C

```
int global;

void func() {
    global++;
}
```

Aquí tenemos una variable global llamada justamente "global" que es incrementada por la función "func()". Y cuyo código en *assembly* podemos ver a continuación.

Código en Assembly

```
#void func(){
push %ebp
mov %esp,%ebp
# global++;
mov 0x8049540,%eax
add $0x1,%eax
mov %eax,0x8049540
#}
pop %ebp
ret
```

Ahora imaginemos que dos hilos de ejecución —llamados A y B a los fines de esta explicación— generan constantemente llamadas a la función. Supongamos también que nos encontramos en una plataforma con una sola CPU y que para una serie de ciclos de reloj tenemos los estados de las columnas indicados en cada línea de la tabla.

Posible secuencia de ejecución

Clock	global	Thread activo	ip de A	eax de A	ip de B	eax de B
1	0	A	mov 0x8049540,%eax	0		
2	0	A	add \$0x1,%eax	1		
3	0	B			mov 0x8049540,%eax	0
4	0	B			add \$0x1,%eax	1
5	1	B			mov %eax,0x8049540	1
6	1	A	mov %eax,0x8049540	1		

En el primer ciclo el hilo A copia el valor de **global** al registro eax, en este caso cero. En el segundo ciclo le suma uno y el sistema operativo realiza un cambio de contexto para

darle el uso de la CPU al hilo B. En este momento el estado de `eax` para el hilo A es uno y así lo guarda el sistema operativo en memoria para restablecer el estado más tarde. En el tercer ciclo se ejecuta la llamada a **`func()`** del hilo B, que inicia copiado el valor de global al registro `eax`. Luego le suma uno y devuelve este resultado a la posición de memoria de global. En este momento el sistema operativo decide volver a darle tiempo de CPU al hilo A. Guarda el estado para el hilo B y restituye los valores para el contexto de A. Como indicamos en el párrafo anterior, el hilo A tiene su registro `eax` en uno. Adicionalmente podemos ver que la instrucción siguiente al momento de guardar el contexto era la que devuelve el valor del registro a memoria. De esta forma en el sexto ciclo se escribe un uno en la variable **`global`**.

Tras haberse ejecutado en forma completa la función **`func()`** dos veces esperaríamos que la variable **`global`** valga dos, dado que arrancó en cero. Pero como podemos ver, para el caso particular esto no es así. Y desde luego esto no es admisible. La necesidad de un método de sincronización que evite este y otros casos similares es evidente.

Sincronismo

Existen varios métodos de sincronización entre hilos y cada sistema operativo usualmente implementa un subconjunto de ellos. Por otra parte los lenguajes informáticos pueden proveer los mecanismos de programación paralela como parte de ellos, como bibliotecas de funciones o bien no proveer programación concurrente en absoluto.

Dado que a través de diferentes implementaciones los mecanismos son conceptualmente equivalentes. Nos concentraremos en la implementación de la biblioteca **`pthread`** (**`POSIX threads`**) que está disponible para varios lenguajes y sistemas operativos. Trasladar los conocimientos a otros casos es usualmente trivial.

Vale aclarar que POSIX es un conjunto de estándares, para sistemas operativos tipo UNIX, definidos por la IEEE con el fin de permitir el desarrollo de aplicaciones que se puedan ejecutar en distintas plataformas. Esta definición es sumamente simplificada, para mayor detalle se puede visitar <http://es.wikipedia.org/wiki/POSIX>

Los procesos más comunes de sincronización son la exclusión mutua, que asegura que dos o más hilos no acceden a un recurso en el mismo momento; y la señalización, que permite esperar a que ocurra un evento determinado en otro hilo.

Recursos para sincronización

Mutex (mutual exclusion)

Es un objeto¹ con dos estados posibles, tomado y liberado, que puede ser manipulado desde varios hilos simultáneamente. Cuando un hilo solicita el mutex lo recibe de inmediato si está liberado. Cualquier otro hilo que lo solicite posteriormente quedará suspendido a la espera del mutex. Si el primer hilo lo libera, alguno de los hilos en espera lo recibirá a continuación. Pudiendo esto repetirse hasta que no haya más hilos y el mutex quede nuevamente liberado.

El nombre de "exclusión mutua" hace alusión a la posibilidad de que dos o más hilos se excluyan mutuamente del uso de un recurso. Un concepto similar es el de "**`critical section`**" donde los hilos son excluidos mutuamente de una porción de código.

1. No hablamos de orientación a objetos sino de un recurso del sistema operativo.

Si pensamos en el ejemplo presentado anteriormente donde se incrementaba una misma variable desde dos hilos. La solución que un mutex aportaría se implementa de la siguiente manera:

```
1. int global;
2. // Se declara e inicializa un mutex para global
3.
4. void func() {
5.     // Se solicita el mutex
6.     global++;
7.     // Se libera el mutex
8. }
```

Demás está decir que hemos comentado conceptualmente los cambios en el código.

Ahora, cuando el primer hilo (A) llegue a la línea cinco encontrará el mutex liberado. Lo tomará inmediatamente y procederá a la línea seis. Si en ese instante otro hilo (B) llega a la línea cinco suspenderá su ejecución hasta que (A) pase por la línea siete y libere el mutex.

Es importante notar que es responsabilidad del programador escribir código que proteja un recurso determinado mediante un mutex. El recurso y el mutex no tienen otro nivel de asociación que el dado por el código escrito.

Condition variables (semaphore, events)

Dependiendo del sistema operativo o la biblioteca utilizada se dispone de uno o varios métodos de señalización. Estos objetos permiten que un hilo pueda notificar a otro que un evento ha ocurrido. De forma que uno o más hilos (por ejemplo A, B y C) puede bloquearse esperando que un cuarto hilo (D) señale un evento. Cuando esto ocurre, uno de los hilos A, B y C reanuda su operación. Lo que ocurra a continuación dependerá del código de cada hilo y más generalmente del motivo por que cual se utilizó el esquema de señalización. Además es posible que tras la señalización todos los hilos en espera se reanuden.

En el caso de la biblioteca **pthread** el elemento de señalización disponible es la *condition variable*. A diferencia de los mutex que permiten sincronizar los hilos en función del acceso a un dato, este mecanismo permite sincronizar en función del valor del dato. Aplicable al caso en que un hilo necesita bloquearse hasta que una variable tenga un valor determinado. Por un ejemplo concreto podemos tomar un contador que inicia en cero y se necesita esperar que llegue a quince.

Pensemos en dos hilos que incrementan el contador constantemente en uno. Y un hilo más que espera a que el contador llegue a quince. Los hilos A y B ejecutarán la función **inc()** dentro de un bucle y el hilo C ejecutará **wait()**.

```
1. int count = 0;
2. // declaración e inicialización de mutex y condition variable
3.
4. void inc() {
5.     // Toma el mutex
6.     ++count;
7.     if (count == 15)
8.         // Señaliza la condition variable
```

```

9.      // Libera el mutex
10. }
11.
12. void wait() {
13.     // Toma el mutex
14.     if (count < 15) {
15.         // Espera la señalización
16.         printf("count llegó a 15");
17.     }
18.     // Libera el mutex
19. }

```

Nuevamente notar que los comentarios representan llamadas a la API de pthread que aún no hemos estudiado y están aquí nombradas conceptualmente.

En la línea dos podemos ver que se declara e inicializa un *mutex* y una *condition variable*. Esto se debe a que siempre se debe usar una *condition variable* con un mutex asociado.

En la función **inc()** los hilos A y B tomarán alternativamente el mutex que protege a **count**, incrementarán su valor en uno y si ha llegado a quince enviarán la señalización. Luego liberarán el mutex.

Por otra parte el hilo C tomará el mutex, verificará que **count** todavía no llegó al valor deseado, y bloqueará su ejecución en espera de la señalización. En la línea quince la API liberará el mutex internamente para que los otros hilos puedan trabajar con **count** incrementándola.

Cuando alguno de ellos haga el paso de 14 a 15, señalice la condition variable, y libere el mutex: el hilo C pasará a la línea de código dieciséis con el mutex nuevamente tomado por él.

Es importante recalcar que en la línea dieciséis, dentro del código de pthread, ocurren tres eventos importantes en forma atómica: se libera el mutex, se espera la señal, y se toma el mutex nuevamente.

Por otra parte cabe preguntarse para que está la consulta por el valor de count en la línea catorce. ¿Por qué no tomar el mutex y pasar de inmediato a la espera de la señal? Lo que ocurre aquí es que los hilos son completamente asincrónicos. Al punto de que habiendo lanzado el hilo C en primer lugar, y luego A y B es perfectamente posible que los hilos A y B ejecuten **inc()** más de quince veces entre los dos antes de que el hilo C llame a la función **wait()**. Y en ese caso el bloqueo en espera de la señal sería infinito porque la señalización habría ocurrido antes.

Al preguntar por el valor de **count** con el mutex tomado sabemos que nadie puede incrementarla. Si es menor a quince podemos entonces entrar en espera de la señal. Ya que en ese momento se liberará el mutex y **count** continuará creciendo por el efecto de los hilos A y B.

POSIX Threads Programming

Thread

Puede parecer un poco obvio pero hay que tener en cuenta que los procesos que no utilizan programación concurrente constan de todas formas de un hilo de ejecución.

Cuando utilizamos pthread para lanzar un hilo, nuestro proceso consta entonces de dos hilos. El que lanzamos y el hilo que ejecuta la función **main()** de nuestra aplicación.

Para empezar observemos de qué forma se lanza un hilo de ejecución con la API de pthread.

```
1. #include <pthread.h>
2. #include <stdio.h>
3. #define NUM_THREADS 5
4.
5. void *PrintHello(void *threadid)
6. {
7.     long tid;
8.     tid = (long)threadid;
9.     printf("Hello World! It's me, thread #%ld!\n", tid);
10.    return NULL;
11. }
12.
13. int main (int argc, char *argv[])
14. {
15.     pthread_t threads[NUM_THREADS];
16.     int rc;
17.     long t;
18.     for(t=0; t<NUM_THREADS; t++){
19.         printf("In main: creating thread %ld\n", t);
20.         rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
21.         if (rc){
22.             printf("ERROR; return code from pthread_create() is %dn", rc);
23.             exit(-1);
24.         }
25.     }
26.     pthread_exit(NULL);
27. }
```

La línea uno incluye la declaración de las funciones de pthread. Será necesaria siempre que se utilice la biblioteca. Adicionalmente se deberá incluir el código binario en la etapa de link.

En la línea quince se declara un array de identificadores de hilos (tipo pthread_t) utilizados secuencialmente en la línea veinte a través de la llamada para crear un hilo. Esta función recibe como primer argumento la dirección de memoria del identificador que se asociará con el hilo. Como segundo argumento los atributos del hilo, que en este caso es NULL para que utilice los valores por defecto. En tercer lugar la función espera la dirección de una función que será llamada por el nuevo hilo, y que siempre debe recibir un puntero genérico como argumento y retornar un puntero genérico como resultado. Finalmente la función espera un puntero genérico que será pasado a la función como argumento.

En este ejemplo se crearán cinco hilos. Cada uno ejecutará la función "PrintHello" con el valor de la variable **t** como argumento. De manera que el segundo hilo imprimirá en pantalla "Hello World! It's me, thread #1". Es importante no perder de vista que en este ejemplo estamos usando el puntero genérico para pasar un valor arbitrario. En la práctica esto no es lo más común. En cambio suele pasarse un puntero a una posición de memoria con información relevante. Muchas veces en forma de una estructura o de una clase (si estamos programando C++). Y esta memoria deberá estar protegida ya que naturalmente es accesible desde más de un hilo. Obviamente la protección depende definitivamente del diseño, como se explicó anteriormente.

Cada hilo termina su ejecución retornando NULL como resultado. Lo que provoca una llamada implícita a **pthread_exit()** con el valor retornado. En cambio debe llamarse en forma explícita en el hilo principal donde se ejecuta **main()**.

Este ejemplo carece de un paso esencial para la programación concurrente. Y es que el cuerpo principal (ejecutado por el hilo inicial de la aplicación) lanza los cinco hilos y termina su ejecución sin verificar que todos los hilos lanzados hayan terminado. Esto es un error grosero.

Para que un hilo sepa si otro terminó su ejecución existe la función **pthread_join()** que recibe el identificador del hilo y un doble puntero **void** donde devolverá el resultado del mismo. Esta función bloquea hasta que el hilo en cuestión termine si es que aún se está ejecutando.

Podemos ahora corregir el ejemplo anterior:

```
1. #include <pthread.h>
2. #include <stdio.h>
3. #define NUM_THREADS 5
4.
5. void *PrintHello(void *threadid)
6. {
7.     long tid;
8.     tid = (long)threadid;
9.     printf("Hello World! It's me, thread #%ld!\n", tid);
10.    return NULL;
11. }
12.
13. int main (int argc, char *argv[])
14. {
15.     pthread_t threads[NUM_THREADS];
16.     int rc;
17.     long t;
18.     for(t=0; t<NUM_THREADS; t++){
19.         printf("In main: creating thread %ld\n", t);
20.         rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
21.         if (rc){
22.             printf("ERROR; return code from pthread_create() is %dn", rc);
23.             exit(-1);
24.         }
25.     }
26.     for(t=0; t<NUM_THREADS; t++){
27.         pthread_join(threads[t], NULL);
28.     }
29.     pthread_exit(NULL);
30. }
```

El lazo de la línea veintiséis se bloqueará en cada llamada a **pthread_join()** de un hilo que aún no haya terminado. Pero eventualmente pasará por los cinco identificadores. Y sólo terminará cuando todos los hilos hayan finalizado. En el ejemplo ignoramos el valor de retorno pasando NULL en el segundo argumento.

Veamos ahora cómo escribir código en C que hace uso de *mutex* y *condition variables* de pthread.

Mutex

Completemos el ejemplo anterior:

```
1. #include <pthread.h>
2.
3. int global;
4. pthread_mutex_t mutex_global = PTHREAD_MUTEX_INITIALIZER;
5.
6. void func() {
7.     pthread_mutex_lock(&mutex_global);
8.     global++;
9.     pthread_mutex_unlock(&mutex_global);
10. }
11.
12. ...
13. pthread_mutex_destroy(&mutex_global);
```

En la línea cuatro estamos declarando una variable de tipo **pthread_mutex_t** que es el mutex propiamente dicho. Y al mismo tiempo la estamos inicializando con los atributos por defecto al asignarle el valor **PTHREAD_MUTEX_INITIALIZER**. Alternativamente, si necesitamos otros atributos para el mutex, podemos llamar a la función **pthread_mutex_init()**.

En la línea siete intentamos tomar el mutex. En esa línea se bloqueará la ejecución hasta que el mutex pueda ser tomado. Si las funciones **pthread_mutex_lock()** y **pthread_mutex_unlock()** retornan distinto de cero estamos en una condición de error. En el ejemplo omitimos la verificación por simplicidad.

La decisión de llamar al mutex "**mutex_global**" no es casual. Como se aclaró anteriormente, el código escrito es quien asocia al mutex con el recurso protegido. Se debe crear un mutex para cada recurso compartido y utilizarlo en todas las porciones de código que utilizan ese recurso. También es importante entender que el recurso puede ser una variable, muchas variables, un archivo o cualquier otro objeto.

Cuando el mutex termina de utilizarse o antes de terminar el proceso debe liberarse el recurso como muestra la línea trece.

Condition variable

Hemos tomado este ejemplo directamente de la documentación de la biblioteca.

```
1. #include <pthread.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4.
5. #define NUM_THREADS 3
6. #define TCOUNT 10
7. #define COUNT_LIMIT 12
8.
9. int count = 0;
10. int thread_ids[3] = {0,1,2};
11. pthread_mutex_t count_mutex;
12. pthread_cond_t count_threshold_cv;
```

```

13.
14. void *inc_count(void *t)
15. {
16.     int j,i;
17.     double result=0.0;
18.     long my_id = (long)t;
19.
20.     for (i=0; i<TCOUNT; i++) {
21.         pthread_mutex_lock(&count_mutex);
22.         count++;
23.
24.         /*
25.          Verificar el valor de count y señalar al hilo que está en espera
26.          cuando el valor COUNT_LIMIT es alcanzado. Esto ocurre mientras
27.          el mutex está tomado.
28.         */
29.         if (count == COUNT_LIMIT) {
30.             pthread_cond_signal(&count_threshold_cv);
31.             printf("inc_count(): thread %ld, count = %d Threshold reached.n",
32.                 *my_id, count);
33.         }
34.         printf("inc_count(): thread %ld, count = %d, unlocking mutexn",
35.             *my_id, count);
36.         pthread_mutex_unlock(&count_mutex);
37.
38.         /* Liberar adrede el procesador para darle tiempo a los otros hilos */
39.         sleep(1); // Alternativas pthread_yield(); o sleep(0);
40.     }
41.     return NULL;
42. }
43.
44. void *watch_count(void *t)
45. {
46.     long my_id = (long)t;
47.
48.     printf("Starting watch_count(): thread %ldn", *my_id);
49.
50.     /*
51.      Tomar el mutex y esperar la señal. Notar que la función
52.      pthread_cond_wait liberará el mutex en forma automática y
53.      atómica mientras espera. La verificación de que no se llegó
54.      a COUNT_LIMIT evita que este hilo se suspenda en forma indefinida.
55.     */
56.     pthread_mutex_lock(&count_mutex);
57.     if (count<COUNT_LIMIT) {
58.         pthread_cond_wait(&count_threshold_cv, &count_mutex);
59.         printf("watch_count(): thread %ld Condition signal received.n", my_id);
60.         count += 125;
61.         printf("watch_count(): thread %ld count now = %d.n", my_id, count);
62.     }
63.     pthread_mutex_unlock(&count_mutex);
64.     return NULL;
65. }
66.
67. int main (int argc, char *argv[])
68. {
69.     int i, rc;
70.     long t1=1, t2=2, t3=3;

```

```

71. pthread_t threads[3];
72. pthread_attr_t attr;
73.
74. /* Se inicializan el mutex y la condition variable */
75. pthread_mutex_init(&count_mutex, NULL);
76. pthread_cond_init (&count_threshold_cv, NULL);
77.
78. /* Para mayor portabilidad, crear hilos de forma
79.    que admintan join, explícitamente */
80. pthread_attr_init(&attr);
81. pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
82. pthread_create(&threads[0], &attr, watch_count, (void *)t1);
83. pthread_create(&threads[1], &attr, inc_count, (void *)t2);
84. pthread_create(&threads[2], &attr, inc_count, (void *)t3);
85.
86. /* Esperar que todos lo hilos terminen */
87. for (i=0; i<NUM_THREADS; i++) {
88.     pthread_join(threads[i], NULL);
89. }
90. printf ("Main(): Waited on %d threads. Done.\n", NUM_THREADS);
91.
92. /* Limpiar y salir */
93. pthread_attr_destroy(&attr);
94. pthread_mutex_destroy(&count_mutex);
95. pthread_cond_destroy(&count_threshold_cv);
96. pthread_exit(NULL);
97. }

```

La función **inc_count()** es la encargada de incrementar el contador y, llegado el caso, señalar la condition variable. Se puede observar que cada ciclo del bucle toma el mutex, incrementa, y luego lo libera. Y en caso de que se llegue al valor límite señala la condition variable llamando a **pthread_cond_signal()**. Esto ocurre en los hilos 1 y 2 ya que ambos se lanzan con la misma función como punto de entrada.

El hilo cero ejecuta la función **watch_count()** que estará a la espera de la señal utilizando la llamada **pthread_cond_wait()** que recibe tanto la condition variable como el mutex asociado. Esto le permitirá liberar el mutex mientras espera la señal.

Semaphore

El estándar POSIX.1 que comprende la declaración de **pthread** no incluye **semáforos**. Sin embargo, una publicación posterior (POSIX.1-2001) detalla la API que, de estar disponible, se encuentra declarada en *semaphore.h*

El uso de semáforos puede simplificar significativamente la resolución de ciertos problemas de programación paralela. Pero el uso de esta implementación nos lleva, en la actualidad, a código más propenso a dejar de compilar cuando se migra a otra plataforma.

pthread y recursos no liberados

Si se utiliza **valgrind** sobre una aplicación con **pthread** es muy normal encontrar *still reachable blocks* de memoria no liberada. Estos avisos pueden ignorarse siempre que valgrind indique que la solicitud de memoria se hizo dentro de una de las llamadas de la

API de pthread.

```
==17698== 272 bytes in 2 blocks are possibly lost in loss record 1 of 1
==17698== at 0x40206D5: calloc (vg_replace_malloc.c:279)
==17698== by 0x400FA87: (within /lib/ld-2.4.so)
==17698== by 0x400FB4B: _dl_allocate_tls (in /lib/ld-2.4.so)
==17698== by 0x403AB68: pthread_create@@GLIBC_2.1 (in /lib/tls/i686/cmov/libpthread-2.4.so)
==17698== by 0x80486A7: main (test.c:43)
```

Como se puede apreciar en el *call stack* anterior, la llamada a `calloc()` que solicitó la memoria no liberada corresponde al código interno de la función **pthread_create**.

Lecturas complementarias

El presente texto no pretende de ninguna manera abarcar la funcionalidad completa de la biblioteca PThread. Así como no detalla las diferencias entre los mecanismos de sincronización anteriormente mencionados.

Los alumnos deberán referirse a la documentación de la biblioteca para conocer los detalles referidos a la inicialización de threads, mutex y condition variables que no se especificaron en este texto. De la misma forma será necesario conocer los posibles valores de retorno de las funciones con el fin de validar el éxito en los casos pertinentes. La lectura de dicha documentación será necesaria para la confección de los trabajos prácticos de la materia.

Tutorial	https://computing.llnl.gov/tutorials/pthreads/
Manual de referencia	http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html
PThread en la wikipedia	http://en.wikipedia.org/wiki/POSIX_Threads