Sistemas Distribuidos

Programación con Threads

Tabla de Contenidos

I. Introducción

En la mayoría de los sistemas operativos, cada proceso, tiene su espacio de direcciones, ejecutándose las instrucciones en forma secuencial en un hilo o thread de control. Las palabras *hilo* y *thread* se utilizarán indistintamente a lo largo de este documento, tanto como conceptos de sistemas operativos como unidades de administración en cada sistema operativo que se suele utilizar. De hecho, esto es prácticamente la definición de proceso. Es decir, en una máquina podrá haber varios procesos, cada uno con:

- Un contador de programa, con la siguiente instrucción a ejecutar.
- Una pila.
- Un conjunto de registros.
- Un espacio de direcciones privadas.

Todos los procesos son así independientes entre sí. Para comunicarse usan semáforos, paso de mensajes, etc. Durante su ejecución, un proceso pasa a través de dos modos: núcleo y usuario. En el modo núcleo el sistema operativo tiene control absoluto y el usuario no puede hacer nada; en modo usuario, se tiene acceso a todos los datos definidos por el proceso.

Sin embargo, en ocasiones, se desea tener varios hilos de control que comparten el mismo espacio de direcciones, ejecutándose, de manera cuasi paralela (concurrente). Esto llevó a desarrollar el manejo múltiple de hilos o multi-threads. Cuando un proceso empieza la ejecución en un ambiente multi-thread, un solo thread es lanzado, el cual empieza con la ejecución de la función main() del programa. A menos que el programa genere explícitamente nuevos threads, continuará como un thread simple y tradicional.

En general, interesa trabajar con los threads de usuario o con una parte de la administración en espacio de usuario, aunque esto depende de la implementación en particular y no cambia, en principio, la programación. De hecho, UNIX no soporta hilos, pero se han desarrollado distintos paquetes de hilos en el espacio de usuario. En general también, los threads son “baratos" de crear y administrar, ya no es necesario que se le asignen y administren *muchos* recursos extras a los que el proceso ya tenía asignado. Por este motivo, se los suele llamar procesos ligeros (lightweight, abreviado con LWP). Cada thread tendrá:

- Su propio contador de programa.
- Su propia pila.
- Su propio conjunto de registros.

Cada hilo se ejecutará en forma secuencial igual que los procesos. Al igual que ellos, comparten el procesador (si hubiera varios procesadores se podrían ejecutar varios hilos a la vez). La idea también es mantener el concepto de que si un hilo se bloquea se puede ejecutar otro, incluso del mismo proceso. O sea, la planificación de hilos es similar a la de procesos. La diferencia está en que un proceso tiene un espacio privado de direcciones, pero los hilos de un mismo proceso comparten el mismo espacio de direcciones y esto genera una disminución importante en el cambio de contacto (context switch) de threads dentro de un mismo proceso.

Las variables globales del proceso pueden ser modificadas por todos los hilos (de ese proceso) al mismo tiempo, así como también comparten la información general del proceso:

- Tabla de ficheros abiertos.
- Temporizadores abiertos.
- Información contable: número de ficheros abiertos, etc.

Al igual que los procesos, los threads pueden estar en diferentes estados:

- Ejecución.
- Bloqueo.
- Listo.
- Terminal.

Un hilo en ejecución está activo, es decir, tiene asignada la CPU. Un hilo bloqueado, está a la espera de que otro lo saque de este estado. Un thread listo esta a la espera de que el planificador lo active. Por último, un hilo terminado, es un hilo que hizo su salida, pero no fue “enhebrado” por otro hilo (en

términos de procesos, que el padre no ha realizado el wait).

Existen dos ámbitos de planificación (aunque esto depende en la mayoría de los casos de la implementación de threads que se utilice), que se muestran en la Fig. 1:

- Ámbito de proceso: un planificador de segundo nivel planifica los hilos de cada proceso. Normalmente en este caso los threads son totalmente manejados a nivel de usuario y es posible que el sistema operativo no tenga ningún soporte ni administración especial de threads, son totalmente *transparentes* para el sistema operativo.
- Ámbito de sistema: los hilos se planifican como los "procesos pesados" o a nivel de procesos. Esto significa que compiten por la CPU como lo hacen los procesos, siguiendo las mismas políticas o políticas similares. Esto no necesariamente significa que la administración (y la *sobrecarga* implícita) de un thread sea la de un proceso.

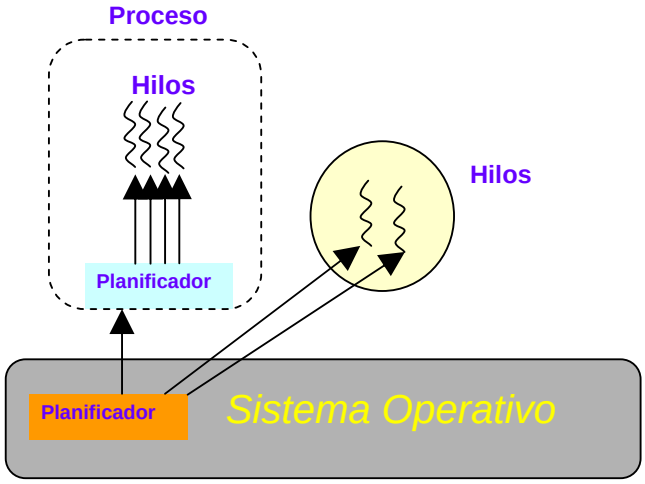

Figura 1: Ambitos de Planificación de Threads.

II. Llamadas al sistema

Existen dos implementaciones para UNIX para el manejo de threads: las librerías POSIX[1] y Sun Solaris 2. A nivel de sintaxis, la diferencia es que las funciones en POSIX comienzan con la letra "p". Por disponer en LINUX de la versión POSIX, se tomará este estándar de referencia.

El paquete típico de threads incluye llamadas para la creación y destrucción de threads, para resolver el problema de exclusión mutua, manejo de variables de condición para la sincronía entre las ejecuciones de dos o más threads y llamadas para el manejo de señales. La Tabla 1 muestra las principales funciones de la librería POSIX, clasificadas según las tareas para las que se utilizan.

Tabla 1: Llamadas de Threads de POSIX.

Descripción	Llamada
Manejo threads	pthread_create pthread_exit pthread_kill pthread_join pthread_self

[1] POSIX, acrónimo de Portable Operating System Interface for uniX, conjunto de estándares ISO e IEEE para proporcionar portabilidad a los programas que se desarrollan en el ambiente Unix. El estándar Posix.1 de threads fue aprobado en 1995.

Tabla 1 (cont.): Llamadas de Threads de POSIX.

Descripción	Llamada
Exclusión mutua	pthread_mutex_init pthread_mutex_destory pthread_mutex_lock pthread_mutex_trylock pthread_mutex_unlock
Variables de condición	pthread_cond_init pthread_cond_destroy pthread_cond_wait pthread_cond_timedwait pthread_cond_signal pthread_cond_broadcast

III. Atributos de un Thread

Cada thread cuenta con una serie de propiedades que lo hacen único. Estos atributos son del tipo:

pthread_attr_t

En la Tabla 2, se presentan las funciones que permiten manipular los diferentes atributos de un thread.

Tabla 2: Funciones para el Manejo de Atributos de Threads.

Atributo	Función
Inicialización	pthread_attr_init
Tamaño del stack	pthread_attr_setstacksize pthread_attr_getstacksize
Dirección stack	pthread_attr_setstackaddr pthread_attr_getstackaddr
Estado de desconexión	pthread_attr_setdetachstate pthread_attr_getdetachstate
Alcance	pthread_attr_setscope pthread_attr_getscope
Herencia	pthread_attr_setinheritsched pthread_attr_getinheritsched
Política de scheduling	pthread_attr_setschedpolicy pthread_attr_getschedpolicy
Parámetros de scheduling	pthread_attr_setschedparam pthread_attr_getschedparam

Los atributos/propiedades de un thread varían de una implementación a otra. Sin embargo, a manera general los atributos que definen a un thread son:

- Estado de espera: permite que otros threads esperen por la terminación de un thread en especial.
- Dirección de stack: apuntador al inicio del stack del thread.

- Tamaño del stack: longitud del stack del thread.
- Alcance (scope): define quién controla la ejecución del thread (el proceso o el núcleo del sistema operativo).
- Herencia: los parámetros de scheduling son heredados o definidos localmente.
- Política de scheduling: define qué proceso se va a ejecutar y en que instante (FIFO, Round-robin, definido por la implementación), la política de adminstración a aplicar por el administrador para este thread en particular.
- Prioridad: un valor de prioridad alto corresponde a una mayor prioridad Es posible modificar varios de estos atributos a través de diferentes llamadas de sistema.

Todas las funciones cuentan con dos parámetros. El primero de ellos, es el puntero al atributo del thread; el segundo, es el valor del atributo o un puntero a un valor.

IV. Ejemplos de Programas con Threads

Veremos ahora, una secuencia de ejemplos sencillos de programación de hilos, con la idea de que ayuden a entender con qué funciones contamos para el manejo de threads y terminar de entender como funcionan éstos. Estos ejemplos están al final del documento y siguen la numeración de cada una de las siguientes subsecciones. Para generar el ejecutable de un programa que usa llamadas de funciones de threads tipo POSIX, es necesario compilarlo con la opción -l pthread (o *enlazarlo*, dado que es necesario para la etapa del *linker* en realidad).

IV. 1. Creación y Espera de Finalización (*Enhebrado*)

Veamos el primer ejemplo de manejo de hilos. En main() se declara primero una variable mythread, que es del tipo pthread_t (definido en pthread.h), y se denomina a menudo "thread id" (también abreviado "tid").

```
/* Ejemplo 1: thread1.c */
#include <pthread.h>     /* funciones pthread_ */
#include <stdlib.h>
#include <unistd.h>      /* sleep() */

void *thread_function(void *arg)
{
      printf("El nuevo hilo te saluda!\n");
      sleep(5);
      printf("El nuevo hilo se despide!\n");
      return NULL;
}

int main(void)
{
      pthread_t mythread;
      printf("El hilo original te saluda!\n");
      if ( pthread_create( &mythread, NULL, thread_function, NULL))
      {
            printf("error en crear thread");
            abort();
      }
      if ( pthread_join (mythread,NULL))
      {
            printf("error en el join del thread");
            abort();
      }
      printf("El hilo original se despide!\n");
      exit(0);
}
```

Después de declarar mythread, se llama a la función pthread_create() para crear un hilo. Cuando un proceso empieza la ejecución en un ambiente multi-thread, un solo thread es lanzado, el cual empieza con la ejecución de la función main() del programa. A menos que el programa genere explícitamente nuevos threads (como en este primer ejemplo), continuará como un thread simple y tradicional. La sintaxis de la llamada de sistema usada para la creación de un thread es:

```
pthread_create(pthread_t *tid; const pthread_attr_t *attr, void *(*rutina)(void *), void *arg);
```

La función crea un thread y lo pone en la fila de listos a ejecutar. Como toda llamada de sistema regresa un valor de 0 si la ejecución fue satisfactoria y -1 en caso de error. Los parámetros de la llamada son:

id: sirve para almacenar el identificador del thread. El núcleo asigna un identificador único a cada thread creado.

attr: especifica todos los atributos del thread. Si se pasa un valor de NULL el thread es creado con los atributos por default, usualmente otorgados por el hilo que lo creó, tomándolo del heap.

rutina: se refiere a la función llamada por el thread, cuando éste comienza la ejecución.

arg: representa los parámetros de la función. Solo se puede pasar un parámetro, en el caso de que se requieran más, es posible pasarlos todos dentro de una estructura.

En el ejemplo, esta llamada tiene como primer argumento un puntero a mythread, y se toman los atributos por defecto (los del thread que hace la llamada). La función que el nuevo hilo ejecutará cuando comience es, en este caso, thread_function(). Cuando la función thread_function() retorna, el nuevo hilo habrá terminado.

En este ejemplo, la función que corresponde al nuevo hilo solo muestra en pantalla un mensaje al inicio de la ejecución, espera 5 segundos y concluye. En este ejemplo, está definido como NULL el cuarto argumento, dado que no necesitamos pasar ningún dato a nuestra trivial thread_function().

El programa entonces, consistirá en dos hilos después de que retorne pthread_create() (si la llamada a fue resuelta satisfactoriamente). ¿Cómo continúan estos hilos? El hilo del main() se mantiene y ejecuta secuencialmente la siguiente línea del programa (en este caso, "if (pthread_join(...))"). El nuevo hilo, cuando finalice la función, se detiene y espera a combinarse con otro hilo como parte de su proceso de limpieza.

Como se mencionó, thread_function() tarda algo más de 5 segundos en completarse. Antes de que thread_function() concluya, nuestro hilo principal ha llamado ya a pthread_join(). Cuando esto ocurre, el hilo principal se detendrá (se duerme) y esperará a que thread_function() concluya. Cuando thread_function() termine, pthread_join() retornará. Ahora el programa en ejecución tiene sólo el hilo principal de nuevo. Cuando el programa sale, todos los nuevos hilos habrán formado una sola hebra [pthread_join()]. La sintaxis para esperar que un thread en particular finalice es:

```
thread_join(pthread_t thread, void **value_ptr);
```

Esta llamada suspende la ejecución de thread que la ejecuta, hasta que el thread indicado por el parámetro thread termine. El parámetro value_ptr permite recuperar el estatus de terminación del thread esperado.

Nuestro ejemplo muestra un sencillo manejo de threads; si el nuevo hilo no se une al que lo creó, seguirá contando para el límite total de hilos del sistema. Esto significa, que si no se hace una limpieza adecuada de hilos, las nuevas llamadas a pthread_create() podrían fallar. Una vez que un hilo se enhebra, ya no puede ser esperado a través de un pthread_join(). Las funciones pthread_attr_getdetachstate() y pthread_attr_setdetachstate(), respectivamente, pueden examinar y establecer el estado de desconexión. Los valores relacionados con dicho estado son

- PTHREAD_CREATE_JOINABLE para establecer que alguien puede esperar por el thread.
- PTHREAD_CREATE_DETACHED para lo contrario.

La opción por default es que los threads sean “alcanzables" (PTHREAD_CREATE_JOINABLE).

Veamos ahora una consideración adicional sobre la creación de hilos. Imaginemos la siguiente porción de código:

```
…
pthread_create( &thread_a, NULL, thread_function, NULL);
pthread_create( &thread_b, NULL, thread_function, NULL);
pthread_create( &thread_c, NULL, thread_function, NULL);
…
```

Después de que la primera llamada pthread_create() concluya, puede asumirse que el hilo "a" (*asociado* al tid thread_a) existe o que ha finalizado y ha parado. Después de la segunda llamada a pthread_create(), tanto el hilo principal como el hilo "b" pueden asumir que el hilo "a" existe (o está parado). De cualquier forma, después de que la segunda llamada create() retorne, el hilo principal no puede asumir cuál hilo (a o b) empezará a ejecutarse antes.

Aunque ambos hilos existen es tarea del kernel y quizás también de la librería de hilos asignarles una porción del tiempo de la CPU. Y no hay ninguna regla establecida acerca de cuál empezará a ejecutarse antes. Aunque es muy probable que el hilo "a" empiece a ejecutarse antes que el hilo "b", esto no está garantizado, lo que es particularmente cierto en máquinas con más de un procesador. Si escribimos un programa que asume que el hilo "a" comienza a ejecutarse antes que el hilo "b", acabaremos con un programa que funciona correctamente el 99% de las veces. Peor aún, con un programa que funcionará el 99% de veces en nuestro sistema y el 0% en un servidor con cuatro procesadores.

Comparación con manejo de procesos (I):

Cuando un proceso crea otro nuevo proceso usando fork(), al nuevo proceso se le considera hijo y al proceso de origen padre. Esto crea una relación jerárquica, de forma tal que, por ejemplo, la función waitpid(), le indicará al proceso actual que espere a que los procesos hijos concluyan. Esta función, se usa para implementar una sencilla rutina de limpieza en nuestro proceso padre.

Para el caso de hilos, el mecanismo es diferente. Normalmente no se habla de hilo padre e hilo hijo; esto se debe, a que en los hilos POSIX, esta relación jerárquica no existe. El hilo principal puede crear otro hilo, y este hilo puede, a su vez, crear otro nuevo hilo, permitiendo el estándar POSIX ver a todos los hilos como un conjunto de elementos idénticos. Así es que el concepto de esperar a que un proceso hijo concluya, no tiene sentido. El estándar de hilos POSIX no registra ninguna información tipo padre-hijo y esta falta de genealogía hace que para esperar a que un hilo concluya, se debe especificar el thread en cuestión, indicando el tid adecuado a pthread_join().

El estándar POSIX proporciona las herramientas necesarias para manejar muti-threads de manera adecuada. El hecho de que no haya una relación padres/hijos permite diferentes alternativas para la programación de hilos. Por ejemplo, si tenemos un hilo llamado hilo 1, y el hilo 1 crea un nuevo hilo 2, no es necesario para el hilo 1 llamar a pthread_join() para el hilo 2. Cualquier otro hilo en el programa puede hacerlo. Esto permite posibilidades muy interesantes cuando se están creando programas con gran cantidad de multi-hilos. Se puede crear, por ejemplo, una lista que contenga todos los hilos detenidos y tener otro hilo, de limpieza, que sencillamente espera a que algún elemento se añada a esta lista. El hilo de limpieza llama a pthread_join() para enhebrarlo consigo mismo. De esta forma, todo el proceso de limpieza será manejado de forma cómoda y eficiente con un simple hilo.

IV. 2. Problemas de Exclusión

El código del segundo ejemplo, thread2.c, se muestra a continuación. Si ejecutamos thread2.c, tendríamos a la salida que myglobal vale 5. El resultado parece inesperado, ya que myglobal es inicializada en cero y tanto el hilo principal como el nuevo hilo, lo incrementan en 4; deberíamos ver que myglobal es igual a 8 al final del programa.

```c
/* Ejemplo 2: thread2.c */
#include <pthread.h>   /* funciones pthread_ */
#include <stdlib.h>
#include <unistd.h>    /* sleep() */

int myglobal;

void *thread_function(void *arg)
{
      int i,j;
      for ( i=0; i<4; i++ )
      {
            j=myglobal; j=j+1;
            printf("Hilo Nuevo...\n");
            sleep(1);
            myglobal=j;
      }
      return NULL;
}

int main(void)
{
      pthread_t mythread; int i;

      if (pthread_create(&mythread, NULL, thread_function, NULL))
      {
            printf("error creating thread.");
            abort();
      }
      for ( i=0; i<4; i++)
      {
            myglobal=myglobal+1;
            printf("Hilo Principal...\n");
            sleep(1);
      }
      if ( pthread_join ( mythread, NULL ) )
      {
            printf("error joining thread.");
            abort();
      }
      printf("\nmyglobal vale %d\n", myglobal);
      exit(0);
}
```

Si revisamos la función thread_function(), myglobal se copia a una variable local llamada "j". Imaginemos entonces que el thread principal incrementa myglobal justo después de que nuestro nuevo hilo copie el valor de myglobal en j. Cuando thread_function() vuelve a escribir el valor de j en myglobal, sobrescribirá la modificación que el hilo principal ha hecho. Cuando se escriben programas con hilos, se querrán evitar efectos no deseados como el descrito. Tenemos que encontrar una forma de que un hilo le indique al otro que "espere" mientras se están haciendo los cambios a myglobal. La solución a este problema es el uso de mutex.

IV. 3. Uso de Mutex

En este tercer ejemplo, thread3.c, se agregaron las llamadas específicas para exclusión mutua: pthread_mutex_lock() y pthread_mutex_unlock(). Como es de esperar, dos hilos no pueden mantener el mismo mutex bloqueado al mismo tiempo. Si el hilo "a" intenta bloquear un mutex mientras que el hilo "b" tiene el mismo mutex bloqueado, el hilo "a" se duerme. Tan pronto como el hilo "b" libere el mutex (a través de una llamada pthreadmutex_unlock()), el hilo "a" será capaz de bloquear el mutex (en otras palabras, retornará desde la llamada pthread_mutex_lock() con el mutex bloqueado).

Del mismo modo, si el hilo "c" trata de bloquear el mutex mientras que el hilo "a" lo mantiene bloqueado, el hilo "c" se quedará dormido durante algún tiempo. Todos los hilos que se duermen a partir de la llamada pthread_mutex_lock(), con un mutex previamente bloqueado, serán "encolados" para acceder a dicho mutex.

```
/* Ejemplo 3: thread3.c */
#include <pthread.h>   /* funciones pthread_ */
#include <stdlib.h>
#include <unistd.h>    /* sleep() */

int myglobal;

pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;

void *thread_function(void *arg)
{
      int i,j;
      for ( i=0; i<4; i++ )
      {
            pthread_mutex_lock(&mymutex);
            j=myglobal; j=j+1;
            printf("Hilo Nuevo...\n");
            sleep(1);
            myglobal=j;
            pthread_mutex_unlock(&mymutex);
      }
      return NULL;
}

int main(void)
{
      pthread_t mythread;
      int i;

      if (pthread_create(&mythread, NULL, thread_function, NULL))
      {
            printf("error creating thread.");
            abort();
      }
      for ( i=0; i<4; i++)
      {
            pthread_mutex_lock(&mymutex);
            myglobal=myglobal+1;
            pthread_mutex_unlock(&mymutex);
            printf("Hilo Principal...\n");
            sleep(1);
      }
      if ( pthread_join ( mythread, NULL ) )
      {
            printf("error joining thread.");
            abort();
      }
      printf("\nmyglobal equals %d\n",myglobal);
      exit(0);
}
```

Cabe aclarar que si se colocan demasiados mutexes, nuestro código no tendrá ningún tipo de concurrencia y se ejecutará mucho más lentamente que una solución con un simple hilo. Si se colocan muy pocos, el código puede errores que pueden resultar fatales. Los mutexes deberán utilizarse para “serializar el acceso a datos compartidos”. No deben usarse para datos que no se van

a compartir, y tampoco deben usarse si la lógica interna de nuestro programa asegura que sólo un hilo está accediendo a una estructura concreta de datos a la vez.

En este ejemplo, usamos un método de inicialización estático. Esto implica declarar una variable pthread_mutex_t y asignarle la constante PTHREAD_MUTEX_INITIALIZER. Puede también crearse mutex dinámicamente utilizando pthread_mutex_init(). Cada vez que se inicializa un mutex usando pthread_mutex_init(), debe destruirse con pthread_mutex_destroy(), la que libera todos los recursos empleados por el mutex cuando se crea. Hay que notar que pthread_mutex_destroy() no libera la memoria usada para almacenar pthread_mutex_t. Es tarea nuestra hacer uso de free() para liberar la memoria. Tanto pthread_mutex_init() como pthread_mutex_destroy() retornan cero en caso de no encontrar errores.

Debe desbloquearse un mutex bloqueado en cuanto sea posible y seguro (para incrementar el rendimiento). Nunca debe desbloquearse un mutex que no estuviese bloqueado, o de lo contrario, la llamada pthread_mutex_unlock() fallará retornando un valor distinto de cero.

Existe también de la posibilidad de hacer uso de la llamada pthread_mutex_trylock(). Si el mutex está desbloqueado, entonces se bloqueará, y la función retornará un valor cero. De cualquier forma, si el mutex se encuentra bloqueado esta llamada devolverá un valor que no es cero como error EBUSY. Entonces, el programa puede pasar a realizar otras cosas y tratar de bloquear al mutex después.

Una solución haciendo uso de variables de condición la provee la llamada pthread_cond_wait(), la que permite esperar a que una determinada condición sea cierta. Consideremos por ejemplo el siguiente escenario: un hilo ha bloqueado un mutex, para ver una lista enlazada, y resulta que la lista está vacía. Este hilo en concreto no podrá hacer nada, está diseñado para eliminar un nodo de la lista, pero no hay nodos disponibles. Mientras que mantiene el mutex bloqueado, nuestro hilo llamará a pthread_cond_wait(&mycond, &mymutex). La primera cosa que pthread_cond_wait() hace es desbloquear simultáneamente el mutex mymutex (para que otros hilos puedan modificar la lista enlazada) y esperar a la condición mycond, para que pthread_cond_wait() despierte, cuando reciba una señal desde otro hilo.

Al momento de estar el mutex desbloqueado, otros hilos pueden acceder y modificar la lista enlazada, posiblemente añadiendo elementos a la misma. En este momento, la llamada pthread_cond_wait() aún no ha retornado. El desbloqueo del mutex ocurre inmediatamente, pero esperar a la condición mycond, la que es una operación de bloqueo del hilo, lo cual significa, que nuestro hilo se irá a dormir sin consumir ciclos de CPU hasta que se despierte.

Nuestro hilo entonces estará durmiendo esperando a que se cumpla una determinada condición, sin realizar ningún bucle que desperdicie tiempo de CPU. Desde la perspectiva de nuestro hilo, sencillamente está esperando a que retorne la llamada pthread_cond_wait().

Ahora, para continuar con la explicación, digamos que otro hilo (lo llamaremos "thread 2") bloquea mymutex y añade un elemento a nuestra lista enlazada. Inmediatamente después de desbloquear el mutex, "thread 2" llama a la función pthread_cond_broadcast(&mycond). Haciendo esto, daremos lugar a que todos los hilos esperando a la variable condicional mycond despierten. Esto significa que nuestro primer hilo, que se encuentra en plena llamada pthread_cond_wait(), despertará.

Ahora, veamos lo que ocurre con nuestro primer hilo. Después de que "thread 2" ha llamado a pthread_cond_broadcast(&mymutex), la función pthread_cond_wait() del primer hilo realizará una última operación: volver a bloquear el mutex. Una vez que pthread_cond_wait() tiene el bloqueo de nuevo, entonces retornará y permitirá al primer hilo seguir ejecutándose. En este momento, puede comprobar inmediatamente la lista para verificar si hay algún cambio interesante.

IV. 4. Problemas de Contexto

El código del cuarto ejemplo, pthread4.c, crea un thread el cual se encarga de ejecutar la función f1(), pasándole como parámetro un entero calculado de forma aleatoria. La función tiene por objeto dormir al thread durante un determinado tiempo (parámetro de entrada) y hace uso de pthread_self() para identificar el hilo.

```
/* Ejemplo 4: thread4.c */
#include <stdio.h>
#include <pthread.h>    /* pthread_  */
#include <stdlib.h>     /* srandom() */
#include <unistd.h>     /* sleep() */

int f1(int x)
{
        int id;
        id = pthread_self();
        printf("\t Nuevo hilo %d a dormir %d segundos \n", id, x);
        sleep(x);
        x = x/2;
        printf("\t Nuevo hilo se desperto y termino con status %d \n", x);
        pthread_exit((void *)x);
}

main()
{
        pthread_t t1;
        int tmp;
        int status;

        printf("Hilo principal crea un hilo...\n");
        srandom(time(NULL));
        tmp = 1 + random() % 3;
        pthread_create(&t1, NULL, (void *)f1, (void *)tmp);
        printf("Esperando que termine hilo creado...\n");
        pthread_join(t1, (void *)&status);
        printf("Hilo %d termino con status: %d \n", t1, status);
        exit(0);
}
```

Por otro lado a diferencia de los ejemplos anteriores, se está usando para finalizar la función la llamada pthread_exit() cuya sintaxis es:

```
pthread_exit(void *status);
```

Esta llamada puede ser invocada en cualquier parte del código. El parámetro status es usado por el thread para notificar la forma en que terminó, puede ser recuperado por otro thread a través de la llamada pthread_join(), tal como se muestra en el ejemplo.

Comparación con manejo de procesos (II):

Los threads comparten el contexto del proceso, por lo que será responsabilidad del usuario, asegurarse que los threads no interfieren uno con otro. Esto ya lo vimos, cuando introdujimos mutex para el manejo de variables comunes a diferentes hilos.

Otro ejemplo de la interferencia entre hilos es la ejecución de la llamada exit(), la que provoca que todo el proceso con todos sus threads termine. De ahí la necesidad de contar con pthread_exit() para finalizar un hilo. Con referencia a la interferencia entre hilos, además de lo mencionado, deberíamos tener en cuenta que:

- *Si un thread cambia de directorio; todos los threads dentro del proceso verán ese nuevo directorio.*
- *Si un descriptor de archivos es cerrado, el archivo es cerrado para todos los threads.*
- *Si un thread entra en un ciclo infinito todo el programa se ve afectado.*
- *La llamada de sistema exec() funciona de la misma manera que en un contexto de un solo proceso, sólo que todos los threads son destruidos.*

IV. 5. Llamada de Sistema sched_yield()

En algunas aplicaciones, se necesita una ejecución *tandem* de las partes involucradas. Por tandem entendemos que un thread ejecuta una instrucción para después dejar la CPU y permitir que otro thread ejecute también ejecute una instrucción. Después de ejecutar su instrucción, este último deja la CPU y permite que el primero ejecute otra.

La llamada de sistema que nos permite realizar lo anterior es sched_yield(). El proceso y/o thread que ejecuta dicha llamada interrumpe su ejecución y se coloca a la cabeza de la lista de procesos listos. En el momento en que el proceso interrumpe su ejecución, la CPU le es asignada al proceso que estaba a la cabeza de la lista de procesos listos. En threads esto ocurre hasta que threads con la misma o mayor prioridad terminan de ejecutar o son bloqueados. Consideremos que un thread ejecuta la función f1() y que otro thread ejecuta f2(), tal como se muestra en el ejemplo 5, thread5.c:

```
/* Ejemplo 5: thread5.c */
#include <pthread.h>
#include <sched.h>
#define SUMSIZE 5

void *f1()
{
      int i;
      for (i = 1; i <= SUMSIZE; i++)
      {
            printf("Soy el thread %d con i: %d \n",pthread_self(),i);
            sched_yield();

      }
      return NULL;
}

void *f2()
{
      int i;
      for (i = 1; i <= SUMSIZE; i++)
      {
            printf("\t Soy el thread %d con i: %d \n",pthread_self(),i);
            sched_yield();
      }
      return NULL;
}

main()
{
      pthread_t thd1, thd2;

      printf("\nEJEMPLO EJECUCION EN TANDEM \n\n");

      pthread_create(&thd1, NULL,(void *)f1, NULL);
      pthread_create(&thd2, NULL,(void *)f2, NULL);

      pthread_join(thd1, NULL);
      pthread_join(thd2, NULL);

      printf("\nFIN DEL EJEMPLO \n");
      exit(0);
}
```

Cada iteración dentro del ciclo definido en las funciones será ejecutada por un thread a la vez. Esta

forma de funcionamiento puede fallar solamente en el principio de la ejecución de los threads, se deja al lector la tarea de justificar esta “falla” y la forma de solucionarla, si es posible. La sintaxis de la llamada a sched_yiel() es:

```
#include <sched.h>

int sched_yield(void);
```

Como se puede constatar, no recibe parámetro alguno. La llamada regresa 0 en caso de éxito y regresa -1 si hubo algún error. La salida del código anterior se muestra a continuación:

```
$ ./thread5

EJEMPLO EJECUCION EN TANDEM
Soy el thread 4 con i: 1
Soy el thread 5 con i: 1
Soy el thread 4 con i: 2
Soy el thread 5 con i: 2
Soy el thread 4 con i: 3
Soy el thread 5 con i: 3
Soy el thread 4 con i: 4
Soy el thread 5 con i: 4
Soy el thread 4 con i: 5
Soy el thread 5 con i: 5
FIN DEL EJEMPLO
```

IV. 6. Inicializando la Ejecución de Threads

La librería POSIX provee una serie de funciones para inicializar threads o atributos de los threads con valores diferentes a los de defecto. En todos los casos, estos cambios pueden estar restringidos al tipo de proceso (usuario y permisos del usuario, específicamente) y/o de la implementación de pthreads, a menos que se indique explícitamente lo contrario (en este caso, se podrían modificar los atributos sin restricción).

Las funciones pthread_attr_getscope() y pthread_attr_setcope() permiten conocer y cambiar la propiedad de alcance (scope). Este atributo decide si el thread puede competir por los recursos dentro del proceso (threads tipo usuario), o si puede competir por los procesos a nivel sistema (threads tipo núcleo). Esto guarda relación con el ámbito en donde será planificada la ejecución del hilo. Los valores asociados al alcance son PTHREAD_SCOPE_PROCESS para los recursos locales al proceso y PTHREAD_SCOPE_SYSTEM para los recursos a nivel sistema.

El atributo que controla si los parámetros de planificación o scheduling se heredan o no del thread creador son pthread_attr_getinheritsched() para consultar el valor y pthread_attr_setinheritsched() para modificarlo. Los posibles valores son PTHREAD_INHERIT_SCHED para que los parámetros se hereden, o PTHREAD_EXPLICIT_SCHED para que sean especificarlos explícitamente.

Mucho de lo relacionado con la política de planificación de threads se encuentra definido dentro del mismo archivo de encabezado pthread.h, relacionado con los atributos de los threads, tal como se lo describe en una de las secciones anteriores. Si se tienen parámetros dentro de una política en particular, éstos también son accesibles tanto para el administrador de threads (que es parte del sistema operativo o de la biblioteca de threads) como para el proceso que crea el thread y el propio thread, que puede conocer la política y los parámetros de la misma pero no siempre puede cambiar estos valores. Sin embargo, se tienen múltiples formas de consultar tanto la política como los parámetros involucrados, y muchas de estas formas son reduntantes (*hacen lo mismo*). En general, las políticas de administración de CPU para los threads incluyen:

- SCHED_FIFO (tiempo real), First-In-First-Out; los threads administrados bajo esta política, si no son desplazados por una prioridad más grande o interrumpidos por una señal, seguirán hasta terminar.
- SCHED_RR (tiempo real), Round-Robin; si los threads no son desplazados por una prioridad

más grande o interrumpidos por una señal, continuarán su ejecución durante un período de tiempo.

- SCHED_OTHER (tiempo-compartido), definido por la implementación.

Que, según el estándar, deberían estar definidas en sched.h. La implementación LinuxThreads, por ejemplo, define SCHED_OTHER como la política por *default* para la administración de threads. Otras de las definiciones que se deben tener en sched.h es la de la estructura sched_param, que debe incluir, como mínimo, el campo sched_priority:

```
struct sched_param
{
        int sched_priority;        /* prioridad del proceso */
        …
};
```

En Solaris, por ejemplo, se tienen más campos, asociados a valores que son utilizados por el administrador de tareas, más específicamente en el manejo de las prioridades del proceso o thread que se administra.

```
struct sched_param
{
        int sched_priority;        /* prioridad del proceso */
        int sched_nicelim;         /* valor limite para nice en la politica SCHED_OTHER */
        int sched_nice;            /* valor de nice para politica SCHED_OTHER */
        int sched_pad[6];          /* ajuste/relleno de parametros */
};
```

En todos los casos, se recomienda verificar los valores específicos del sistema operativo y/o implementación de pthread en particular. En general, es suficiente con analizar las *man pages* que corresponden. En el sexto ejemplo, thread6.c, se utilizan las funciones

```
pthread_attr_init(&my_tattr))
```

con la que se obtienen los atributos del thread correspondiente al main(),

```
pthread_attr_getschedpolicy(&my_tattr, &policy);
```

con la que se obtiene la política aplicada por el administrador de CPU para este thread,

```
pthread_attr_getschedparam(&my_tattr, &param);
```

con la que se obtienen los parámetros de administración de CPU para este thread,

```
retval = pthread_attr_setschedparam(&my_tattr, &param);
```

con la que se intenta cambiar en my_tattr los parámetros de administración de CPU, dado que antes de esta llamada está la asignación

```
param.sched_priority = ALTA_PRIORIDAD;
```

donde se cambia el valor de la prioridad para la administración de CPU. Se debe notar que la función pthread_attr_setschedparam(&my_tattr, &param) no afecta al thread que hace la llamada sino que afecta solamente al contenido de my_tattr. Si pthread_attr_setschedparam(&my_tattr, &param) se ejecuta satisfactoriamente y luego estos atributos (my_tattr) son usados para la creación de un thread, entonces este thread creado tendrá la prioridad asignada en param.sched_priority. Un thread puede intentar el cambio de su propia política de administración y/o prioridad con la función

```
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *param);
```

De hecho, puede intentarlo con cualquier thread siempre que tenga su tid. También se puede usar la

función

```
int pthread_setschedprio(pthread_t thread, int prio);
```

Siguiento con el ejemplo, finalmente se realiza la llamada para la creación de un nuevo thread, con los atributos resultantes de las llamadas a funciones de biblioteca previamente explicadas.

```
/* Ejemplo 6: thread6.c */
#include <stdio.h>
#include <pthread.h>
#include <sched.h>

#define ALTA_PRIORIDAD 10

pthread_attr_t     my_tattr;
pthread_t          my_tid;
struct sched_param param;

void *haz_algo(int arg)
{
        printf("Haciendo algo (%d)... \n", arg);
}

main()
{
        int retval;
        int policy;
        int par;

        if (pthread_attr_init(&my_tattr))
        {
                printf("No se pudieron inicializar los atributos del objeto\n");
                exit(1);
        }

        pthread_attr_getschedpolicy(&my_tattr, &policy);
        if (policy == SCHED_OTHER)
                printf("Administracion SCHED_OTHER\n");

        retval = pthread_attr_getschedparam(&my_tattr, &param);

        if (retval != 0)
        {
                printf("No se pudieron conocer los parametros de planificacion, retval = %d
                exit(1);
        }

        param.sched_priority = ALTA_PRIORIDAD;
        retval = pthread_attr_setschedparam(&my_tattr, &param);
        if (retval != 0)
                printf("No se pudieron cambiar los parametros, retval = %d\n", retval);

        par = 1;
        if (pthread_create(&my_tid, &my_tattr, (void *)haz_algo, (void *)par))
                printf("No se pudo crear el thread\n");
        else if ( pthread_join (my_tid, NULL ) )
                printf("Error en el join del thread\n");
        exit(0);
}
```

El código de todos los programas se repiten en los anexos del final, para tener un espacio unificado de presentación de código fuente.

V. Bibliografía

- IEEE Std 1003.1, 2003 Edition, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 6, Copyright (C) 2001-2003 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. Available on line at http://www.opengroup.org/unix/online.html
- R. Gómez Cárdenas, Introducción a los threads, ITESM-CEM, Dpto. Ciencias Computacionales, disponible en http://webdia.cem.itesm.mx/ac/rogomez/Apuntes/artiThreads.pdf
- Sun Microsystems, Inc., http://cvs.opensolaris.org/source/xref/on/usr/src/head/sched.h
- A. Tripathi, Lecture Notes 6, Concurrent Programming using POSIX Threads, Introduction to Operating Systems, disponible en http://www-users.itlabs.umn.edu/classes/Fall-2004/csci5103/ClassNotes/LectureNotes-6/6-on-1.pdf

Anexo 1: thread1.c

```c
/* Ejemplo 1: thread1.c */
#include <pthread.h>     /* funciones pthread_ */
#include <stdlib.h>
#include <unistd.h>      /* sleep() */

void *thread_function(void *arg)
{
      printf("El nuevo hilo te saluda!\n");
      sleep(5);
      printf("El nuevo hilo se despide!\n");
      return NULL;
}

int main(void)
{
      pthread_t mythread;
      printf("El hilo original te saluda!\n");
      if ( pthread_create( &mythread, NULL, thread_function, NULL))
      {
            printf("error en crear thread");
            abort();
      }
      if ( pthread_join (mythread,NULL))
      {
            printf("error en el join del thread");
            abort();
      }
      printf("El hilo original se despide!\n");
      exit(0);
}
```

Anexo 2: thread2.c

```c
/* Ejemplo 2: thread2.c */
#include <pthread.h>   /* funciones pthread_ */
#include <stdlib.h>
#include <unistd.h>    /* sleep() */

int myglobal;

void *thread_function(void *arg)
{
      int i,j;
      for ( i=0; i<4; i++ )
      {
            j=myglobal; j=j+1;
            printf("Hilo Nuevo...\n");
            sleep(1);
            myglobal=j;
      }
      return NULL;
}

int main(void)
{
      pthread_t mythread; int i;

      if (pthread_create(&mythread, NULL, thread_function, NULL))
      {
            printf("error creating thread.");
            abort();
      }
      for ( i=0; i<4; i++)
      {
            myglobal=myglobal+1;
            printf("Hilo Principal...\n");
            sleep(1);
      }
      if ( pthread_join ( mythread, NULL ) )
      {
            printf("error joining thread.");
            abort();
      }
      printf("\nmyglobal vale %d\n", myglobal);
      exit(0);
}
```

Anexo 3: thread3.c

```c
/* Ejemplo 3: thread3.c */
#include <pthread.h>   /* funciones pthread_ */
#include <stdlib.h>
#include <unistd.h>    /* sleep() */

int myglobal;

pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;

void *thread_function(void *arg)
{
      int i,j;
      for ( i=0; i<4; i++ )
      {
            pthread_mutex_lock(&mymutex);
            j=myglobal; j=j+1;
            printf("Hilo Nuevo...\n");
            sleep(1);
            myglobal=j;
            pthread_mutex_unlock(&mymutex);
      }
      return NULL;
}

int main(void)
{
      pthread_t mythread;
      int i;

      if (pthread_create(&mythread, NULL, thread_function, NULL))
      {
            printf("error creating thread.");
            abort();
      }
      for ( i=0; i<4; i++)
      {
            pthread_mutex_lock(&mymutex);
            myglobal=myglobal+1;
            pthread_mutex_unlock(&mymutex);
            printf("Hilo Principal...\n");
            sleep(1);
      }
      if ( pthread_join ( mythread, NULL ) )
      {
            printf("error joining thread.");
            abort();
      }
      printf("\nmyglobal equals %d\n",myglobal);
      exit(0);
}
```

Anexo 4: thread4.c

```
/* Ejemplo 4: thread4.c */
#include <stdio.h>
#include <pthread.h>    /* pthread_  */
#include <stdlib.h>     /* srandom() */
#include <unistd.h>     /* sleep() */

int f1(int x)
{
        int id;
        id = pthread_self();
        printf("\t Nuevo hilo %d a dormir %d segundos \n", id, x);
        sleep(x);
        x = x/2;
        printf("\t Nuevo hilo se desperto y termino con status %d \n", x);
        pthread_exit((void *)x);
}

main()
{
        pthread_t t1;
        int tmp;
        int status;

        printf("Hilo principal crea un hilo...\n");
        srandom(time(NULL));
        tmp = 1 + random() % 3;
        pthread_create(&t1, NULL, (void *)f1, (void *)tmp);
        printf("Esperando que termine hilo creado...\n");
        pthread_join(t1, (void *)&status);
        printf("Hilo %d termino con status: %d \n", t1, status);
        exit(0);
}
```

Anexo 5: thread5.c

```c
/* Ejemplo 5: thread5.c */
#include <pthread.h>
#include <sched.h>
#define SUMSIZE 5

void *f1()
{
      int i;
      for (i = 1; i <= SUMSIZE; i++)
      {
            printf("Soy el thread %d con i: %d \n",pthread_self(),i);
            sched_yield();

      }
      return NULL;
}

void *f2()
{
      int i;
      for (i = 1; i <= SUMSIZE; i++)
      {
            printf("\t Soy el thread %d con i: %d \n",pthread_self(),i);
            sched_yield();
      }
      return NULL;
}

main()
{
      pthread_t thd1, thd2;

      printf("\nEJEMPLO EJECUCION EN TANDEM \n\n");

      pthread_create(&thd1, NULL,(void *)f1, NULL);
      pthread_create(&thd2, NULL,(void *)f2, NULL);

      pthread_join(thd1, NULL);
      pthread_join(thd2, NULL);

      printf("\nFIN DEL EJEMPLO \n");
      exit(0);
}
```

Anexo 6: thread6.c

```
/* Ejemplo 6: thread6.c */
#include <stdio.h>
#include <pthread.h>
#include <sched.h>

#define ALTA_PRIORIDAD 10

pthread_attr_t     my_tattr;
pthread_t          my_tid;
struct sched_param param;

void *haz_algo(int arg)
{
        printf("Haciendo algo (%d)... \n", arg);
}

main()
{
        int retval;
        int policy;
        int par;

        if (pthread_attr_init(&my_tattr))
        {
                printf("No se pudieron inicializar los atributos del objeto\n");
                exit(1);
        }

        pthread_attr_getschedpolicy(&my_tattr, &policy);
        if (policy == SCHED_OTHER)
                printf("Administracion SCHED_OTHER\n");

        retval = pthread_attr_getschedparam(&my_tattr, &param);

        if (retval != 0)
        {
                printf("No se pudieron conocer los parametros de planificacion, retval = %d
                exit(1);
        }

        param.sched_priority = ALTA_PRIORIDAD;
        retval = pthread_attr_setschedparam(&my_tattr, &param);
        if (retval != 0)
                printf("No se pudieron cambiar los parametros, retval = %d\n", retval);

        par = 1;
        if (pthread_create(&my_tid, &my_tattr, (void *)haz_algo, (void *)par))
                printf("No se pudo crear el thread\n");
        else if ( pthread_join (my_tid, NULL ) )
                printf("Error en el join del thread\n");
        exit(0);
}
```