Introducción a los threads

Dr. Roberto Gómez Cárdenas
ITESM-CEM, Dpto. Ciencias Computacionales
rogomez@campus.cem.itesm.mx
http://webdia.cem.itesm.mx/dia/ac/rogomez

September 19, 2000

*El presente documento representa un borrador de lo que será el capítulo de un libro. Acaba de ser terminado y debido a la urgencia de tiempo no se presto mucha atención a la forma, pero el fondo es muy bueno. Cualquier aclaración, comentario o correción se agradecera que se le notifique al autor vía personal o a través del correo electrónico, (*rogomez@campus.cem.itesm.mx*).*

Un proceso se puede definir como la entidad lógica que genera el CPU cuando ejecuta el código de un programa. Durante la ejecución del programa el CPU va siguiendo un cierto camino a través de las diferentes instrucciones que definen un programa. Este camino se conoce como thread de control o simplemente *thread*[1]. Un proceso tradicional Unix tiene un solo thread de control; sin embargo es posible definir varios threads, cada uno de los cuales puede ejecutar diferentes instrucciones al mismo tiempo (concurrentemente).

A todo proceso se le asigna un tiempo de atención conocido como *quantum*. Durante ese tiempo el proceso dispone del CPU para realizar todo lo que quiera. Sin embargo si una llamada de sistema bloquea al proceso este tendra que dejar el CPU, sin importar el quantum que le quede. Si se analiza la secuencia de instrucciones se puede deducir que un cierto número de ellas no necesitan ser ejecutadas secuencialmente. Por esperar la llegada de un evento, es posible que algunas instrucciones no se ejecuten hasta que dicho evento ocurra, independientemente de si las instrucciones esta relacionadas o no con el evento.

Los threads nos permiten contar con un paralelismo dentro de los procesos de tal forma que el proceso no esté obligado a regresar el control a causa de un bloqueo y puede aprovechar plenamente de su quantum, el cual va a comparir con los diferentes threads que residen dentro del proceso.

Esta perspectiva permite mejorar el desempeño particular de los procesos; pero también el desempeño global del sistema es mejorado debido a que el número de conmutaciones de procesos se ve reducido.

En este capítulo se describirán las principales características de los threads, su implementación, sincronización y las diferentes llamadas de sistema asociadas a ellos.

1 Conceptos básicos de threads

Un thread es una secuencia de intrucciones ejecutadas dentro del contexto de un proceso. Cuenta con su propio contador de programa (PC) así como con un stack para llevar un control de variables

[1] El termino thread aparece traducido en la literatura como hilo, sin embargo el autor prefirió respetar el nombre en inglés este no es traducido en todo el documento.

locales y direcciones de regreso en las llamadas a funciones. Los threads se ejecutan independiente y concurrentemente; comparten instrucciones del proceso y casi todos los datos. Esto último implica que si un thread realiza un cambio en los datos compartidos, este será percibido por otros threads en el mismo proceso.

Si dos threads de ejecución comparten un recurso en un marco de tiempo se debe tener cuidado de que no interfieran uno con otro. Sin embargo, en sistemas multiprocesadores se aprovecha mejor la concurrencia e interacción entre aplicaciones y dentro de las aplicaciones. Esto se debe al hecho de que es posible asignar un thread a cada procesador.

1.1 Ventajas y desventajas

Entre las principales ventajas de utilizar threads podemos mencionar las siguientes:

- Mejoramiento del tiempo de respuesta de las aplicaciones. En una interfaz hombre-máquina, como parte de una aplicación, no es necesario esperar el final de una operación para empezar otra.
- Explotación eficaz de las arquitecturas multiprocesadores de algunas máquinas.
- Mejoramiento de la estructura de los programas; al utilizar threads los programas serán estructurados en entidades de ejecución; lo cual mejora la legibilidad y el mantenimiento de éstos.
- Uso de menos recursos del sistema; el hecho de realizar paralelismo a nivel threads, y no a nivel procesos, provoca que el compartir datos y las comunicaciones no carguen al sistema.
- Uso eficaz de RPC's con threads; es posible utilizar un conjunto de estaciones de trabajo en red, como si se tratara de una máquina paralela de memoria compartida.

Sin embargo no todo es maravilla, si no se tiene cuidado en el manejo de threads se pueden encontrar algunas dificultades. Las desventajas que tiene el manejo de threads son:

- Debido a que los threads comparten gran información acerca del estado del proceso donde se crean, la ejecución concurrente de los threads puede afectar dicho estado de forma sorprendente. Programar threads requiere mayor cuidado que la programación de sistemas normales, ya que no existe una protección por parte del núcleo del sistema operativo.
- La idea de combinar concurrencia con el hecho de que los recursos están compartidos por todo el mundo, es relativamente nueva. Se debe practicar mucho y tener bastante cuidado para escribir aplicaciones multithreads para no generar información inconsistente.
- No es obvio extender la semántica de algunas instrucciones en ambientes multithreads, por ejemplo si un proceso ejecuta un `fork()` ¿qué contexto corre el `fork()` ?, es decir ¿donde empieza la ejecución del nuevo proceso?

1.2 Multiprogramación y multithread

Es posible contar con varios threads ejecutandose al mismo tiempo. Es necesario poder diferenciar diferentes escenarios que se pueden dar debido a esta *convivencia*; para ello usaremos los siguientes terminos:

Single-thread: el sistema cuenta con un solo thread.

Multithread: dentro del sistema es posible contar con dos o más threads.

Threads nivel usuario o aplicación: threads manejados por rutinas de bibliotecas en el espacio de direcciones del usuario.

Concurrencia: existe cuando dos threads están "conviviendo" al mismo tiempo.

Paralelismo: se da cuando al menos dos threads se están ejecutando simultáneamente.

Existe una cierta analogía entre los multithreads y la multiprogramación. Estas dos técnicas fueron introducidas para aprovechar el tiempo de espera de cálculo. Sin embargo, las cantidades de espera no tienen una medida común, varían de acuerdo a la granularidad de la aplicación. Esta es una de las grandes diferencias entre un thread y un proceso. Como un thread es de granularidad mucho más fina, este necesita un contexto de ejecución relativamente más ligero (que no maneja muchos recursos).

Otra diferencia es que la ejecución de un thread no puede ser interrumpida. Si este es el caso, sería necesario la implementación de un sistema de calendarización complejo (parecido al de los procesos). La consecuencia de esto sería un aumento en el costo de los cambios de contexto. Tomando en cuenta la granularidad fina de los threads y su corto tiempo de vida, esto sería inaceptable.

En procesos multithreads en un solo procesador el procesador puede alternar los recursos de ejecución de los procesos entre los threads. En este caso hablamos de una *ejecución concurrente*. El mismo proceso en multiprocesadores con memoria compartida cada thread se puede ejecutar en un procesador separado al mismo tiempo. Se dice que es una *ejecución paralela*.

Los ambientes multithreads permiten que varios threads residan dentro del contexto general de un solo proceso. La mayor parte de este proceso es compartido por todos los threads dentro del proceso. Esto incluye:

- El código, todos los threads de un proceso están ejecutando el mismo programa
- Todos los datos estáticos y globales (en un programa en C esto equivale a todas las variables declaradas fuera de las funciones)
- El heap (donde residen todas las variables dinámicas)
- Todos los atributos y recursos sostenidos por el kernel, incluyendo el identificador de los usuarios efectivos y reales, el directorio de trabajo y cualquier descriptor de archivos abierto.

Por otro lado, el contexto privado del thread es muy pequeño, pero principalmente incluye:

- Su propio registro de estado y stack (variables declaradas dentro de funciones)
- Su propio nivel de prioridad
- Su propio identificador

Aparte de esta lista, cada thread tiene asociado un conjunto de atributos los cuales serán explicados más adelante.

Los threads comparten mucho contexto del proceso, es responsabilidad del usuario asegurarse que los threads no interfieren uno con otro. No es posible que el usuario decida qué thread, dentro de un conjunto, entre a ejecución. Sin embargo funciones tal como `sched_yield()`[2] permiten que un thread ceda el control voluntariamente.

Algunos ejemplos de lo que puede ocurrir si no se tiene cuidado con el manejo de threads son:

[2] Sintaxis y uso serán vistos más adelante.

- Si un thread cambia de directorio; todos los threds dentro del proceso verán ese nuevo directorio.
- Si un decriptor de archivos es cerrado, el archivo es cerrado para todos los threads.
- Si un thread modifica datos en memoria compartida por otros threads, se deben de sincronizar de tal forma que otro thread no pueda corromper datos u obtener una vista inconsistente de los datos.
- La ejecución de la llamada `exit()` provoca que todo el proceso, con todos sus threads termine.
- 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, solo que todos los threads son destruidos.

2 Modelos de implementación de threads

En todo sistema computacional el usuario tiene acceso a cierto conjunto de datos, mientras que el núcleo del sistema operativo tiene acceso a sus estructuras de datos. 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 el usuario tiene acceso a todos los datos definidos por él dentro del proceso.

Sin embargo, ¿qué pasa con los threads? ¿quién los controla? Las respuestas dependerán del modelo de implementación del thread. Existen tres modelos, uno de los cuales es una combinación de los otros dos. A continuación se explican las características de cada uno de estos modelos.

2.1 Los threads núcleo

En este modelo los threads son vistos por el núcleo como una entidad programable. Cuando la aplicación genera sus threads éstos se convierten en recursos del núcleo. El programador se olvida de muchos de los detalles inherentes a los threads.

Una de las ventajas que presenta este modelo es que pueden aprovechar la existencia de diferentes procesadores en caso de que el sistema cuente con éstos. Una desventaja es que su programación puede ser tan costosa como la de los procesos mismos. Si un thread necesita un determinado recurso debe competir por él con el resto de los threads y procesos del sistema.

La figura 1 nos ofrece un esquema de lo que son los threads a nivel núcleo. Se presentan tres threads los cuales son ejecutados dentro del contexto de un mismo proceso. Estos tendrán que competir por recursos con todo objeto existente dentro del núcleo en ese momento.

2.2 Los threads usuario

Los threads usuario suelen ejecutarse sobre un sistema operativo existente, siendo invisibles al núcleo. Son manejados dentro del espacio de direcciones del thread y gracias a esto no presentan las desventanjas generadas por el cambio de contexto. Son "baratos" de crear, ya que residen en el mismo espacio de memoria donde se encuentra el proceso y no es necesario que se le asignen recursos extras a los que el proceso ya tenia asignado.

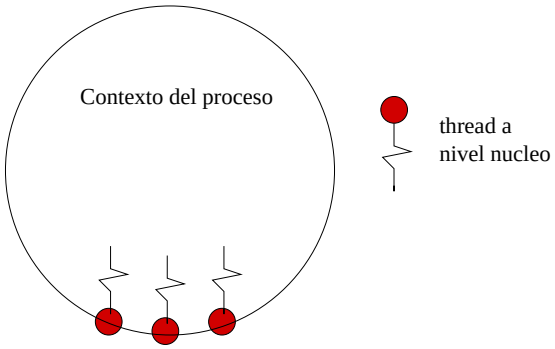

Figure 1: *Esquema de threads a nivel núcleo*

Una de las ventajas que presenta este modelo, es el hecho de que compiten por los recursos solo entre los threads que pertenceen al proceso, y no contra todos los threads y procesos del sistema operativo.

La figura 2 ejemplifica el concepto de un thread a nivel usuario. La figura muestra tres threads controlados por un usuario. Cada vez que solicitan un recurso se lo piden al núcleo el cual se encarga de conseguirlo.

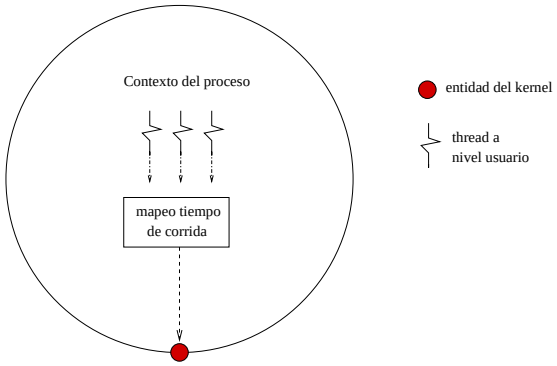

Figure 2: *Esquema de threads a nivel usuario*

2.3 Un esquema híbrido

Un esquema híbrido pretende combinar las ventajas de los modelos descritos anteriormente. En este caso se cuentan con dos niveles de programación, con una correspondencia entre los threads a nivel de usuario y las entidades del núcleo.

El usuario puede escribir programas con threads a nivel usuario y luego especificar el número de entidades programables por el núcleo y asociadas al proceso. En la ejecución se realiza la correspondencia entre threads a nivel usuario y los de nivel núcleo para lograr el paralelismo. El grado de control va a depender completamente de la implementación.

La figura 3 presenta un esquema en el cual tres threads son ejecutados dentro del mismo contexto. Dos de ellos siguen el modelo usuario, mientras que el otro el modelo núcleo.

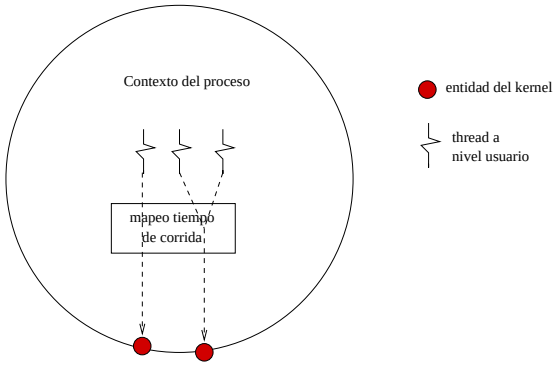

Figure 3: *Esquema de threads híbrido*

2.4 Los procesos ligeros y los threads

Solaris presenta un modelo propio de implementación de threads al cual le llama procesos ligeros o LWP por su nombre en inglés (ligthweigth processes). En Solaris las funciones utilizan threads de control llamados LWP. Un LWP puede considerarse y verse como una CPU virtual.

Todos los LWPs se encuentran agrupados dentro de una "piscina" de LWPs. Cada proceso cuenta con uno o más LWPs, cada uno de lo cualea interviene durante la ejecución un thread de tipo usuario. La creación de un thread no involucra la creación de un LWP.

Es importante aclarar que no existe una relación uno a uno entre los threads y los LWPs, los threads pueden emigrar de un LWP a otro. Cuando un thread usuario se bloquea, su LWP se transfiere a otro thread para que pueda ejecutarse. Sin embargo, si por alguna causa todos los LWPs están bloqueados, el sistema de threads va a crear y a añadir otro para atender al resto de los threads.

El sistema operativo es el encargado de la administración de la ejecución de los LWPs, sin importar el número de threads que residen dentro del LWP. Los threads dentro del LWP son administrados por el sistema de threads.

Cuando un thread tipo usuario realiza una llamada de sistema, el LWP llama al núcleo y queda atado al thread, al menos hasta que la llamada termine. Desde este punto de vista, el sistema distingue entre dos tipos de threads: threads acotados y threads no-acotados.

Los *threads no acotados* son calendarizados dentro de la piscina de los LWPs. Este tipo de threads pueden pasar de un LWP a otro. El sistema invoca a un LWP conforme se necesita y lo asigna a threads ejecutables. Si el thread se bloquea, o si otro thread necesita ser ejecutado, el estado del thread se almacena en la memoria del proceso y el sistema le asigna a otro thread el LWP.

Un *thread acotado* se puede atar un thread a un LWP, lo que trae como consecuencia que éste sea calendarizado globalmente, que cuente con una *máscara* de señales alterna[3] y con un temporizador (timer) propio. Esto implica que un thread sólo puede ejecutarse en el interior del LWP al que esta asociado y que, reciprocamente, éste LWP sólo puede servir a la ejecución de este thread.

Algunas veces el contar con más threads que LWPs es una desventaja, ya que el CPU pierde tiempo en el cambio de contexto. Un ejemplo de lo anterior se presenta cuando, en operaciones con matrices, se asigna cada renglón a un thread distinto. Si existe un LWP por renglón cada vez que se cambie de

[3] El terminoi mascara de señales será explicado en mayor detalle en secciones posteriores

renglón se tendrá que cambiar de LWP, lo cual podría ser tan "caro" en recursos, como un cambio de contexto entre dos procesos.

Es mejor contar con una mezcla balanceada de LWPs y de threads. Un ejemplo podría ser una aplicación a tiempo real, en la que algunos threads cuenten con prioridad a nivel sistema y calendarización a tiempo real, mientras que otros threads son necesarios para realizar cálculos en background. Un segundo ejemplo se presenta en un sistema de ventanas, en el que se cuenta con threads no acotados para la mayor parte de las operaciones y los threads acotados son usados para servicios con prioridad alta y atención a tiempo real como el ratón.

3 Las llamadas de sistema de los threads

Aunque todavía falta ver algunos aspectos de los threads antes de empezar a utilizarlos dentro de algún código, es importante decir algo acerca de las llamadas de sistema relacionadas con los threads. Existen dos implementaciones de la mayoría de estas llamadas de sistema.[4]. Se cuentan con dos librerías: POSIX y Solaris 2,[5] en la tabla 4 se muestran las diferentes llamadas de sistema para la creación y manipulación de threads. Como puede apreciarse la única diferencia entre las dos (a nivel nombre) es la `p` de POSIX.

El paquete típico de threads contiene un sistema *runtime* para el manejo de threads de forma transparente a los programadores/usuarios. Usualmente 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 algunas llamadas para el manejo de señales.

Descripción	**POSIX**	**Solaris 2**
Manejo thredas	`pthread_create`	`thread_create`
	`pthread_exit`	`thread_exit`
	`pthread_kill`	`thread_kill`
	`pthread_join`	`thread_join`
	`pthread_self`	`thread_self`
Exclusión mutua	`pthread_mutex_init`	`thread_mutex_init`
	`pthread_mutex_destory`	`thread_mutex_destory`
	`pthread_mutex_lock`	`thread_mutex_lock`
	`pthread_mutex_trylock`	`thread_mutex_trylock`
	`pthread_mutex_unlock`	`thread_mutex_unlock`
Variables condición	`pthread_cond_init`	`thread_cond_init`
	`pthread_cond_destroy`	`thread_cond_destroy`
	`pthread_cond_wait`	`thread_cond_wait`
	`pthread_cond_timedwait`	`thread_cond_timedwait`
	`pthread_cond_signal`	`thread_cond_signal`
	`pthread_cond_broadcast`	`thread_cond_broadcast`

Figure 4: *Lamadas de threads de POSIX y Sun Solaris 2*

Por motivos de portabilidad, en este documento se eligió trabajar con la versión POSIX de las llamadas relacionadas con threads.

[4] POSIX es el acrónimo de Portable Operating System Interface for uniX, el cual junta un conjunto de estandares ISO e IEEE para proporcionar portabilidad a los programas que se desarrollan en Unix. Como dato adicional, el estándar Posix.1 de threads fue aprobado en 1995.

[5] El concepto de threads fue desarrollado después de la salida de Solaris 1

4 Los atributos de un thread

Cada thread cuenta con diferentes propiedades que lo hacen único. Se adopta un enfoque orientado a objetos con respecto a la representación y asignación de propiedades. Bajo este paradigma cada thread cuenta con un objeto atributo asociado a varios threads. Los objetos atributos son del tipo:

```
pthread_attr_t
```

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 de la dirección:* longitud del stack del thread
- *Alcance (scope):* define quien controla la ejecución del thread: el proceso o el núcleo del sistema operativo
- *Herencia:* los parámetros de calendarización son heredados o definidos localmente
- *Política de calendarización:* la política que va a definir que proceso se va a ejecutar y en que instante.
  - FIFO
  - Round-robin
  - definido por la implementación
- *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.

La función `pthread_attr_init` asigna los atributos de default a los threads y la función `pthread_attr_destroy` hace que el valor del objeto atributo sea inválido.

Cada atributo de los hilos tiene una función asociada para cambiar su valor o para destruirlo. En la siguiente tabla se presentan las funciones que permiten manipular los diferentes atributos de un thread. Todas las funciones cuentan con dos parámetros. El primero de ellos es un apuntador al objeto atributo del thread, el segundo es el valor del atributo o un apuntador a un valor. Estos valores varían según el atributo. Por ejemplo, para modificar la política de calendarización se tendría que utilizar las siguientes funciones:

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

int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr.
                               const struct sched_param *param);
```

Los threads cuentan con un stack cuya ubicación y tamaño se pueden modificar a través de las llamadas `pthread_attr_getsatckaddr()` y `pthread_attr_getstacksize()` Con la primera es posible obtener la dirección de inicio y la segunda se usa para cambiarla.

Atributo	**Función**
Inicialización	`pthread_attr_init` `pthread_attr_destroy`
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 calendarización	`pthread_attr_setschedpolicy` `pthread_attr_getschedpolicy`
Parámetros calendarización	`pthread_attr_setschedparam` `pthread_attr_getschedparam`

Relación atributos y funciones para manejarlos

Una vez que un hilo se "desconecta" 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 establer que alguien puede esperar por el thread, y `PTHREAD_CREATE_DETACHED` para lo contrario. La opción por default es que los threads sean "alcanzables".

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). 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 calendarización 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_INHEIT_SCHED` para que los parámetros se hereden, o `PTHREAD_EXPLICIT_SCHED` para que sean especificados explícitamente.

Todo lo relacionado con la calendarización de eventos se encuentra definido dentro del archivo de encabezado `sched.h`. La política de calendarización se encuentra en la estructura `sched_param` que tiene los siguientes campos:

```
struct sched_param {
     int   sched_priority;  /* politica */
     int   sched_nicelim;   /* valor limite para politica SCHED_OTHER */
     int   sched_nice;     /* valor para politica  SCHED_OTHER */
     int   sched_pad[6];    /* ajuste de parametros */
};
```

El campo `sched_priority` establece la política de calendarización que el hilo debe seguir. Los posibles valores que puede tener este campo son:

- `SCHED_FIFO` (tiempo real), First-In-First-Out; los threads calendarizados bajo esta política, si no son desplazados por una prioridad más grande o interrumpidos por una señal, seguiran hasta terminar.
- `SCHED_RR` (tiempo real), Round-Robin; si los threads no son desplazados por una prioridad más grande o interrumpidos por una señal, continuaran su ejecución durante un periodo de tiempo.
- `SCHED_OTHER` (tiempo-compartido), definido por la implementación.

La implementación más común de `SCHED_OTHER` es una política de prioridad apropiativa. Una implementación que respete POSIX puede apoyar cualquiera de estas políticas de calendarización.

A manera de ejemplo de las funciones anteriores se presenta el siguiente código. El ejemplo crea un thread con los atributos por default le asigna la función `haz_algo` y luego cambia la prioridad del hilo a `ALTA_PRIORIDAD`

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <sched.h>

#define ALTA_PRIORIDAD 10

pthread_attr_t my_tattr;
pthread_t my_tid;
struct sched_param param;
int fd;

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

main()
{
  if (pthread_attr_init(&my_tattr))
     perror("No se pudieron inicializar los atributos del objeto");
  else
     if (pthread_attr_getschedparam(&my_tattr, &param))
       perror("No se pudieron conocer los parametros de calendarizacion");
     else {
       param.sched_priority = ALTA_PRIORIDAD;
       if (pthread_attr_setschedparam(&my_tattr, &param))
         perror("No se pudieron cambiar los parametros");
       else
         if (pthread_create(&my_tid, &my_tattr, haz_algo, (void *)&fd))
           perror("no se pudo crear el thread");
     }
}
```

El código anterior asocia el objeto atributo `my_attr` al thread `my_tid` durante su creación. Antes de esto la prioridad de calendarización fue cambiada.

5 Creando e inicializando 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 cree explicitamente nuevos threads, continuará como un thread simple y tradicional. La libreria de threads ofrece diferentes rutinas la creación e inicialización de estos. 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 todo salió bien 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 padre tomandolo del heap.

`rutina` se refiere a la función llamada por el thread cuando este comienza ejecución. Desde el punto de vista del thread esta función es equivalente a la función `main()` de un proceso tradicional. Si el thread regresa de la función el thread termina. Es necesario hacer un cast al tipo definido para no tener problemas en la compilación.

`arg`, representa los parámetros de la función. Generalmente se usa para permitir que varios threads, corriendo la misma función, para tener una identidad individual, o para asignar datos sobre el que van a ejecutar. Solo se puede pasar un argumento, en el caso de que se requieran más es posible pasarlos todos dentro de una estructura.

Los threads comparten el espacio de direcciones del proceso donde son creados. Un thread es dinámico si se puede crear en cualquier instante durante la ejecución del proceso y si no es necesario especificar por adelantado el número de threads.

Un ejemplo de creación de un thread se presenta a continuación:

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

main()
{
   pthread_t   tid;
   int fd;
   if (fd = open("toto",O_RDONLY)) == -1)
      perror("Imposible de abrir toto");
   else
```

```
      if ( pthread_create(&tid, NULL, procesa_fd, (void *)&fd) )
        perror("No se pudo crear el thread");
                    :
                    :
```

5.1 Identificación y terminación de un thread

Es posible identificar al thread a través de la llamada: `pthread_self()`. Dicha llamada regresa el identificador del thread que la llamo, no requiere de ningún parámetro y regresa un valor de tipo entero.

Un thread termina su vida cuando ejecuta la última instrucción de una instrucción, cuando ejecuta una `return` o cuando ejecuta un `exit()`. Sin embargo al ejecutar esto último el thread no sera el único quye termine, si no que provocará que todos los threads del proceso terminen. Existe una llamada de sistema que permite que el thread que la llame sea el único que termine su ejecución:

```
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 termino. Puede ser recuperado por otro thread a través de la llamada `pthread_join()` La sintaxis de esta última es:

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

Esta llamada suspende la ejecución de thread que la mando llamar, hasta que el thread indicado por el parámetro `thread` termine. El parámetro `value_ptr` permite recuperar el estatus de terminación del thread esperado y especificado por la llamada `pthread_exit()`.

Un ejemplo de uso de las llamadas anteriores se presenta en el siguiente código.

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

int f1(x)
  int x;
{
   int id;

   id = pthread_self();
   printf("\t Hilo %d a dormir %d segundos \n",id,x);
   sleep(x);
   x=x/2;
   printf("\t Ya me desperte 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 \n");
  pthread_join(t1, (void *)&status);
  printf("Hilo %d termino con status: %d \n",t1,status);
}
```

El código crea un thread el cual se encarga de ejecutar la función `f1()`, pasandole como parámetro un entero calculado de forma aleatoria. La función tiene por objeto dormir al thread durante un determinado tiempo (el parámetro de entrada).

Para poder compilar un programa que usa llamadas de sistema de threads tipo POSIX, es necesario compilarlo con la opción `-l pthread`. La compilacion y salida del código anterior es es:

```
rogomez@armagnac:184>gcc exjoin.c -lpthread -o exjoin
rogomez@armagnac:185>exjoin
Hilo principal crea un hilo
Esperando que termine
 Hilo 4 a dormir 3 segundos
 Ya me desperte y termino con status 1
Hilo 4 termino con status: 1
rogomez@armagnac:186>
```

5.2 Sugerencias para evitar errores comunes

Es recomendable que toda creación de un thread sea precedida por una llamada `pthread_join()`. La razón de esto es que no hay forma de asegurar que el thread creado va a terminar su ejecución antes del que lo creó. Si este es el caso, el thread principal terminará su ejecución (al llegar al final `"}"`) del procedimiento principal (`main()`) y todos los threads creados por él serán destruidos.

El siguiente código ejemplifica lo anterior:

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

void f(void *arg)
{
  int x;

  x=random() % 5;
  printf("Primer thread durmiendo %d sgs \n",arg);
  sleep(x);
  return((void)NULL);
}

main()
{

  pthread_t tid;
```

```
  pthread_create(&tid,  NULL,(void *)f, NULL);
  printf("Fin del programa \n");
}
```

La salida de la ejecución del código anterior se presenta a continuación:

```
rogomez@armagnac:584>gcc toto.c -lpthread -o toto
rogomez@armagnac:585>toto
Fin del programa
rogomez@armagnac:586>toto
Fin del programa
rogomez@armagnac:587>
```

Como puede apreciarse nunca despliega información alguna. Esto es debido a que el thread principal termina primero que el thread creado.

Ahora bien, hay que tomar en cuenta que el mal uso del `pthread_join` puede generar otros problemas. Uno de estos lo denominaremos *concurrencia serial* y consiste en crear threads que se ejecutan uno después del otro y no al mismo tiempo. Si uno de los threads presenta un ciclo infinito (i.e. un servidor) esto podría ocasionar que el resto de los threads no sean creados.

Consideremos el siguiente código:

```
  #include <pthread.h>
  #define MAX 10

  main()
  {
     pthread_t thd1;
               :
               :
     for (i=0; i<= MAX; i++) {
       pthread_create(&thd1, NULL,(void *)fx, NULL);
       pthread_join(thd1, NULL);
     }
               :
               :
 }
```

Si la función `fx()` tiene un ciclo infinito, el ciclo sólo tendrá una iteración. En efecto el thread principal espera que el thread creado termine para poder continuar con el ciclo. En vista de que el thread creado nunca termina el thread principal nunca podrá pasar a la siguiente iteración.

Aún si la función `fx()` no cuenta con un ciclo infinito, el código anterior no puede considerarse como correcto. El objetivo de la concurrencia es contar con dos o más threads ejecutandose *al mismo tiempo*. En el código anterior los threads son ejecutados uno después del otro, por lo que no hay concurrencia. El resultado es el mismo de llamar a la función `fx()` sin crear ningún thread, ni esperar su terminación. En efecto, el thread principal crea un thread, espera la terminación de este y regresa al ciclo, donde vuelve a crear otro y espera por su terminación.

Para poder crear un número `MAX` de threads que manden llamar a la función `fx()` con parámetro `i`, es posible utilizar el siguiente código. Se deja como ejercicio al lector al análisis de este.

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

```
#define MAX 10

fx(int x)
{
    :
}

main()
{
   pthread_t thd[MAX];
            :
            :
   for (i=0; i<= MAX; i++)
     pthread_create(&thd[i], NULL,(void *)fx, (void *)i);

   for (i=0; i<= MAX; i++)
      pthread_join(thd[i], NULL);

            :
            :
}
```

5.2.1 La llamada de sistema *sched_yield*()

Dentro de una aplición distribuida es díficil decir el momento en el que un thread va a entrar en ejecución. En algunas aplicaciones se necesita una ejecución *tandem* de las partes involucradas. Por tandem entendemos el hecho de que un thread ejecuta una instrucción para después dejar el CPU y permitir que otro thread ejecute también ejecute una instrucción. Después de ejecutar su instrucción, este último deja el 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, el CPU le es asignado 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.

La sintaxis de la llamada 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 algun error.

Un ejemplo de uso es el siguiente:

```
#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");
}
```

Consideremos que un thread ejecuta la función `f1()` y que otro thread ejecuta `f2()`. Cada iteración dentro del ciclo definido en las funciones será ejecutado por un thread a la vez. La salida del código anterior se muestra a continuación:

```
rogomez@armagnac:116>gcc tandem.c -lpthread -lrt -o tandem
rogomez@armagnac:117>tandem

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
rogomez@armagnac:118>
```

References

[And91] Concurrent Programming - *Andrews R. Gregory* The Benjamming/Cummings Publisihng Inc. 1991 - 1a. Edición

[Bro94] Brown Chris - *Unix Distributed Programming* Prentice Hall

[Rif96] Rifflet Jean-Marie - *UNIX 99 exercieses corrigés* Ediscience International 1996 1a. Edición

[RoRo97] Robbins Kay A. y Robbins Steven - *Unix Programación Práctica* Prentice Hall, 1997

[Ste90] Stevens Richard W. - *Unix Network Programming* 1992 Prentice Hall

[Ste92] Stevens Richard W. - *Advanced Programming in the Unix Environnenment* Addison-Wesley Professional Computing Series 1992 1a. Edición

[Zi96] Zignin Bernard- *Techniques du multithread (du parallelisme dans les processus)* Ed. Hermes, 1996

[Sh98] Shapley John - *Interprocess Communications in Unix, The nooks & Crannies* Ed. Prentice Hall, 2da. edición, 1998

[Sun94] Sun Microsystems Inc.- *Multithread Programming Guide*, Agosto 1994

[Tan93] Andrew S. Tanenbaum, *Sistema Operativos Modernos* Ed. Prentice Hall, 1993