Introducción a los Hilos en Linux

Horacio Goetendía Bonilla

12 de diciembre de 2003

Índice

1. Conceptos

1.1. Proceso

Un concepto fundamental en todos los sistemas operativos es el de proceso. Un proceso es sucintamente un programa en ejecución. Que está conformado de un programa ejecutable, sus datos, pila ,contador y otros registros. En un sistema operativo multitarea todos los procesos existentes se turnan para el uso del recurso más preciado en el computador; el CPU. Cada vez que el tiempo de uso del CPU termina para un proceso, este debe de volverse a inicializar en el mismo estado que se encontraba al detenerse, con lo cual inferimos que la información restaurada para el proceso debe de almacenarse en algún lugar.

En Linux la información relativa al proceso "suspendido"distinta del contenido de su propio espacio de dirección se almacena en una tabla; a esta tabla se le denomina *tabla de procesos*, la cual es una lista enlazada de estructuras para lo cual cada estructura es para cada proceso existente en ese instante.

Partes de un proceso suspendido:	
Espacio de dirección (Imagen)	Estructura en la Tabla de Procesos

1.2. Hilos

Los hilos son conocidos como procesos ligeros. aun que no son realmente procesos. Un hilo es esencialmente un contador de programa, una pila, y un conjunto de registros, el resto de estructuras de datos de la tabla de procesos pertenecen al proceso. Un proceso es modelado como una tarea con un simple hilo. Como los hilos son más pequeños comparados con los procesos, la creación de un hilo es relativamente menos pesada. Por ejemplo cuando un proceso sen intercala con otros para usar la CPU ese tiempo de cargar y descargar los datos de las respectivas estructuras de la tabla de procesos(tiempo de contexto) es mucho más pesada para un proceso que para un simple hilo. Es por eso que un hilo es considerado como un proceso liviano por que su cambio de contexto solo consta básicamente en cambiar un registro contador. Una manera gráfica de este concepto lo podemos observar en la Figura 1

Por lo tanto los hilos nos dan la posibilidad de escribir aplicaciones concurrentes que se pueden ejecutar tanto en sistemas monoprocesador como multiprocesador de forma transparente, obteniendo un aumento de rendimiento considerable cuando se encuentran disponibles procesadores adicionales. Además, los hilos pueden incrementar el rendimiento en un entorno monoprocesador cuando la aplicación realiza operaciones típicas de bloqueo o produce retrasos, como E/S de ficheros o sockets.

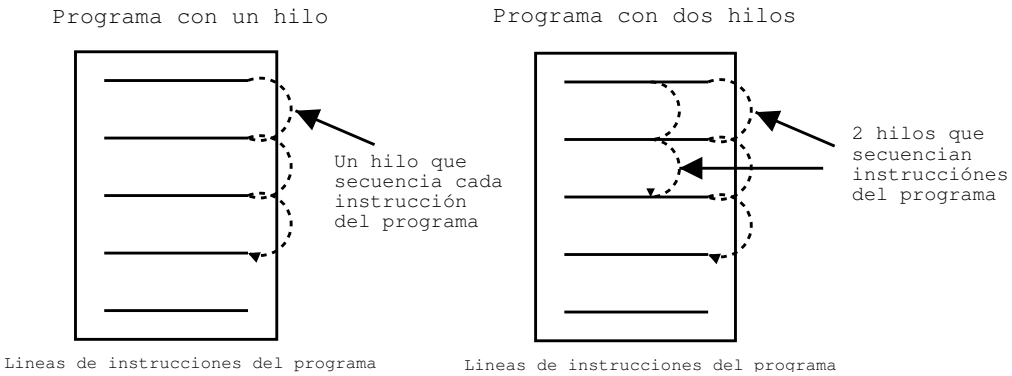

Figura 1: Los hilos en los programas.

2. Los hilos Posix

Las llamadas al sistema para la gestión básica de hilos POSIX son las siguientes:

```
Llamadas al sistema:
pthread_create
pthread_exit
pthread_kill
pthread_join
pthread_self
```

Los respectivos ficheros cabecera son:

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

La variable para identificar un hilo es (en este caso como ejemplo ponemos `tid`):

```
Variable de identificación del hilo:
pthread_t tid;
```

Para la creación de un hilo que siempre se asocia a una función (en este caso a mifuncion()) se usa:

```
Para la creación de un hilo:
pthread_create(&tid, NULL, mifuncion, (void *) misargs);

printf("Hilo creado. Esperando su finalizacion...");
fflush(stdout);
```

Una vez creado el hilo, la ejecución continúa de forma concurrente entre el procedimiento `main() y mifuncion()`. El primero espera a que `mifuncion()` termine llamando a `pthread_join()`:

Llamado a pthread_join:

```
pthread_join(tid, NULL);
printf("Hilo finalizado...");
fflush(stdout);
```

Esta es la función a la que se asocia el hilo. Para finalizar el hilo se invoca a `pthread_exit()`:

Función asociada al hilo:

```
void *mifuncion(void *arg)
{
printf("Soy mifuncion...");
printf(". Argumento 1:%d. Argumento 2:%d",*((int *)(arg)),*((int *)(arg)+1));
fflush(stdout);
printf("Saliendo de mifuncion...");
fflush(stdout);
pthread_exit(NULL);
}
```

Un ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void funcion();
int main (int argc, char *argv[])
{
    pthread_t tid; /* declaramos el hilo */
    int ret; /* el valor de retorno para gestionar
                el estado de la creación del hilo */
    ret = pthread_create (&tid, NULL, (void *) funcion, NULL);

    if (ret)
   { perror ("No se pudo crear el primer hilo");
    exit (EXIT_FAILURE);
   }

    pthread_join (pthrd1, NULL);
    exit (EXIT_SUCCESS);
}

void funcion ()
{    /* Ponle lo que quieras hacer aqui */
}
```

2.1. Como compilar un programa con pthreads

Para crear programas que hagan uso de la librería pthreads necesitamos, en primer lugar, la librería en sí. Esta viene en la mayoría de distribuciones Linux, y seguramente se instale al mismo tiempo que los paquetes incluidos para el desarrollo de aplicaciones (es decir, cuando instalamos la libc o algún paquete tipo libc-devel) Una vez tenemos la librería instalada, deberemos compilar el programa y linkearlo con la librería dependiendo del compilador que estemos usando.

```
Compilando programas:
gcc programa_con_pthreads.c -o programa_con_pthreads -lpthread
```

3. Problemas de concurrencia de los hilos

Como vimos trabajar con hilos es trabajar nativamente con programas concurrentes, uno de los mayores problemas con los que nos podremos encontrar, y que es tácito en la concurrencia, es el acceso a variables y/o estructuras compartidas o globales, es decir usar variables que son modificadas por otros, por ejemplo: Almacenamos un contador en x=3 y justo en ese instante acaba nuestro tiempo de CPU (x es una variable global) al volver a usar la CPU nos damos con la sorpresa del que ahora x vale 6, esto se debe a que en otro hilo se modificó el valor de x.

3.1. Modos de prevenir el problema de la concurrencia en los Hilos Posix

La librería de Pthreads nos ofrece unos mecanismos básicos pero muy útiles para definir esto. Estos mecanismos son los llamados semáforos binarios, y se usan para implementar las llamadas regiones críticas (RC) o zonas de exclusión mutua (ZE).

Y qué es una RC? Pues una parte de nuestro código que es susceptible de verse afectada por cosas como la del ejemplo de la variable x. Como regla general, siempre que haya variables o estructuras globales que vayan a ser accedidas por más de un hilo a la vez, el acceso a éstas deberá ser considerado una región crítica, y protegido con los medios que vamos a explicar a continuación. Incluso si estamos seguros que solo un hilo va a acceder a una determinada estructura, no sería mala idea meter ese código en una RC porque tal vez en un futuro ampliemos nuestro código y no recordemos que teníamos esos accesos por ahí escondidos, con el consiguiente riesgo de bugs que ello conlleva.

Lo que los hilos Posix nos ofrece son los semáforos binarios, semáforos mutex o simplemente mutexs. Un semáforo binario es una estructura de datos que actúa como un semáforo porque puede tener dos estados: o abierto o cerrado. Cuando el semáforo está abierto, al primer hilo que pide un bloqueo se le asigna ese bloqueo y no se deja pasar a nadie más por el

semáforo. Mientras que si el semáforo está cerrado, porque algún hilo ya tiene el bloqueo, el thread que lo pidió parará su ejecución hasta que no sea liberado el susodicho bloqueo.

Solo puede haber un solo hilo teniendo el bloqueo del semáforo, mientras que puede haber más de un hilo esperando para entrar en la RC, situados en la cola de espera del semáforo. Es decir, los threads se excluyen mutuamente (de ahí lo de mutex para el nombre) el uno al otro para entrar.

Pues con una cosa tan sencilla en concepto se implementan las RC: se pide el bloqueo del semáforo antes de entrar, éste es otorgado al primero que llega, mientras que los demás se quedan bloqueados esperando a que el que entró primero libere el bloqueo o exclusión. Una vez el que entró sale de la RC, éste debe notificarlo a la librería de pthreads para que mire si había algún otro thread esperando para entrar en la cola. Si lo había, le da el bloqueo al primero y deja que siga ejecutándose.

Las funciones que ofrece los hilos Posix para llevar esto a cabo son:

3.1.1. Función de inicialización de mutex

```
Función para evitar problemas que acarrea la concurrencia:
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

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

mutex: Es un puntero a un parámetro del tipo `pthread_mutex_t`, que es el tipo de datos que usa la librería Pthreads para controlar los mutex.

attr: Es un puntero a una estructura del tipo `pthread_mutexattr_t` y sirve para definir qué tipo de mutex queremos: normal, recursivo o errorcheck.

Si este valor es NULL (recomendado), la librería le asignará un valor por defecto.

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

3.1.2. Función de petición de bloqueo

```
Función de petición de bloqueo:
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Esta función pide el bloqueo para entrar en una RC. Si queremos implementar una RC, todos los thread tendrán que pedir el bloqueo sobre el mismo semáforo.

mutex: Es un puntero al mutex sobre el cual queremos pedir el bloqueo o sobre el que nos bloquearemos en caso de que ya haya alguien dentro de la RC.

Como resultado, devuelve 0 si no hubo error, o diferente de 0 si lo hubo.

3.1.3. Función de liberación de bloqueo

Función de liberación de bloqueo
`int pthread_mutex_unlock(pthread_mutex_t *mutex)`

Esta es la función contraria a la anterior. Libera el bloqueo que tuviéramos sobre un semáforo.

mutex: Es el semáforo donde tenemos el bloqueo y queremos liberarlo.

Retorna 0 como resultado si no hubo error o diferente de 0 si lo hubo.

3.1.4. Función de liberación de memoria

Función de liberación de memoria
`int pthread_mutex_destroy(pthread_mutex_t *mutex)`

Esta función le dice a la librería que el mutex que el estamos indicando no lo vamos a usar más, y que puede liberar toda la memoria ocupada en sus estructuras internas por ese mutex.

mutex: El mutex que queremos destruir.

La función, como siempre, devuelve 0 si no hubo error, o distinto de 0 si lo hubo.

4. Ejemplos:

Si tenemos dos hilos de la siguiente manera:

Hilo 1

```
void *funcion_hilo_1(void *arg)
{
    int resultado;
    ...
    if (i == valor_cualquiera) {
         ...
         resultado = i * (int)*arg;
         ...
    }
     pthread_exit(&resultado);
}
```

Hilo 2

```
void *funcion_hilo_2(void *arg)
{     int otro_resultado;
    ...
    if (funcion_sobre_arg(*arg) == 0) {
        ...
        i = *arg;
        ...
    }
   pthread_exit(&otro_resultado);
}
```

Este código, que tiene la variable í'como global, aparentemente es inofensivo, pero nos puede traer muchos problemas si se ejecuta en paralelo y se dan ciertas condiciones. Supongamos que el hilo 1 se empieza a ejecutar antes que el hilo 2, y que casualmente se produce un cambio de contexto (el sistema operativo suspende la tarea actual y pasa a ejecutar la siguiente) justo después de la línea que dice `if (i==valor_cualquiera)`. La entrada en ese `if` se producirá si se cumple la condición, que suponemos que sí.

Pero justo en ese momento el sistema hace un cambio de contexto y pone a ejecutar al hilo2, que se ejecuta el tiempo suficiente como para ejecutar la línea `i = *arg`. Al poco rato hilo 2 deja de ejecutarse y vuelve a ejecutarse el hilo 1, pero, ¿qué valor tiene ahora i?. El que el hilo 1 está "suponiendo"que tiene (o sea, el mismo que comprobó al entrar en el `if` o el que le ha asignado el hilo 2. La respuesta es que `i` ha tomado el valor que le asignó hilo 2, con lo que el resultado que devolverá el hilo 1 después de sus cálculos será totalmente inválido e inesperado. Claro que todo esto puede que no pasara si el sistema tuviera muy pocos procesos en ese momento (con lo cual cada proceso se ejecutaría por más rato) y si el código del hilo 1 fuera lo suficientemente corto como para no sufrir ningún cambio de contexto en medio de su ejecución... Pero NUNCA deberemos hacer suposiciones de éstas, porque no sabremos dónde se van a ejecutar nuestros programas y siempre más vale prevenir.

El problema que tienen estos bugs es que son los más difíciles de detectar en el caso que no nos fijáramos en que podría pasar una cosa de estas el día que escribimos el código. Puede que a veces vaya todo a la perfección y que otras salga todo mal. A esto se le conoce por *Race Conditions* o Condiciones de Carrera porque según como vaya la cosa puede funcionar o no.

Las versiones corregidas mediante las funciones ya vistas sería:

Hilo 1 (Versión correcta)

```
void *funcion_hilo_1(void *arg)
{
    int resultado;
    ...
    pthread_mutex_lock(&mutex_acceso);
    if (i == valor_cualquiera) {
         ...
         resultado = i * (int)*arg;
         ...
    }
    pthread_mutex_unlock(&mutex_acceso);
     pthread_exit(&resultado);
}
```

Hilo 2 (Versión correcta)

```
void *funcion_hilo_2(void *arg)
{    int otro_resultado;
    ...
    if (funcion_sobre_arg(*arg) == 0) {
        ...
        pthread_mutex_lock(&mutex_acceso);
        i = *arg;
        pthread_mutex_unlock(&mutex_acceso);
        ...     }
   pthread_exit(&otro_resultado);
}
```

El main

```
int main(void)
{
     ...
     pthread_mutex_init(&mutex_acceso, NULL);
     ...
}
```

5. Algunos problemas

Lo visto anteriormente parece que soluciona completamente el problema de los accesos concurrentes, pero como buen sistémico que dice las soluciones de ahora son los problemas del mañana esto también nos puede traer más problemas.

Y los problemas aquí también son los *Deadlocks* (o Cerraduras Mortales) Los *deadlocks* se producen cuando un hilo se bloquea esperando un recurso que tiene bloqueado otro hilo que está esperando un recurso. Si el recurso para el segundo thread no llega nunca, no se desbloqueará nunca, con lo cual tampoco se desbloqueará nunca el primer thread. Lo que trae como resultado es que nuestro programa se bloquee.

La solución no es enteramente recibir alguna función para intentar prevenir (aunque los hilos Posix las posean) que esto se produzca, no hay ningún mecanismo fiable al 100 para prevenirlos.

Un deadlock circular..Hilo 1

```
void *funcion_hilo_1(void *arg)
{
    ...
    pthread_mutex_lock(&mutex_1);
    ...
    pthread_mutex_unlock(&mutex_2);
    ...
}
```

Un deadlock circular..Hilo 2

```
void *funcion_hilo_2(void *arg)
{
    ...
    pthread_mutex_lock(&mutex_2);
    ...
    pthread_mutex_unlock(&mutex_1);
    ...
}
```

5.1. Posibles soluciones al problema del deadlock

5.1.1. Semáforos recursivos

Estos semáforos solo aceptarán una sola petición de bloqueo por el mismo thread. Con los semáforos normales, si el mismo thread hace 10 llamadas a `pthread_mutex_lock` sobre el mismo semáforo, luego tendrá que hacer 10 llamadas a `pthread_mutex_unlock`, es decir, tantas como haya hecho a `pthread_mutex_lock`.

En cambio, los del tipo recursivo solo aceptarán una sola llamada a `pthread_mutex_lock`. Las siguientes llamadas serán ignoradas, con lo que ya eliminamos un tipo de deadlock.

Para poder crear un semáforo recursivo, tendremos que decírselo a `pthread_mutex_init`, indicándole como atributo el resultado de una llamada a `pthread_mutexattr_settype`.

El procedimiento es:

1. Definir una variable del tipo `pthread_mutexattr_t`: `pthread_mutexattr_t mutex_attr;`
2. Inicializarla con la llamada a `phtread_mutexattr_init`: `pthread_mutexattr_init(&mutex_attr);`

3. Indicarle el tipo explícitamente mediante `pthread_mutexattr_settype`: `pthread_mutexattr_settype(&mutex_attr, tipo);`

   Donde tipo puede ser `PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_DEFAULT (el que se usa por defecto), PTHREAD_MUTEX_RECURSIVE ó PTHREAD_MUTEX_ERRORCHECK`.

4. Probar antes de entrar:

   Si creemos que la siguiente llamada a `pthread_mutex_lock` va a ser bloqueante y que puede provocar un deadlock, la librería de Pthreads nos ofrece una función más para comprobar si eso es cierto: `pthread_mutex_trylock`.

Función de comprobación
`int pthread_mutex_trylock(pthread_mutex_t *mutex);`

   mutex: Es el mutex sobre el cual queremos realizar la prueba de bloqueo.

   La función devuelve EBUSY si el el thread llamante se bloqueará o 0 en caso contrario. Si no se produce el bloqueo, la función actúa igual que `phtread_mutex_lock`, adquiriendo el bloqueo sobre el semáforo.

Referencias

[1] Alex Shumacher: *Programación en POSIX Threads*
, 1987

[2] Sandra Loosemore : *The GNU C Library Reference Manual*
Free Software Foundation, 1999

[3] Mark Mitchell, Jeffrey Oldham, Alex Samuel : *Advanced Linux Programming*
New Riders, 2001

[4] Kay A. Robbins, Steven Robbins : *Unix Programación Práctica*
Prentice Hall, Primera Edición

[5] Kurt Wall : *Programación en Linux con ejemplos*
Prentice Hall, 2000