Programación en POSIX Threads (Pthreads)
Victor J. Sosa Sosa

1. Programación en POSIX Threads (Pthreads)

1.1. Introducción

La biblioteca de pthreads es una biblioteca que cumple los estándares POSIX y que nos permite trabajar con distintos hilos de ejecución (threads) al mismo tiempo.
La diferencia entre un thread y un proceso es que los procesos no comparten memoria entre sí, a no ser que se haya declarado explícitamente usando alguno de los mecanismos de IPC (InterProcess Communication) de Unix, mientras que los threads sí que comparten totalmente la memoria entre ellos. Además, para crear threads se usan las funciones de la biblioteca pthread o de cualquier otra que soporte threads mientras que para crear procesos usaremos la llamada al sistema fork(), que se encuentra en todos los sistemas unix.
Ya que pthreads es una biblioteca POSIX, se podrán portar los programas hechos con ella a cualquier sistema operativo POSIX que soporte threads. Ejemplos de ello son IRIX, los unix'es de BSD, Digital Unix OSF/1, etc.

1.2. 1.2. Como compilar un programa con pthreads

Para crear programas que hagan uso de la biblioteca pthreads necesitamos, en primer lugar, la biblioteca en sí. Ésta 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)
Si no es así, o usas un sistema que no sea linux, la biblioteca no debería ser muy difícil de encontrar en la red, porque es bastante conocida y se suele usar bastante.
Una vez tenemos la biblioteca instalada, deberemos compilar el programa y “linkarlo” con la biblioteca dependiendo del compilador que estemos usando.
La forma más usual de hacer esto es, si estamos usando como compilador GNU gcc con el comando:

```
gcc programa_con_pthreads.c -o programa_con_pthreads -lpthread
```

Si por el contrario no estamos usando el compilador de GNU, lo mejor será que miremos la página man del compilador de C instalado en el sistema. Por ejemplo, en el caso del compilador de Digital para OSF/1, éste tiene un parámetro especial para compilar con pthreads:

```
cc programa_con_pthreads.c -o programa_con_pthreads -pthread
```

Como verás la sintaxis no es muy diferente pero el compilador de Digital no aceptará la de gcc y viceversa.

1.3. 1.3. Creación y manipulación de threads

Para crear un thread nos valdremos de la función `pthread_create` de la biblioteca y de la estructura de datos pthread_t que identifica cada thread diferenciándolo de los demás y que contiene todos sus datos.
El prototipo de la función es el siguiente:

```
int   pthread_create(pthread_t   *   thread, pthread_attr_t *attr,
void * (*start_routine)(void *), void *arg)
```

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

- attr: Es un parámetro del tipo pthread_attr_t y que se debe inicializar previamente con los atributos que queramos que tenga el thread. Entre los atributos hay la prioridad, el quantum, el algoritmo de planificación que queramos usar, etc. Si pasamos como parámetro aquí NULL, la biblioteca le asignará al thread unos atributos por defecto (RECOMENDADO).
- start_routine: Aquí pondremos la dirección de la función que queremos que ejecute el thread. La función debe devolver un puntero genérico (void *) como resultado, y debe tener como único parámetro otro puntero genérico.
- La ventaja de que estos dos punteros sean genéricos es que podremos devolver cualquier cosa que se nos ocurra mediante los castings de tipos necesarios.
- Si necesitamos pasar o devolver más de un parámetro a la vez, se puede crear una estructura y meter allí dentro todo lo que necesitemos. Luego pasaremos o devolveremos la dirección de esta estructura como único parámetro. (ver código de ejemplo)
- arg: Es un puntero al parámetro que se le pasará a la función. Puede ser NULL si no queremos pasarle nada a la función.
- En caso de que todo haya ido bien, la función devuelve un 0 o un valor distinto de 0 en caso de que hubo algún error.

Una vez hemos llamado a esta función, ya tenemos a nuestro(s) thread(s) funcionando, pero ahora tenemos dos opciones: esperar a que terminen los threads, en el caso de que nos interese recoger algún resultado, o simplemente decirle a la biblioteca de pthreads que cuando termine la ejecución de la función del thread elimine todos sus datos de sus tablas internas.
Para ello, disponemos de dos funciones más de la biblioteca: `pthread_join` y `pthread_detach`.

```
int pthread_join(pthread_t th, void **thread_return)
```

- Esta función suspende el thread llamante hasta que no termine su ejecución el thread indicado por th. Además, una vez éste último termina, pone en thread_return el resultado devuelto por el thread que se estaba ejecutando.
- th: Es el identificador del thread que queremos esperar, y es el mismo que obtuvimos al crearlo con pthread_create.
- thread_return: Es un puntero a puntero que apunta (valga la redundancia) al resultado devuelto por el thread que estamos esperando cuando terminó su ejecución. Si este parámetro es NULL, le estamos indicando a la biblioteca que no nos importa el resultado.
- Devuelve 0 en caso de todo correcto, o valor diferente de 0 si hubo algún error.

```
int pthread_detach(pthread_t th)
```

- Esta función le indica a la biblioteca que no queremos que nos guarde el resultado de la ejecución del thread indicado por th. Por defecto la biblioteca guarda el resultado de ejecución de todos los threads hasta que nosotros hacemos un pthread_join para recoger el resultado.
- Es por eso que si no nos interesa el resultado de los threads tenemos que indicarlo con esta función. Así una vez que el thread haya terminado la biblioteca eliminará los datos del thread de sus tablas internas y tendremos más espacio disponible para crear otros threads (IMPORTANTE)
- th: Es el identificador del thread
- Devuelve 0 en caso de que todo haya ido bien o diferente de 0 si hubo error.

Hasta ahora hemos estado hablando de devolver valores cuando un thread finaliza, pero aún no hemos dicho como se hace. Pues bien, para ello tenemos la función `pthread_exit`

`void pthread_exit(void *retval)`

- Esta función termina la ejecución del thread que la llama.
- retval: Es un puntero genérico a los datos que queremos devolver como resultado. Estos datos serán recogidos más tarde cuando alguien haga un pthread_join con nuestro identificador de thread.
- No devuelve ningún valor.

Con todo lo que hemos visto hasta ahora ya estamos preparados para hacer nuestro primer programa con pthreads. El programa de ejemplo creará MAX_THREADS threads que ejecutarán la función funcion_thr.
Esta función sacará un mensaje por pantalla del tipo "hola, soy el thread número x", donde x será un número diferente para cada thread.
Para pasar esos parámetros a la función usaremos un struct del C, donde meteremos la cadena que debe imprimir cada thread más su identificador. (La cadena la podríamos haber metido directamente dentro de la función, pero así veremos como se pasa más de un parámetro al thread)
Una vez termina su ejecución, el thread devolverá como resultado su identificador (codificado en un entero), que será imprimido por pantalla por el thread padre que esperará que todos los threads terminen.

```
/*** <CORTAR AQUI > ***/
/*** Archivo ej1.c *****/
/* Creamos MAX_THREAD threads que sacan por pantalla una cadena y su
   identificador
   Una vez terminan su ejecución devuelven como resultado su
identificador */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_THREADS     10
// tabla con los identificadores de los threads
pthread_t tabla_thr[MAX_THREADS];
// tipo de datos y tabla con los parametros
typedef struct {
  int id;
  char *cadena;
} thr_param_t;
thr_param_t param[MAX_THREADS];
// tenemos que crear una tabla para los parámetros porque los pasamos
por
// referencia. Así, si solo tuviéramos una variable para los
parámetros
// al modificar esta modificaríamos todas las que habíamos pasado
anteriormente
// porque los threads no se quedan con el valor sino con la dirección
void *funcion_thr(thr_param_t *p)
{
  // Esta es la funcion que ejecutan los threads
  // como veras, no tiene mucho secreto...
  printf("%s %d\n", p->cadena, p->id);
  // una vez terminamos, devolvemos el valor
  pthread_exit(&(p->id));
}
```

```
int main(void)
{
  int i, *res;
  // creamos los threads
  printf("Creando threads...\n");
  for (i=0; i<MAX_THREADS; i++) {
    param[i].cadena = strdup("Hola, soy el thread");
    param[i].id     = i;
    pthread_create(&tabla_thr[i], NULL, (void *)&funcion_thr,
     (void *)&param[i]);
  }
  // esperamos que terminen los threads
  printf("Threads creados. Esperando que terminen...\n");
  for (i=0; i<MAX_THREADS; i++) {
    pthread_join(tabla_thr[i], (void *)&res);
    printf("El thread %d devolvio el valor %d\n", i, *res);
  }
  // sacamos el mensajito y salimos del programa
  printf("Todos los threads finalizados. Adios!\n");
  return 0;
}
/*** <CORTAR AQUI> ***/
```

Para compilar (bajo Linux y gcc): `gcc ej1.c -o ej1 -lpthread`

El ejemplo en sí es MUY tonto, pero es el esquema básico que siguen todas las aplicaciones que lanzan threads para realizar un cálculo y esperan su resultado:

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

Esto es un ejemplo de lo que se llama paralelismo estructurado.
Un ejemplo de un programa que use la función `pthread_detach` podría ser el de un servidor (de cualquier cosa: de correo, de http, de ftp, etc) que cree un hilo para cada petición que reciba. Como que no nos interesa el resultado de la ejecución, una vez hayamos creamos el thread llamaremos la funcion `pthread_detach`.
Esto es lo que se conoce por paralelismo no estructurado. Es decir, nuestros programas no siguen una estructura concreta sino que se van ramificando según nuestras necesidades.

1.4. Otras funciones útiles de la biblioteca pthreads

Hasta ahora hemos visto las funciones más básicas para tratar con pthreads, pero aún queda alguna otra función útil:
`pthread_t pthread_self(void)`

- Esta función devuelve al thread que la llama su información, en forma de variable del tipo pthread_t.

Es útil si el propio thread que se está ejecutando quiere cambiar sus atributos, hacerse él mismo un `pthread_detach`, etc.

- Devuelve el identificador del thread. Ejemplo:

```
#include <pthread.h>
...
void *funcion_threads(void *param)
{
   pthread_t yo_mismo;
   ...
   /* nosotros mismos nos hacemos el detach */
   yo_mismo = pthread_self();
   pthread_detach(yo_mismo);
```

```
    ...
}
int main(void)
{
   ...
}
int pthread_kill(pthread_t thread, int signo)
```

- Envía un signal especificada al thread especificada. Un signal útil de enviar puede ser el SIGKILL, o alguno de los definidos por el usuario, SIGUSR1 y SIGUSR2.
- Aunque pueda parecer útil a primera vista, la única utilidad que tiene es matar un thread desde el proceso padre. Si se quiere usar con fines de sincronización hay formas mejores de hacerlo tratándose de threads: mediante semáforos y variables de condición (enseguida lo veremos)
- thread: identifica el thread al cual le queremos enviar el signal.
- signo: número de la señal que queremos enviar al thread. Podemos usar las constantes definidas en /usr/include/signal.h
- Devuelve 0 si no hubo error, o diferente de 0 si lo hubo.

Hasta aquí la primera parte del tutorial de programación en Pthreads. Aquí hemos visto las funciones básicas que nos ofrece la biblioteca para la creación, manipulación y eliminación de threads, pero aún nos quedan algunas cosas por ver.

2. 2. Problemas de concurrencia con Pthreads

2.1. 2.1. Introducción

Cuando decidimos trabajar con programas concurrentes uno de los mayores problemas con los que nos podremos encontrar, y que es inherente a la concurrencia, es el acceso a variables y/o estructuras compartidas o globales
Esto se entenderá mejor con un ejemplo:

```
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 'i' 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 fácil... ;-) 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 pasará 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 en cristiano, Condiciones de Carrera) porque según como vaya la cosa puede funcionar o no.

2.2. 2.2. Mecanismos de Pthreads para prevenir esto

La biblioteca 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 mútua (ZE).
Y qué es una RC? Pues una parte de nuestro código que es susceptible de verse afectada por cosas como la del ejemplo. Como regla general, SIEMPRE que haya variables o estructuras globales que vayan a ser accedidas por más de un thread a la vez, el acceso a éstas deberá ser considerado una región crítica, y 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 Pthreads nos ofrece son los semáforos binarios, semáforos mutex o simplemente mutexs, como cada uno quiera llamarlo ;-) 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 thread 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 thread 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 thread poseyendo el bloqueo del semáforo, mientras que puede haber más de un thread esperando para entrar en la RC, encolados en la cola de espera del semáforo. Es decir, los threads se excluyen mútuamente (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 biblioteca 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 Pthreads para llevar esto a cabo son:

```
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 biblioteca 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 (esto se verá más adelante)
- Si este valor es NULL (recomendado), la biblioteca le asignará un valor por defecto.
- La función devuelve 0 si se pudo crear el mutex o -1 si hubo algún error.

`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.

`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.

`int pthread_mutex_destroy(pthread_mutex_t *mutex)`

- Le dice a la biblioteca que el mutex que le 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.

Estas son las funciones más básicas. Ahora, reescribiremos el pseudocódigo del ejemplo anterior con lo que hemos visto hasta ahora.

```
Variables globales:

pthread_mutex_t mutex_acceso;
int i;
```

```
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;
         ... <
    }
```

```
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_mutex_unlock(&mutex_acceso);
     pthread_exit(&resultado);
}
```

```
    }
   pthread_exit(&otro_resultado);
}
```

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

En color azul han sido añadidas las líneas que antes no estaban.
Como se puede ver lo único que hay que hacer es inicializar el semáforo, pedir el bloqueo antes de las RC y liberarlo después de salir de la RC, aunque a veces es cuestión también de tener un poco de vista.
Contra más pequeñas hagamos las RC, más concurrentes serán nuestros programas, porque tendrán que esperar menos tiempo en el caso de que haya bloqueos.

2.3. 2.3. Problemas... ¿Más problemas?

Aunque esto realmente soluciona el problema de los accesos concurrentes, también nos puede traer más problemas.
Y los problemas aquí también tienen nombre: los Deadlocks (o Abrazos 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.
Resultado: nuestro fantástico programa bloqueado.
Solución? Pues aunque la biblioteca de Pthreads nos de algún mecanismo para intentar prevenir que esto se produzca, no hay ningún mecanismo fiable al 100% para prevenirlos.
El modelo más sencillo de Deadlock es el circular:

```
Hilo 1
void *funcion_hilo_1(void *arg)
{
    ...
    pthread_mutex_lock(&mutex_1);
    ...

pthread_mutex_unlock(&mutex_2);
    ...
}
```

```
Hilo 2
void *funcion_hilo_2(void *arg)
{
    ...
    pthread_mutex_lock(&mutex_2);
    ...

pthread_mutex_unlock(&mutex_1);
    ...
}
```

Parece un poco raro, pero puede llegarse a producir.
Mecanismos que ofrece la biblioteca Pthreads:

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:

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

2. 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 biblioteca de Pthreads nos ofrece una función más para comprobar si eso es cierto: `pthread_mutex_trylock`.

```
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 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.

3. Funciones avanzadas

Otras funciones de uso avanzado con pthreads son las siguientes:

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_attr_setdetachstate(pthread_attr_t *attr, int
detachstate);

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int
*detachstate);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int
policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int
*policy);

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

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

Su utilización va más allá de los objetivos del curso y su estudio se deja al alumno