9.5 Threads

En este tema veremos cómo usar múltiples hilos de control (threads) para realizar varias tareas dentro del ambiente de un solo proceso. Todos los hilos que crea un proceso tienen acceso a los recursos de este, como los descriptores de archivo y memoria.

9.5.1 Concepto de threads

Un típico proceso de UNIX puede ser visto como un solo hilo de control: cada proceso es haciendo sólo una cosa a la vez. Con múltiples hilos de control, nosotros podemos diseñar nuestros programas para hacer más de una cosa a la vez dentro de un solo proceso, con cada hilo manejando una tarea separada. Esta técnica tiene muchos beneficios:

- Podemos simplificar el código que trabaja con eventos asincrónicos asignados un hilo separado para manejar un evento determinado. Cada hilo puede manejar este evento usando un esquema de programación síncrono. Un esquema de programa síncrono es mucho más simple que uno asíncrono.
- Múltiples procesos usan mecanismos complejos proveídos por el sistema operativo para compartir memoria y descriptores de archivos. Los threads, por otro lado, automáticamente tienen acceso a los mismos espacios de memoria y descriptores de archivo.
- Algunos problemas pueden ser divididos de tal forma que el programa final puede ser mejorado. Un proceso sencillo que tiene múltiples tareas a realizar implícitamente las serializa, porque solo hay un hilo de control. Con varios hilos de control, el proceso de tareas independientes puede ser entrelazadas asignado una tarea separada a cada thread. Dos tareas pueden ser entrelazadas solo si ellas no dependen entre sí.
- De manera similar, programas interactivos pueden mejorar significativamente su tiempo de respuesta usando threads para separar aquellas porciones del programa que interactúan con el usuario del resto del programa.

Algunas gentes asocian la programación multihilos con sistemas multiprocesadores. Los beneficios de un modelo de programación mutihilos puede ser obtenidos incluso si nuestro programa se encuentra corriendo es una computadora de un solo procesador. Un programa puede ser simplificado usando threads sin importar el número de procesadores, porque el número de procesadores no afecta la estructura del programa. Además, aunque tu programa sea bloqueado a causa alguna tarea que está realizando, todavía podemos ver mejora en el tiempo de respuesta, ya que es posible que alguno de los threads puede estar corriendo cuando otros están bloqueados.

Un thread también guarda información relacionada a la tarea que está ejecutando. Esto incluye un identificador del thread dentro del esquema del proceso, un conjunto de registros, una pila, políticas y prioridad de ejecución, una máscara de señales, una variable errno e información específica del thread. Todo lo que existe dentro de un proceso es compartido entre los threads del mismo, incluyendo el texto del programa ejecutable, la memoria global y heap, las pilas y los descriptores de archivos.

9.5.1 Creación de un thread

La llamada `pthread_create` nos permite crear un thread y su definición es la siguiente:

```
#include <pthread.h>
int pthread_create(pthread_t *restrict tipd,
```

```
      const thread_attr_t *restrict attr, void (*start_run) (void*),
      void *restrict arg);
```

`tipd` es un apuntador a la localidad de memoria en donde se encuentra el id que le fue asignado al proceso que acabamos de crear. `attr` es usado para especificar varios atributos del thread. El valor de estos atributos los veremos más adelante, por el momento estableceremos este valor en `NULL`.

`start_rtn` es la función que estará ejecutando el thread en cuanto sea creado. La función solo puede recibir un apuntado a un tipo de dato `void`. En caso de que sea necesario recibir algún parámetro de entrada, éste puede ser pasado a la función a través de `arg` que es un apuntador a un tipo de dato `void`.

Cuando un thread es creado, no existe garantía de cual se ejecute primero: el thread recientemente creado o el proceso que lo creo. Este thread tiene acceso al espacio de memoria del proceso y hereda las variables de ambiente y máscara de señales del proceso que lo generó; sin embargo, el conjunto de señales pendientes para ese thread empieza vacío.

Si la llamada se ejecuta correctamente, regresa cero. En caso de que exista un error, regresa el código del error generado.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

pthread_t ntid;

void printids(const char *txt) {
      pid_t pid;
      pthread_t tid;

      pid = getpid();
      tid = pthread_self();
      fprintf(stdout, "%s pid %d tid %u (0x%x)\n", txt, pid, (unsigned int) tid,
(unsigned int) tid);
}

void* thr_fn(void *arg) {
      printids("new thread: ");
      return ((void *) 0);
}

int main(int argc, char *argv[]) {
      int err;

      err = pthread_create(&ntid, NULL, thr_fn, NULL);
      if (err != 0) {
            fprintf(stderr, "no se puede crear el thread: %s\n", (char *)
strerror(err));
            return -1;
      }
      printids("main thread:");
```

```
        sleep(1);
        return 0;
}
```

En este ejemplo tiene dos peculiaridades. La primera es que es necesario poner a dormir el proceso principal. Si no se pone a dormir, el proceso puede terminar antes de que el thread tenga algún chance de correr. Este comportamiento depende de la implementación de threads que haga el sistema operativo, así como de los algoritmos de calendarización.

La segunda peculiaridad es que el nuevo thread obtiene su identificador llamando al método `pthread_self` en lugar de leerlo de la memoria compartida o de obtenerlo como resultado de una función que inicia el thread.

Para compilar correctamente este código es necesario usar el parámetro `-pthread`:

```
$ gcc thread1.c -phtread
```

9.5.3 Identificación de un thread

Un thread, al igual que un proceso, tiene un identificador. A diferencia del identificador de un proceso, el cual es único en todo el sistema, el identificador de un thread solo tiene significado dentro del contexto del proceso al que pertenece.

Un thread puede obtener su identificador llamando a la función `pthread_self`. Su definición es la siguiente:

```
#include <phtread.h>
pthread_t pthread_self(void);
```

Algunas veces será necesario comparar los identificadores de procesos y para esto tenemos la función:

```
#include <phtread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Que regresa un valor diferente de cero si es ambos identificadores son iguales. En otro caso, devuelve 0.

9.5.4 Terminación de un thread

Si cualquier thread dentro un proceso hace una llamada a `exit`, `_Exit`, `_exit`, entonces el proceso entero termina. De manera similar, cuando la acción por omisión es terminar el proceso, una señal es enviada al thread para terminarlo.

Un thread sencillo puede salir de tres maneras, parando el flujo de control, sin terminar el proceso entero:

1. Un thread termina de la ejecución de su rutina. Regresa el código de salida.
2. El thread puede ser cancelado por otro thread del mismo proceso.
3. El thread puede ejecutar la función `pthread_exit`.

```
#include <phtread.h>
```

```
void pthread_exit(void *rval_ptr);
```

`rval_ptr` es un apuntador, similar al argumento que recibe `start_routine`. Este apuntador esta disponibles a otros threads del proceso a través de la llamada `pthread_join`.

```
#include <phtread.h>
int pthread_join(pthread_t tid, void **rval_ptr);
```

El proceso que invoca la llamada se bloquea a que el thread específico termina, ya sea a través de invocar la llamada `pthread_exit`, termine la ejecución de `start_routine` o si es cancelado. Si el proceso termina la ejecución de `start_routine`, `rval_ptr` contendrá el valor de retorno. Si el proceso es cancelado, el apuntador hará referencia a `PTHREAD_CANCELED`.

Con la llamada `pthread_join`, automáticamente ponemos al thread es un estado que nos permite recuperar sus recursos. Si el thread ya se encuentra en este punto la llamada devolverá EINVAL.

Si no estamos interesados en obtener el valor de terminación del thread, simplemente ponemos NULL en rval_ptr.

En el siguiente código podemos ver a dos threads. El primero terminará a través una llamada `pthread_exit`, mientras que el segundo se termina por el fin de ejecución de la función `start_routine`.