Threads en Linux

Definición de thread

Es un mecanismo por el cual un programa puede hacer mas de una cosa al mismo tiempo.

Como los procesos, los threads parecen correr simultáneamente; pero hay que tener el concepto que el sistema corre una tarea por vez, y esto lo administra el scheduler. Excepción a estos son los sistemas tengan la capacidad de correr mas de un proceso simultáneamente como en el caso de las arquitectura de multi-procesamiento simétrico (SMP).

Podemos decir entonces que un programa que crea un thread tendrá a partir de ese momento dos “time-slot's” en la cola de ejecución del scheduler.

Pero estos dos time-slots's están ejecutando un mismo programa, muy probablemente en puntos de ejecución distintos y comparten el mismo espacio de memoria, descriptores de archivos y otros recursos del sistema.

Esta característica de compartir espacio de memoria y descriptores de archivos facilita la comunicación entre los threads y el programa que los creó. Es decir la problemática de IPC (Inter-Process Communication) está resuelta.

Pero nada podemos asegurar en cuanto a la ejecución. Mucho menos que estos guardén alguna relación en los tiempos de ejecución. Estos temas son privados de Scheduller. No porque un programa creó un thread podemos asegurar que siempre se ejecutará el programa padre antes que el thread, por ejemplo.

Thread implementado en Linux

Cuando se crea un thread en Linux(utilizando pthread_create), se crea un nuevo proceso que ejecuta el thread de referencia.

Pero a diferencia de un proceso creo con fork, este nuevo proceso comparte el espacio de memoria, descriptores de archivos y otros recursos del sistema; con el proceso que originó el thread.

Creando un thread en Linux

El siguiente programa crea un thread.

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

void* print_xs (void* unused) {
  while(1)
    fputc('x',stderr);
  return NULL;
}

int main () {
  pthread_t thread_id;
  pthread_create(&thread_id,NULL,&print_xs,NULL);
  while(1)
    fputc('o',stderr);
  return ;
}
```

thread-create.c

Suponiendo que a este programa le asignemos por nombre thread-pid.c, entonces para compilarlo es suficiente con invocar:

```
$ gcc thread-create.c -o thread-create -lpthread
```

Analizando un poco en código nos encontramos con algunas líneas nuevas, estas son:

La línea “pthread_t thread_id;” crea la variable thread de tipo pid thread. Esto es necesario para almacenar la identificación de proceso thread a crear.

La línea ”fprintf (stderr, “main thread pid is %d\n”, (int) getpid ());” imprime el número de proceso padre por la salida standar de error.

La línea ”pthread_create (&thread, NULL, &thread_function, NULL);” realiza la llamada para crear el thread, el cual comenzara su ejecución con la función thread_function.

Analizando un poco mas en detalle la llamada vemos que posee cuatro argumentos:

1. Un puntero a la variable de tipo pthread_t a donde dejará el pthread del thread creado.
2. Un puntero al objeto de atributos del thread recién creado, en este caso NULL.
3. Un puntero a la función a ejecutar por el thread. Esta función tiene que ser de tipo void* y un argumento de tipo void*.
4. Los argumentos (de tipo void*) a pasarle al thread recién creado.

Inmediatamente después de ejecutar esta llamada el programa principal continúa con su ejecución. En este caso, se cuelga :) (Atención que esto suele ser devorador de CPU, para comprobarlo basta con ejecutar el comando top en otra terminal y ver como aumenta el consumo de CPU)

El detalle de lo realizado por la función thread_function es irrelevante y me relevo de explicarlo.

Thread implementado en Linux, visto por el scheduller

Si se ejecuta el programa analizado encontraremos algo así al ejecutar el comado ps en otra consola:

```
mdoallo@pbxdgsinf2:~$ ps -eLf | grep thread-create

sh-3.1$ ps -LfC thread-create

UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD

mdoallo     5387   4141   5387 10       2 15:49 pts/1       00:00:01
./thread-

create

mdoallo     5387   4141   5388 39       2 15:49 pts/1       00:00:06
./thread-

create

sh-3.1$
```

El proceso 5387 (con Light Weigth Process 5387) es el proceso original que invocó la llamada **pthread_create**, esta llamada hizo que el sistema cree el proceso 5387 (con Light Weigth Process 5388).

Notar que el tiempo de CPU insumido por los procesos son importantes (0:06) en comparación al resto, esto es así gracias a los “**while(1)**;” que existen en ambos procesos.

Finalización de procesos y threads

Es importante conocer que si el proceso padre de los threads finaliza antes que estos, los mismos finalizarán por mas que no lo deseen.

Por ejemplo, si en el programa anterior evitáramos el “while(1);” en el proceso padre y le permitiríamos

finalizar; obligaría al thread a finalizar también.

Les recomiendo que lo hagan. ¡No van a romper nada!

Pero no hace falta que el proceso padre espere con un “while(1);”, para esto existe la llamada **pthread_join**.

Entonces para mejorar nuestro ejemplo, este podría quedar de la siguiente manera:

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

void* char_print (void* parameters) {
  int i;
  for(i = 0; i < 300; ++i)
    fputc('x', stderr);
  return NULL;
}

int main() {
  pthread_t thread_id;
  pthread_create(&thread_id, NULL, &char_print, NULL);
  pthread_join (thread_id, NULL);
  return 0;
}
```

thread-join.c

Analizando en detalle este segundo ejemplo encontramos que la llamada a **pthread_join** tiene la siguiente forma, ”pthread_join (thread_id, NULL);” el primer argumento indica el thread por el cual el programa espera y el segundo sirve para recibir parámetros del thread citado, esta última también de tipo puntero a void* (Al igual que el argumentos de la llamada pthread_create para pasarle argumentos al thread creado)

Retorno de valores de un thread

El siguiente programa muestra un ejemplo para devolver un entero desde un thread al programa padre utilizando casteo.

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

void* compute_prime (void* arg) {
  int candidate = 2;
  int n = *((int*) arg);
  while (1) {
    int factor;
    int is_prime = 1;
    for (factor = 2; factor < candidate; ++factor)
      if (candidate % factor == 0) {
     is_prime = 0;
     break;
      }
      if (is_prime) {
     if (--n == 0) {
       printf("Terminé\n");
       return (void*) candidate;
     }
      }
      ++candidate;
  }
  return NULL;
}

int main () {
```

```
  pthread_t thread;
  int which_prime = 10000;
  int prime;
  pthread_create (&thread, NULL, &compute_prime, &which_prime);
  sleep(60);
  pthread_join (thread, (void*) &prime);
  printf("The %dth prime number is %d.\n", which_prime, prime);
  return 0;
}
```

thread-join-2.c

Pasaje de argumentos a un thread

Para pasar argumentos a un thread hay que usar un puntero a alguna estructura, vector o variable utilizando casteo.

En el siguiente ejemplo le indicamos al thread que caracter queremos imprimir en pantalla y la cantidad de caracteres a imprimir, por medio de argumentos.

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

struct char_print_parms {
  char character;
  int count;
};

void* char_print (void* parameters) {
  struct char_print_parms* p = (struct char_print_parms*)parameters;
  int i;
  for(i = 0; i < p->count; ++i)
    fputc(p->character, stderr);
  return NULL;
}

int main() {
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  thread1_args.character = 'x';
  thread1_args.count = 300;
  pthread_create(&thread1_id, NULL, &char_print, &thread1_args);

  thread2_args.character = 'o';
  thread2_args.count = 300;
  pthread_create(&thread2_id, NULL, &char_print, &thread2_args);

  pthread_join(qhread1_id,NULL);
  pthread_join(thread2_id,NULL);

  return 0;
}
```

thread-create2.c

Atributos de un thread

Para ajustar los atributos de un thread hay proceder de la siguiente manera:

1. Crear un objeto de tipo pthread_attr_t
2. Utilizar la llamada **pthrad_attr_init** para iniciar el objeto creado en el punto 1.
3. Modificar los atributos a gusto, veremos luego como.
4. Pasar un puntero al objeto cuando se invoca la llamada **pthread_create**.
5. Utilizar la llamada **pthread_attr_destroy** para liberar la variable pueda ser re-utilizada.

Un thread puede ser creado como *joinable* o *detach*. Por defecto, si no se especifica, un thread será joinable.

Un thread joinable requiere que el proceso padre utilice pthread_join para liberarlo del sistema, si no queda como zombie, cuando este finalice y el padre no finalizó.

Un thread detach es liberado automáticamente por el sistema luego de su finalización, pero esto tiene un inconveniente, es imposible sincronizar otro thread con su finalización, ni obtener resultado del mismo.

El siguiente fragmento de programa muestra como crear un thread de tipo detach:

```
#include <pthread.h>

void* thread_function (void* thread_arg) {

  /* Thread detach ... */

}

int main () {

  pthread_attr_t attr;

  pthread_t thread;

  pthread_attr_init (&attr);

  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);

  pthread_create (&thread, &attr, &thread_function, NULL);

  pthread_attr_destroy (&attr);

  /* El thred detach está corriendo ... */

  /* No es necesario la llamada a pthread_join. */

  /* ¿Pero que pasa si el programa padre finaliza? */

  return 0;

}
```

Sincronización de Threads

Comencemos analizando el problema de la falta de sincronismo entre thread's para comprender su importancia.

Por ejemplo, si un thread ha actualizado parcialmente una estructura de datos cuando a otro thread se le otorga su ventana de ejecución (scheduller) y accede a la misma estructura de datos y los manipula a su antojo; muy probablemente el thread anterior se cuelgue cuando continúe con su ejecución. A este problema se lo conoce como "*race conditions*".

Para eliminar este problema necesitamos una forma de hacer las operaciones de estos thread "*atómicas*".

Una operación es atómica cuando es indivisible e ininterrumpible; cuando esta comienza no será interrumpida, ni pausada hasta que se complete. Ninguna otra operación tomará lugar mientras tanto.

Mutex

Linux posee para los threads un mecanismo de llaves mutuamente exclusivas denominado *mutexes* (por la abreviación de mutuamente exclusiva).

Se trata de un mecanismo que sólo un thread a la vez puede cerrar. Si un thread cierra un mutex y luego un segundo thread también trata de cerrarlo, este segundo thread quedará bloqueado. Solamente cuando el primer thread libera el mutex, el segundo thread es desbloqueado, permitiéndole continuar con la ejecución.

Para implementar este mecanismo se debe crear una variable previamente de tipo pthread_mutex_t y pasarla como puntero a la llamada **pthread_mutex_init**.

La llamada pthread_mutex_init tiene dos argumentos:

1. Un puntero a la variable de tipo pthread_mutex_t
2. Un puntero al objeto de atributos del mutex a inicializar. (Se puede poner NULL y tomará los atributos por defecto)

Luego que la variable mutex existe y está inicializada el thread podrá cerrarla mediante la llamada **pthread_mutex_lock**, que tiene un sólo argumento, un puntero a la variable mutex a cerrar.

Para liberar un mutex se invoca la llamada **pthread_mutex_unlock**, que también tiene un sólo argumento, un puntero a la variable mutex a liberar.

Es interesante la llamada **pthread_mutex_trylock**, puede opera igual que pthread_mutex_lock pero en lugarde bloquear al thread en el caso que la variable ya fuera cerrada por otro thread, devolverá EBUSY como código de error sin bloquear el thread.

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

FILE *fd;
pthread_mutex_t mutex;

void *escribe (void *data) {
  char str[30];
  strcpy(str,(char*)data);
  printf("La cadena a escribir en el archivo es:  %s",str);
  /* se lock el mutex para que nadie mas escriba en el archivo */
  pthread_mutex_lock(&mutex);
  printf("Mutex locked\n");
  printf("Escribo en el archivo\n");
  if (fprintf(fd,"%s",str) < 0 ) {
```

```
    perror("fwrite");
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
  }
  sleep(3);
  /* se unlock el mutex para que otro thread pueda usar el archivo*/
  printf("Mutex unlocked\n");
  pthread_mutex_unlock(&mutex);
  pthread_exit(NULL);
}

int main() {
  int count,cant;
  char buf[30];
  pthread_t id;
  printf("Abro archivo al que escribiran los threads\n");
  if (((fd = fopen("archivo.txt","w+")) == NULL)) {
    perror("fopen");
    exit (1);
  }
  printf("Inicializo el mutex\n");
  pthread_mutex_init(&mutex,NULL);
  count = 20;
  cant = 1;
  while(count) {
    sprintf(buf,"Soy el thread numero %d\n",cant);
    printf("El contenido del buffer es: %s",buf);
    /*creacion del thread*/
    if (pthread_create(&id,NULL,&escribe,buf)!= 0 ) {
      perror("pthread_create");
      exit(1);
    }
    cant++;
    count--;
    sleep(2);
  }
  pthread_join(id,NULL);
  fclose(fd);
  exit(0);
}
```

mutex1.c