Pietro Pino (pepy91)

GESTIONE DEI PROCESSI E DEI THREADS IN UN SISTEMA LINUX (PARTE 2)

16 June 2013

I THREADS

I **threads**, come i processi, sono un meccanismo che permette ad un programma di fare più di una cosa nello stesso momento. Concettualmente, un thread esiste dentro un processo. Quando invochiamo un programma, Linux crea un nuovo processo e in quel processo crea un singolo thread, che esegue il programma sequenzialmente. Quel thread può creare altri threads, e, insieme, eseguono lo stesso programma nello stesso processo, ma ogni thread esegue una parte diversa del programma nel tempo dato.
Quando un programma crea un altro thread, non viene copiato nulla (a differenza di quanto avviene nei processi). Il thread creante e quello creato condividono lo stesso spazio di memoria, i file descrittori e altre risorse del sistema. Se un thread cambia il valore di una variabile, per esempio, l'altro thread in seguito vedrà il valore modificato.

I sistemi GNU/Linux implementano il thread POSIX API (conosciuto come ***pthreads***). Tutte le funzioni thread e i tipi di dati sono dichiarati nell'header file **<pthread.h>**. Le funzioni non sono incluse nella libreria C, ma si trovano in **libpthread**, che è possibile aggiungere inserendo ***-lpthread*** sulla linea di comando quando compiliamo il programma.

CREAZIONE DI UN THREAD

Ogni thread in un processo è identificato da un **thread ID**. Quando ci riferiamo ai thread ID nei programmi C, bisogna usare il tipo **pthread_t**. Per creare un thread, o da un processo o da un thread, usiamo la chiamata

```
int pthread_create (pthread_t *THREAD, pthread_attr_t *ATTR,

void *(*START_ROUTINE)(void *), void *ARG)
```

Analizziamo ogni parametro passato alla funzione *pthread_create*:

1. *pthread_t *THREAD* : puntatore ad una variabile pthread_t, in cui è contenuto il thread ID del nuovo thread;
2. *pthread_attr_t *ATTR*: puntatore ad un oggetto pthread attribute. Questo oggetto controlla i dettagli di come il thread interagisce con il resto del programma. Se passiamo NULL come valore a questo parametro, verrà creato un thread con *default thread attributes*;
3. *void *(*START_ROUTINE)(void *)*: puntatore ad una funzione thread. Si tratta di una funzione ordinaria che contiene il codice che il thread dovrebbe eseguire. Quando la funzione ritorna, il thread esce. In Linux, le funzioni thread prendono un singolo parametro, di tipo *void **, e ritornano un tipo *void **. Il parametro in questione viene chiamato **thread argument**: in Linux, viene passato il valore al thread senza considerarlo. Il programma può usare questo parametro per passare dati ad un nuovo thread, cosi come può utilizzare il valore di ritorno per ritornare dati da un thread uscente al suo creatore;
4. *void *ARG*: valore del thread argument di tipo *void **. Qualunque cosa passiamo è semplicemente passato all'argomento della funzione thread quando il thread comincia ad eseguire.

Una chiamata a *pthread_create* ritorna immediatamente, e il thread originale continua ad eseguire le istruzioni che seguono la chiamata. Nel frattempo, il nuovo thread comincia ad eseguire la funzione thread.
Ecco il codice per creare un thread:

```
#include <pthread.h>

#include <stdio.h>


void * print_xs(void *unused){   /*Stampa 'x' su stderr. Il parametro non

è usato. Non viene ritornato alcun valore*/

    while(1)

    fputc('x',stderr);

    return NULL;

  }
```

```
int main(){

    pthread_t thread_ID;

    pthread_create (&thread_ID,NULL,&print_xs,NULL);  /*Crea un nuovo thread,
che eseguirà la funzione print_xs */

    while (1)    /*Stampa 'o' continuamente su stderr*/
        fputc ('o',stderr);

    return 0;

}
```

Compila ed esegui questo programma utilizzando questo codice:

```
$gcc thread_create.c -o thread_create -lpthread
```

PASSARE DATI AI THREADS

Il **thread argument** fornisce un metodo per passare dati ai threads. Poichè il tipo dell'argomento (argument) è *void **, non è possibile passare dati direttamente via argument. Invece, usiamo il *thread argument* per passare un puntatore ad alcune strutture o array di dati. Una tecnica comune è quella di definire una struttura per ogni funzione thread, che contiene i "parametri" che la funzione thread si aspetta. Usando il *thread argument*, è facile riusare la stessa funzione thread per molti threads, i quali eseguono lo stesso codice, ma su dati differenti.

Vediamo il codice di un programma che crea due nuovi threads, uno per stampare 'x' e un altro per stampare 'o'. Invece di stamparli all'infinito, ogni thread stampa un numero fissato di caratteri e dopo esce ritornando dalla funzione thread. La stessa funzione thread,***char_print()***, è usata per entrambi i threads, ma ognuno è configurato differentemente usando ***struct char_print_parms***:

```
#include <pthread.h>

#include <stdio.h>
```

```
struct char_print_parms{
      char character;
      int num;
 };

void* char_print (void* parameters){
/*Cast del puntatore al tipo corretto*/
      struct char_print_parms *p = (struct char_print_parms *)parameters;
      int i;
      for (i=0 ; i < p->num ; ++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.num=30000;
      pthread_create (&thread1_ID, NULL, &char_print, &thread1_args);  /*Crea un nuovo
```

```
che stampa 30000 volte 'x' */


      thread2_args.character = 'o';

      thread2_args.num = 20000;

      pthread_create (&thread2_ID, NULL, &char_print, &thread2_args);  /*Crea un nuovo
che stampa 20000 volte 'o*/


      return 0;

 }
```

Ma ATTENZIONE!! La soluzione proposta ha un bug: i parametri strutturati passati ai thread (*thread1_args* e *thread2_args*) sono locali al main(), e inoltre il thread principale, che esegue la funzione *main* passa ai thread che ha creato i puntatori a questi parametri. Se il thread che esegue il *main* completa prima degli altri, la memoria che contiene i parametri strutturati passati ai threads verrà deallocata mentre gli altri due thread ci stanno appena accedendo.
Una soluzione è obbligare il *main* ad aspettare finchè gli altri due threads non finiscono. E questo avviene grazie alla funzione ***pthread_join()***, che prende due argomenti: il thread ID del thread da aspettare e un puntatore ad una variabile *void* * che riceverà il valore di ritorno del thread finito. Se ciò non interessa, passiamo NULL come secondo argomento.
Questa è la chiamata di sistema:

```
int pthread_join (pthread_t TH, void **thread_RETURN);
```

e la inseriamo nel codice precedente, tra il secondo thread e il return 0:

```
.....

pthread_join(thread1_ID,NULL);


pthread_join(thread2_ID,NULL);
```

```
return 0;

}
```

Il prossimo post sarà a proposito di un argomento forse tra i più importanti e delicati dell'intero complesso, un argomento che andrà trattato con molta cautela per evitare errori incommensurabili: i **SEMAFORI** e i **MUTEX**. Alla prossima! PS: Grazie a tutti per i voti che mi avete assegnato, spero che possiate appassionarvi a questi argomenti perchè significa che ho fatto un buon lavoro. Grazie ancora!

Estratto da "http://www.electroyou.it/mediawiki/index.php?title=UsersPages:Pepy91:gestione-dei-processi-e-dei-threads-in-un-sistema-linux-parte-2"