Pietro Pino (pepy91)

GESTIONE DEI PROCESSI E DEI THREADS IN UN SISTEMA LINUX (PARTE 3)

4 July 2013

I SEMAFORI

La causa della maggior parte dei bugs che coinvolgono i threads è che i threads accedono agli stessi dati. Spesso i programmi pieni di bug contengono un codice che agirà solo se un thread viene schedulato più spesso, o prima, di un altro thread. Questi bug sono chiamati **race conditions**.
Supponiamo che un programma abbia una serie di processi in coda che sono trattati da numerosi thread concorrenti. La coda dei processi è rappresentata da una lista di oggetti di tipo ***struct job***. Dopo che ogni thread termina un'operazione, controlla la coda per vedere se è disponibile un altro processo. Se la coda dei processi (***job_queue***) è non-nulla, il thread rimuove la testa di quella lista e imposta *job_queue* per il prossimo processo sulla lista.
Vediamo il codice della FUNZIONE THREAD PER TRATTARE PROCESSI DALLA CODA

```
#include <malloc.h>

struct job{

    struct job *next;

  };


struct job *job_queue; /*Lista concatenata per processi in sospeso*/


void* thread_function (void *arg){  /*Trattamento processi in coda

                                fino a quando la coda è vuota*/

      while (job_queue!=NULL){
```

```
            struct job *next_job = job_queue; /*Prendi il prossimo processo disponibil
            job_queue = job_queue->next; /*Rimuovi questo processo dalla lista*/
            process_job (next_job); /*Esegui il lavoro*/
            free (next_job); /*Pulisci*/
        }
     return NULL;
    }
```

Supponiamo ora che due thread finiscono un processo nello stesso momento, ma rimane solo un processo nella coda. Il primo thread controlla se *job_queue* è nulla; trovando che non lo è, il thread entra e fornisce il puntatore al processo nel *next_job*. A questo punto, Linux interrompe il primo thread e schedula il secondo. Anche il secondo thread controlla *job_queue* e vedendo che non è vuoto, assegna lo stesso puntatore al processo a *next_job*. Per una sfortunata coincidenza, ci sono due thread che eseguono lo stesso programma. Questo è un esempio di *RACE CONDITIONS*. Per eliminarlo, c'è bisogno di un modo per fare operazioni **atomiche**, cioè indivisibili e che non è possibile interrompere: una volta che l'operazione comincia, non si può fermare finchè non completa, e nessun'altra operazione può sostituirla nel frattempo. In questo particolare esempio, si vuole controllare *job_queue*: se non è vuota, rimuove il primo processo, tutto con una singola operazione atomica.

LA MUTUA ESCLUSIONE

La soluzione al problema della race condition nella coda dei processi è di lasciare che solo un thread alla volta acceda alla coda dei processi. Una volta che un thread inizia a controllare la coda, nessun altro thread può accedere finchè il primo thread ha deciso se trattare un processo e, se così fosse, finchè ha rimosso il processo della lista. GNU/Linux fornisce i cosiddetti **mutex**, che sta per **MUTual EXclusion locks**. Un mutex è una serratura speciale che solo un thread alla volta può chiudere. Se un thread chiude un mutex e dopo un secondo thread cerca di chiudere lo stesso mutex, il secondo thread viene **bloccato**. Solo quando il primo thread apre il mutex, il secondo viene **sbloccato**, lasciando riprendere l'esecuzione. Quindi solo un thread prenderà la serratura e tutti gli altri thread saranno bloccati.
Per creare un mutex, creiamo una variabile di tipo **pthread_mutex_t** e passiamo un puntatore ad esso alla funzione **pthread_mutex_init()**, la quale, oltre al puntatore a quella variabile, prende in ingresso, come secondo argomento un attributo dal semaforo. La variabile mutex deve essere inizializzata solo una volta. Vediamo il codice di dichiarazione e inizializzazione di una variabile mutex:

```
pthread_mutex_t mutex; /*mutex da inizializzare*/
```

```
pthread_mutex_init (&mutex,NULL);
```

La funzione generale è questa:

```
 int pthread_mutex_init (pthread_mutex_t *MUTEX, const pthread_mutexattr_t *MUTEXATTR);
```

MUTEXATTR rappresenta gli attributi associati al semaforo e può assumere i seguenti valori:

- **fast**: inizializzazione con il valore speciale **PTHREAD_MUTEX_INITIALIZER**;
- **recursive**: inizializzazione con il valore speciale **PTHREAD_RECURSIVE_MUTEX_NP**;
- **error_checking**: inizializzazione con il valore speciale **PTHREAD_ERRORCHECK_MUTEX_NP**;
- **NULL**.

Un thread può tentare di chiudere un mutex con la chiamata

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Se il mutex era aperto, si chiude e la funzione ritorna subito. Se il mutex era chiuso da un altro thread, la funzione blocca l'esecuzione e ritorna solo quando il mutex è aperto dall'altro thread. Più di un thread può essere bloccato su un mutex chiuso nello stesso tempo. Quando il mutex è aperto, solo uno dei thread bloccati si sblocca e chiuderà il mutex; gli altri thread rimarranno bloccati.
Per aprire un mutex, esiste la chiamata

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Il comportamento dipende dagli attributi del semaforo:

- fast: lo stato di mutex diventa unlocked;
- error checking: lo stato diventa unlocked se il thread è proprietario altrimenti viene tornato l'errore EPERM;
- recursive: se il contatore è >0, allora lo stato è locked, e il contatore decrementato; se il contatore è =0, allora lo stato diventa unlocked.

Vediamo il codice della FUNZIONE THREAD PER LA CODA DEI PROCESSI, PROTETTA DA UN MUTEX:

```
#include <malloc.h>

#include <pthread.h>


struct job{

   struct job *next;

 };


struct job *job_queue;


pthread_mutex_t job_queue_mutex=PTHREAD_MUTEX_INITIALIZER;/*Mutex che protegge job_queu


void* thread_function (void *arg){

   while (1){

         struct job *next_job;

         pthread_mutex_lock (&job_queue_mutex); /*Chiude il mutex sulla coda dei process


         if (job_queue==NULL) /*Ora può controllare se la coda è vuota*/

                next_job = NULL;


         else {

                next_job = job_queue; /*Prende il successivo processo disponibile*/

                job_queue = job_queue->next; /*Lo rimuove dalla lista*/

              }
```

```
        pthread_mutex_unlock (&job_queue_mutex); /*Apre il mutex sulla coda
                                dei processi perchè la coda non serve per ora*/

        if (next_job==NULL) /*La coda era vuota? Se sì, finisce il thread*/
                break;

        process_job (next_job);
        free (next_job);
      }
   return NULL;
  }
```

In questo codice, prima di accedere alla coda, ogni thread chiude per prima cosa un mutex. Solo quando l'intera sequenza di controllo della coda e rimozione del processo sono completate, ecco che il mutex viene aperto. Da notare che se la coda è vuota, non si esce dal nodo subito perchè questo lascerebbe il mutex sempre chiuso e impedirebbe ad ogni altro thread di accedere alla coda dei processi. Invece, bisogna settare *next_job* a NULL e uscire solo dopo l'apertura del mutex. L'uso del mutex per chiudere *job_queue* non è automatico: deve essere il programmatore ad aggiungere il codice per chiuderlo prima di accedere a quella variabile e dopo per aprirla successivamente.
Per esempio, una funzione per aggiungere un processo alla coda dei processi potrebbe avere questo codice:

```
void enqueue_job (struct job *new_job){
     pthread_mutex_lock (&job_queue_mutex);
     new_job->next = job_queue;
     job_queue = new_job;
```

```
    pthread_mutex_unlock (&job_queue_mutex);

}
```

MUTEX DEADLOCKS

I mutex forniscono un meccanismo per permettere ad un thread di bloccare l'esecuzione di un altro. Questo apre alla possibilità di una nuova classe di bug, chiamati **deadlocks**. Un *deadlock* si ha quando uno o più thread aspetta qualcosa che mai accadrà. Un semplice caso di deadlock potrebbe accadere quando lo stesso thread tenta di chiudere un mutex due volte. Il comportamento, in questo caso, dipende dal tipo di mutex che si sta usando:

- **fast mutex**: causerà un deadlock. Un tentativo di chiudere il mutex bloccherà finchè il mutex verrà aperto. Ma poichè il thread che ha chiuso il mutex è bloccato sullo stesso mutex, la serratura non può mai essere rilasciata;
- **recursive mutex**: non causa un deadlock; un mutex ricorsivo può essere chiuso molte volte dallo stesso thread. Il mutex ricorda quante volte la funzione *pthread_mutex_lock()* è stata chiamata dal thread che possiede il lock; quel thread deve effettuare lo stesso numero di chiamate alla funzione *pthread_mutex_unlock()*, in modo da aprire il mutex e da farlo chiudere da un altro thread;
- **error-checking mutex**: la seconda chiamata consecutiva alla funzione *lock* ritornerà il codice EDEADLK.

ELIMINAZIONE DI UN MUTEX

La funzione che eliminerà il mutex è

```
int pthread_mutex_destroy (pthread_mutex_t *MUTEX);
```

Il semaforo MUTEX viene eliminato e le risorse ad esso relative rilasciate. Il semaforo deve essere *unlocked* (aperto) e in questo caso avremmo 0 come valore di ritorno. Altrimenti, se il semaforo è *lock* (chiuso) viene ritornato l'errore EBUSY.

TEST DI UN MUTEX

Ogni tanto è conveniente verificare se un mutex è chiuso senza blocchi su di esso. Per esempio, un thread potrebbe aver bisogno di chiudere un mutex, ma se è già chiuso, potrebbe avere altro lavoro da fare invece di essere bloccato. Poichè *pthread_mutex_lock()* non ritornerà finchè il mutex non diventa aperto, abbiamo bisogno di un'altra funzione. GNU/Linux fornisce la funzione **pthread_mutex_trylock()**. Se chiamiamo questa funzione su un mutex aperto, chiuderemo il mutex come se avessimo chiamato la funzione *lock*, e il *trylock* ritornerà 0. Comunque, se il mutex è già chiuso da un altro thread, il*trylock* non bloccherà, ma ritornerà con il codice di errore EBUSY.

Per oggi è tutto. Il prossimo articolo, che sarà l'ultimo su questo bellissimo argomento, riguarderà i semafori generalizzati, usati, come vedremo, per sincronizzare più thread. Alla prossima, amici di ElectroYou!!

Estratto da "http://www.electroyou.it/mediawiki/index.php?title=UsersPages:Pepy91:gestione-dei-processi-e-dei-threads-in-un-sistema-linux-parte-3"