Programmazione concorrente – comunicazione tra processi

Leonardo Giordani
<leo.giordani(at)libero.it>

L'autore:

Student at the Faculty of Telecommunication Engineering in Politecnico of Milan, works as network administrator and is interested in programming (mostly in Assembly and C/C++). Since 1999 works almost only with Linux/Unix.

Tradotto in Italiano da:
Alessio Calcagno
<alessiothepally(at)yahoo.it>

Premessa:

Questa serie di articoli ha lo scopo di introdurre il lettore al concetto di multitasking e alla sua implementazione in linux. . Partendo dai concetti teorici che stanno alla base del multitasking arriveremo a scrivere una completa applicazione che mostri come avviene la comunicazione tra processi con un semplice ma efficiente protocollo di communicazione.

Prerequisiti per capire questi articoli sono:

- Una minima conoscenza della shell
- Conoscenza base del linguaggio C

Prima di leggere questo articolo dovresti aver letto il precedente: November 2002, article 272.

Introduzione

Anche in questo articolo ci occuperemo del multitasking in linux . Come abbiamo già visto nel precedente articolo, creare un processo figlio attraverso l'uso della chiamata di sistema fork() necessita di poche linee di codice, poichè il sistema operativo si cura dell'inizializzazione e della gestione del processo. Ma demandando al sistema operativo il compito di gestire i processi ne perdiamo il controllo e questo porta allo sviluppatore dei problemi di sincronizzazione riassunti da questa domanda: come fanno due processi indipendenti a lavorare assieme?

Il problema è più complesso di quanto possa sembrare, infatti non è solo una questione di sincronizzazione dell'esecuzione dei processi ma anche della condivisione dei dati sia in lettura che in scrittura.

Innanzitutto introduciamo qualche problema classico di accesso concorrente a dei dati; se due processi leggono lo stesso insieme di dati questo ovviamente non comporta nessun problema ma non si può dire la stessa cosa se uno dei due può modificare i dati.Infatti l'atro processo ritornerà valori differenti a seconda di quando legge i dati, prima o dopo che il primo processo li abbia modificati. Ad esempio se abbiamo due processi A e B e un intero d; il processo A incrementa d mentre il processo B stampa a video il valore di D.

A { d−>d+1 } & B { d−>output }

dove l'& rappresenta un esecuzione concorrente. Una prima esecuzione potrebbe essere:

(−) d = 5 (A) d = 6 (B) output = 6

Ma se il processo B fosse eseguito prima noi otterremmo:

(−) d = 5 (B) output = 5 (A) d = 6

Si può cosi' capire immediatamente quanto sia importante gestire correttamente queste situazioni. Il rischio di avere dati inconsistenti è grande ed inaccettabile.Prova a pensare che i dati su cui si lavora rappresentino il tuo conto in banca e allora non sottostimerai il problema.

Nel precedente articolo abbiamo già parlato del problema della sincronizzazione introducendo la funzione waitpid, la quale permette ad un processo di attendere la fine di un altro processo prima di proseguire.Ciò ci permette di risolvere alcuni problemi riguardanti la lettura e la scrittura di dati condivisi: una volta definito l'insieme dei dati condivisi da P1 il processo P2 che lavora sugli stessi dati aspetterà la fine di P1 prima di procedere con la propria esecuzione.

Chiaramente questo metodo rappresenta una prima soluzione ma è molto lontano dalla migliore, infatti P2 deve stare idle (a non far nulla) per un tempo che potrebbe essere anche lungo aspettando che P1 termini la sua esecuzione mentre P1 potrebbe già aver finito da molto tempo di lavorare con i dati condivisi. Ciò che dobbiamo fare è aumentare la granularità dei nostri controlli per esempio governando l'accesso ad un singolo dato o a un insieme di dati. La soluzione di questo problema può essere dato da un insieme di primitive della libreria standard conosciuta come SysV IPC.

SysV keys

prima di introdurre argomenti riguardanti da vicino la teoria della concorrenza permettici di introdurre una tipica struttura SysV: IPC keys. Una chiave IPC è un numero utilizzato per definire univocamente una struttura di controllo IPC (descritta in seguito).

Un modo per creare una chiave IPC è utilizzare la funzione ftok

key_t ftok(const char *pathname, int proj_id);

la quale ha come parametri il nome di un file esistente (pathname) ed un intero e restituisce una chiave basata sull' inode del file. Non è assicurato che la chiave sia unica quindi una buona soluzione è creare una libreria che tenga traccia delle chiavi assegnate ed eviti i duplicati.

Semaphores

L'idea di un semaforo per il controllo del traffico può essere usata senza grandi modifiche per il controllo di accesso ai dati. Un semaforo è una particolare struttura contenente un valore maggiore o uguale a zero e che gestisce una coda di processi che aspettano una particolare condizione del semaforo stesso.

I semafori possono essere usati per controllare l'accesso ad una risorsa: il valore del semaforo rappresenta il numero di processi che possono accedere alla risorsa.Ogni qualvolta che un processo accede alla risorsa il valore del semaforo deve essere decrementato e incrementato nuovamente quando la risorsa viene rilasciata. Se l'accesso alla risorsa è esclusivo (solo un processo alla volta può accedervi) il valore iniziale del semaforo sarà 1.

Un altro compito che può essere svolto dai semafori è il contattore della risorsa: il valore rappresenta in questo caso il numero di risorse disponibili (ad esempio il numero di celle di memoria disponibili).

Consideriamo ora un caso pratico in cui viene usato un semaforo. Immaginiamo che abbiamo un buffer nel quale molti processi S1.....Sn possono scrivere ma dal quale un solo processo L possa leggere. Quindi un solo processo alla volta potrà accedere al buffer senza causare problemi ed inoltre i processi Si potranno scrivere solo se il buffer non sarà pieno mentre i processi L potranno leggere solamente se il buffer non sarà vuoto.

Considerando che l'accesso al buffer sarà esclusivo il primo semaforo prenderà solo i valori 0 ed 1 mentre gli altri 2 semafori prenderanno dei valori corrispondenti alle posizioni vuote e piene del buffer.

Impariamo ora ad implementare i semafori in C usando la libreria SysV e la funzione semget(2).

In questa libreria un semaforo non è semplicemente un intero non negativo ma un insieme (array) di valori. Quando si crea un semaforo è necessario specificare il numero di tali valori. Inoltre bisogna sapere che la creazione di un semaforo non ne permette anche l' inizializzazione che invece viene effettuata in un secondo momento tramite la semctl. Diamo ora l'interfaccia della funzione semget:

int semget(key_t key, int nsems, int semflg);

dove :

key è una chiave IPC

nsem è il numero di semafori che noi vogliamo creare

semflg è l'access controll implementato con 12 bits.

 I primi tre riguardano le politiche di creazione gli altri 9 riguardano i permessi d'accesso del creatore, degli appartenenti al gruppo del creatore e di tutti gli altri.

Alcune costanti per il parametro semflg sono:

IPC_CREAT che crea un nuovo array di semafori se non ne esisteva già uno avente lo stesso valore di key altrimenti restituisce l' id del semaforo già esistente.

IPC_EXCL che in congiunzione con IPC_CREAT specifica la creazione di un nuovo array di semafori. Se un array di semafori con la chiave specificata esisteva già restituisce il valore −1.

Esempio:

semid = semget(key, 1, 0600| IPC_CREAT| IPC_EXCL);

Crea un nuovo semaforo con chiave key e permessi di lettura scrittura al proprietario e nessuno diritto a tutti gli altri.

Creiamo il nostro primo semaforo:

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/types.h>
#include <linux/ipc.h>
#include <linux/sem.h>

int main(void)
{
  key_t key;
  int semid;

  key = ftok("/etc/fstab", getpid());

  /* create a semaphore set with only 1 semaphore: */
  semid = semget(key, 1, 0666 | IPC_CREAT);

  return 0;
}
```

dobbiamo ancora imparare come gestire e rimuovere i semafori tramite la primitiva semctl, la cui interfaccia è la seguente:

int semctl(int semid, int semnum, int cmd,.....)

che opera in accordo all'azione identificata dal parametro cmd, sull'insieme (array) di semafori identificati da semid e sul semaforo identificato da semnum (i semafori partono da 0). Introdurremo alcune azioni ma una lista completa può essere trovata nella pagina del manuale.

Questa funzione può avere tre o quattro argomenti, quando ne ha quattro l'ultimo argomento ha tipo union semun definita come segue:

```
union semun {
 int val;                  /* value for SETVAL */
 struct semid_ds *buf;     /* buffer for IPC_STAT, IPC_SET */
 unsigned short *array;    /* array for GETALL, SETALL */
                           /* Linux specific part: */
 struct seminfo *__buf;    /* buffer for IPC_INFO */
};
```

Per settare il valore di un semaforo bisogna utilizzare la direttiva SETVAL e il valore deve essere specificato nell'union semun. Andiamo a modificare il programma precedente settando il valore del semaforo ad 1.

```
[...]

  /* create a semaphore set with only 1 semaphore */
  semid = semget(key, 1, 0666 | IPC_CREAT);
```

```
  /* set value of semaphore number 0 to 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

[...]
```

Poi quando dobbiamo rilasciare il semaforo deallocando la struttura usata per la sua gestione useremo la direttiva IPC_RMID of semctl. questa direttiva rimuove il semaforo e manda un messaggio a tutti i processi che sono in attesa per accedere alla risorsa condivisa.

```
[...]

  /* set value of semaphore number 0 to 1 */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

  /* deallocate semaphore */
  semctl(semid, 0, IPC_RMID);

[...]
```

Come visto sopra creare e gestire una struttura per controllare la concorrenza non è difficile;quando introduciamo la gestione degli errori le cose divengono leggermente più complicate ma solo da un punto di vista della complessità del codice.

Il semaforo può essere ora utilizzato attraverso la funzione semop(2) che restituisce −1 in caso di fallimento di una delle operazioni comandate.

int semop(int semid, struct sembuf *sops, unsigned nsops);

dove semid è l'identificatore dell'array di semafori, sops è un array contenente le operazioni che debbono essere eseguite e nsops il numero di queste operazioni. Ogni operazione è rappresentata da una sembuf struct.che è composta dai seguenti tre campi:

unsigned short sem_num; short sem_op; short sem_flg;

Dove sem_num indica il semaforo su cui operare, sem_flag specifica la politica di attesa sul semaforo (assumiamo per ora che 0 vada bene) ed infine sem_op che è un numero intero ed opera nella seguente maniera

1. sem_op < 0
   Se il valore del semaforo è più grande o uguale a quello di sem_op allora sem_op viene aggiunto al valore del semaforo (in realtà sottratto poichè sem_op è negativo). Se il valore assoluto di sem_op è piu grande del valore del semaforo allora il processo cadrà in uno stato di addormentato finchè non saranno disponibili un tal numero di risorse. (in altre parole se s è il valore del semaforo allora se s+sem_op<0 allora sospendi il processo finchè s+sem_op>0 quando s+sem_op>0 allora esegui l'istruzione s=s+sem_op che decrementerà s del valore di sem_op)
2. sem_op = 0
   Il processo dormirà finchè il valore del semaforo non raggiungerà lo zero.
3. sem_op > 0
   The value di sem_op viene aggiuntoal valore del semaforo liberando le risorse precedentemente prese..

Il programma seguente prova a mostrare come usare i semafori per implementare il buffer dell'esempio precedente. Noi creeremo 5 processi chiamati W (scrittori) e un processo chiamato R lettore. Ogni processo W

prova a prendere il possesso del buffer bloccandola con un semaforo e se il buffer non è pieno mette un elemento al suo interno e rilascia la risorsa. Il lettore prova a bloccare la risorsa, poi se il buffer non è vuoto prende un elemento e rilascia la risorsa (attenzione a non provocare deadlock).

Leggere e scrivere nel buffer sono solo operazioni virtuali, questo perché, come abbiamo visto nel precedente articolo ogni processo ha il proprio spazio di memoria e a questo non può accedere nessun altro processo. Cosi senza avere la memoria condivisa è impossibile che i vari processi possano accedere allo stesso buffer ma ognuno accederà alla propria copia del buffer. Vedremo in seguito come condividere la memoria ma procediamo per gradi.

Perchè abbiamo bisogno di tre semafori? Uno ci servirà per la mutua esclusione dal buffer e avrà valori comprei tra 0 e 1 mentre gli altri due gestiranno le condizioni di overflow e underflow. Un solo semaforo non potrebbe gestire il tutto.

Chiarifichiamo il problema: con un semaforo chiamato O, il cui valore rappresenta il numero di posizioni vuote all'interno del buffer, ogni volta che un processo W (scrittore) mette qualcosa all'interno del buffer verrà decrementato il valore del semaforo di uno fino a quando il valore del semaforo raggiungerà il valore zero cioè buffer pieno inoltre il lettore aumenterà di uno il semaforo ogniqualvolta leggerà un dato (una posizione vuota si è aggiunta). Avremo bisogno poi di un semaforo U che gestisca l' underflow che rappresenta il numero di elementi all'interno del buffer in questo caso il lettore decrementerà di uno il semaforo (vi è un elemento in meno) mentre lo scrittore aggiungerà un 1 al valore del semaforo ogni volta che metterà un elemento all'interno del buffer.

La condizione di overflow è rappresentata dall'impossibilità di aggiungere elementi all'interno del buffer mentre la condizione di underflow è rappresentata dal buffer vuoto e dall'impossibilità di leggere un ulteriore elemento.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <linux/types.h>
#include <linux/ipc.h>
#include <linux/sem.h>

int main(int argc, char *argv[])
{
  /* IPC */
  pid_t pid;
  key_t key;
  int semid;
  union semun arg;
  struct sembuf lock_res = {0, -1, 0};
  struct sembuf rel_res = {0, 1, 0};
  struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};
  struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};

  /* Other */
  int i;

  if(argc < 2){
    printf("Usage: bufdemo [dimensione]\n");
    exit(0);
  }

  /* Semaphores */
  key = ftok("/etc/fstab", getpid());

  /* Create a semaphore set with 3 semaphore */
```

```
  semid = semget(key, 3, 0666 | IPC_CREAT);

  /* Initialize semaphore #0 to 1 - Resource controller */
  arg.val = 1;
  semctl(semid, 0, SETVAL, arg);

  /* Initialize semaphore #1 to buf_length - Overflow controller */
  /* Sem value is 'free space in buffer' */
  arg.val = atol(argv[1]);
  semctl(semid, 1, SETVAL, arg);

  /* Initialize semaphore #2 to buf_length - Underflow controller */
  /* Sem value is 'elements in buffer' */
  arg.val = 0;
  semctl(semid, 2, SETVAL, arg);

  /* Fork */
  for (i = 0; i < 5; i++){
    pid = fork();
    if (!pid){
      for (i = 0; i < 20; i++){
    sleep(rand()%6);
    /* Try to lock resource - sem #0 */
    if (semop(semid, &lock_res, 1) == -1){
      perror("semop:lock_res");
    }
    /* Lock a free space - sem #1 / Put an element - sem #2*/
    if (semop(semid, &push, 2) != -1){
      printf("---> Process:%d\n", getpid());
    }
    else{
      printf("---> Process:%d  BUFFER FULL\n", getpid());
    }
    /* Release resource */
    semop(semid, &rel_res, 1);
      }
      exit(0);
    }
  }

  for (i = 0;i < 100; i++){
    sleep(rand()%3);
    /* Try to lock resource - sem #0 */
    if (semop(semid, &lock_res, 1) == -1){
      perror("semop:lock_res");
    }
    /* Unlock a free space - sem #1 / Get an element - sem #2 */
    if (semop(semid, &pop, 2) != -1){
      printf("<--- Process:%d\n", getpid());
    }
    else printf("<--- Process:%d  BUFFER EMPTY\n", getpid());
    /* Release resource */
    semop(semid, &rel_res, 1);
  }

  /* Destroy semaphores */
  semctl(semid, 0, IPC_RMID);

  return 0;
}
```

Commentiamo le parti più interessanti di codice:

```
struct sembuf lock_res = {0, -1, 0};
```

```
struct sembuf rel_res = {0, 1, 0};
struct sembuf push[2] = {1, -1, IPC_NOWAIT, 2, 1, IPC_NOWAIT};
struct sembuf pop[2] = {1, 1, IPC_NOWAIT, 2, -1, IPC_NOWAIT};
```

Queste quattro righe specificano le azioni che possiamo effettuare sul nostro insieme di semafori. Le prime due sono azioni singole mentre le altre sono azioni doppie.La prima azione cerca di bloccare la risorsa decrementando di uno il valore del semaforo numero 0 e la politica adottata se la risorsa è occupata è nessuna cioè il processo aspetterà. La medesima azione ma all'incontrario, viene effettuata dalla rel_res che incrementa di uno il valore del semaforo.

Le azioni push e pop sono un poco speciali. Sono infatti un array di due azioni la prima sul semaforo numero 1 e la seconda sul semaforo numero 2. Mentre il primo viene incrementto il secondo viene decrementato e viceversa. Inoltre la politica non è piu di attesa sul semaforo ma l' IPC_NOWAIT forza il processo a proseguire nell'esecuzione se la risorsa è occupata.

```
/* Initialize semaphore #0 to 1 - Resource controller */
arg.val = 1;
semctl(semid, 0, SETVAL, arg);

/* Initialize semaphore #1 to buf_length - Overflow controller */
/* Sem value is 'free space in buffer' */
arg.val = atol(argv[1]);
semctl(semid, 1, SETVAL, arg);

/* Initialize semaphore #2 to buf_length - Underflow controller */
/* Sem value is 'elements in buffer' */
arg.val = 0;
semctl(semid, 2, SETVAL, arg);
```

Qui inizializziamo i valori dei semafori. Il semaforo numero zero prende il valore 1 poichè gestisce l'accesso in mutua esclusione alla risorsa, il secondo prende la lunghezza del buffer e il terzo zero poichè non vi sono elementi nel buffer alla partenza.

```
/* Try to lock resource - sem #0 */
if (semop(semid, &lock_res, 1) == -1){
  perror("semop:lock_res");
}
/* Lock a free space - sem #1 / Put an element - sem #2*/
if (semop(semid, &push, 2) != -1){
  printf("---> Process:%d\n", getpid());
}
else{
  printf("---> Process:%d  BUFFER FULL\n", getpid());
}
/* Release resource */
semop(semid, &rel_res, 1);
```

Il processo W prova a bloccare la risorsa attraverso l'azione lock_res . se questo va a buon fine allora effettua un'azione push. Dopo di questo rilascia la risorsa.

```
/* Try to lock resource - sem #0 */
if (semop(semid, &lock_res, 1) == -1){
  perror("semop:lock_res");
}
/* Unlock a free space - sem #1 / Get an element - sem #2 */
if (semop(semid, &pop, 2) != -1){
  printf("<--- Process:%d\n", getpid());
}
else printf("<--- Process:%d  BUFFER EMPTY\n", getpid());
/* Release resource */
```

```
semop(semid, &rel_res, 1);
```

Il comportamento del processo R è simile a quello dei processi W: blocca la risorsa esegue un azione pop sul semafori 1 e 2 e infine rilascia la risorsa.

In the next article we will speak about message queues, another structure for the InterProcess Communication and synchronisation. As always if you write something simple using what you learned from this article send it to me, with your name and your e−mail address, I will be happy to read it. Good work!

Recommended readings

- Silberschatz, Galvin, Gagne, **Operating System Concepts – Sixth Edition**, Wiley&Sons, 2001
- Tanenbaum, WoodHull, **Operating Systems: Design and Implementation – Second Edition**, Prentice Hall, 2000
- Stallings, **Operating Systems – Fourth Edition**, Prentice Hall, 2002
- Bovet, Cesati, **Understanding the Linux Kernel**, O'Reilly, 2000
- The Linux Programmer's Guide: http://www.tldp.org/LDP/lpg/index.html
- Linux Kernel 2.4 Internals http://www.tldp.org/LDP/lki/lki−5.html

2005−05−04, generated by lfparser_pdf version 2.51