Corso di Programmazione Concorrente

Semafori

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Semafori vs Semafori di Dijkstra

- Rispetto alla versione presentata astrattamente, le chiamate di sistema Linux/UNIX sui semafori
  - lavorano su un array di semafori e non su un singolo semaforo per motivi di efficienza
  - permettono di specificare nelle chiamate stesse di quanto incrementare/decrementare il valore del semaforo
    - semantica "*wait-for-zero*" se si specifica 0
  - … e tanti altri dettagli ancora …

Semafori: Chiamate di Sistema

- `semget()`

  per ottenere un *array* di semafori
- `semop()`

  per eseguire le normali operazioni sui semafori corrispondenti alle primitive di sincronizzazione
- `semctl()`

  per varie operazioni di controllo, inclusa la deallocazione di un vettore di semafori

`ftok()`: Chiavi per IPC

```
#include <sys/types.h>
#include <sys/ipc.h>

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

- alcune chiamate di sistema per IPC, usano chiavi per identificare univocamente le risorse allocate
- le chiavi possono essere ottenute con `ftok()` e risultano univocamente associate ai due parametri:
  - un file esistente di percorso assoluto `pathname`
  - un identificativo di progetto `proj_id`
- restituisce `-1` in caso di errore

semget() (1)

man 2 semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

- viene creato un array di `nsems` semafori se
  - `key` è `IPC_PRIVATE`
  - `key` non è associato a nessun vettore di semafori preesistenti e `semflg` è `IPC_CREAT`
- altrimenti restituisce un identificatore intero univocamente associato ad un array preesistente, oppure -1 in caso di errore

semget() (2)

man 2 semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

- semflg contiene alcuni flag + 9 bit di permessi
  - flag
    - IPC_CREAT
    - IPC_EXCL (creazione in esclusiva)

semget() (3)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

man 2 semget

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

- semflg contiene anche 9 bit di permessi
  - set di diritti identico a quello usato per i file

- lettura
  - per conoscerne il valore
- scrittura
  - per aggiornarne il valore
- esecuzione
  - nessun significato

semop()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop( int semid,
           struct sembuf *sops,
           unsigned nsops)
```

`man 2 semop`

- esegue `nsops` operazioni sul vettore di semafori `semid`
- ciascuna operazione è specificata da una struttura dati `sembuf`
- restituisce `0` in caso di successo e `-1` in caso di errore

semop(): Struttura sembuf

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

- ciascuna operazione è specificata da una struttura `sembuf` contenente almeno:

```
short sem_num;  /* numero di semaforo: primo = 0 */
short sem_op;   /* operazione sul semaforo */
short sem_flg;  /* flag */
```

- `sem_flg` è composto dai flag
  - `IPC_NOWAIT`
    - rende la chiamata non bloccante causando fallimenti della chiamata anziché sospensioni ed attese passive
  - `IPC_UNDO`
    - se un processo esegue `exit()` viene eseguito l'*undo* di tutte le modifiche che ha apportato al vettore di semafori

semop(): Operazioni

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

Il campo `sem_op` determina la semantica

- `> 0` `sem_op` viene sommato al valore del semaforo
  - rilascio della risorsa protetta da questo semaforo
- `< 0` il semaforo viene decrementato del valore assoluto di `sem_op`, il processo chiamante viene sospeso in attesa passiva che il semaforo torni verde
  - richiesta della risorsa protetta da questo semaforo
- `= 0` Il processo chiamante viene sospeso sino a quando il semaforo torna a 0 (semantica *wait-for-zero*)

semctl()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl( int semid,
            int semnum,
            int cmd,
            union semun arg)
```

`man 2 semctl`

- esegue operazioni di controllo sul vettore di semafori `semid`
- `cmd` specifica quale operazione eseguire
- restituisce un valore non negativo dipendente da `cmd`, `-1` in caso di errore

semctl(): Union semun

```
int semctl(int semid, int semnum, int cmd, union semun arg)
```

- ciascuna operazione è specificata da cmd ed utilizza una struttura unione
  - su alcune installazioni va dichiarata esplicitamente!

```
union semun {
  int val;                    /* valore per SETVAL */
  struct semid_ds *buf;       /* buffer per IPC_STAT,IPC_SET */
  unsigned short int *array;  /* array per GETALL,SETALL */
  struct seminfo *__buf;      /* buffer per IPC_INFO */
};
```

semctl(): Comandi

```
int semctl(int semid, int semnum, int cmd, union semun arg)
```

Valori permessi per `cmd`:

- `IPC_RMID` rimuove il vettore di semafori
- `GETALL` restituisce il valore dei semafori dentro `arg.array`
- `GETVAL` restituisce il valore del semaforo `semnum`
- `GETZCNT` riporta il numero di processi in attesa del semaforo `semnum`
- `SETALL` setta tutti i semafori del vettore usando `arg.array`
- `SETVAL` setta il valore del semaforo `semnum` a `arg.val`
- `GETNCNT`…
- `GETPID`…

```
man 2 semctl
```

Allocare e Deallocare Semafori

```
int binary_semaphore_allocation(key_t key,
                                int sem_flags)
{
  return semget(key,1,sem_flags);
}

/* restituisce -1 in caso di errore */
int binary_semaphore_deallocate(int semid)
{
  union semun useless;
  return semctl(semid,1,IPC_RMID,useless);
}
```

Inizializzare Semafori

```
int binary_semaphore_initialize(int semid)
{
  union semun argument;
  unsigned short values[1];
  values[0] = 1; // Initial Value
  argument.array = values;
  return semctl (semid,0,SETALL,argument);
}
```

Eseguire una P

```
/*  Attende, eventualmente con una sospensione del
    processo chiamante in attesa passiva, che il
    semaforo torni positivo. Quindi lo decrementa
*/

int binary_semaphore_P(int semid)
{
  struct sembuf operations[1];
  /* Utilizziamo il primo ed unico semaforo */
  operations[0].sem_num = 0;
  //L'op. decrementa il valore del semaforo
  operations[0].sem_op = -1;
  /* UNDO automatico all'uscita */
  operations[0].sem_flg = SEM_UNDO;

  return semop(semid,operations,1);
}
```

Eseguire una V

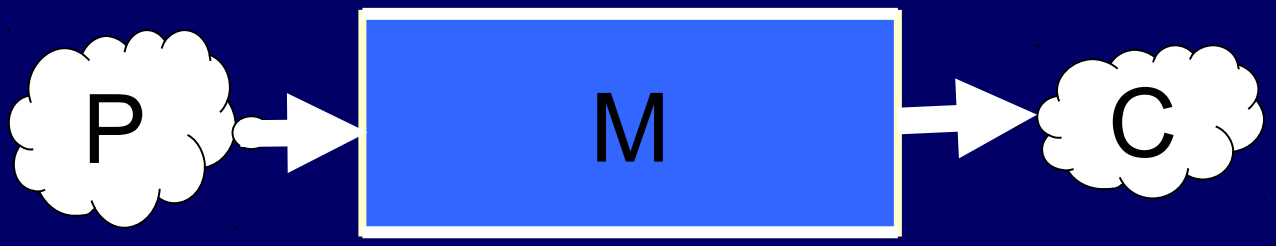
```
/* Incrementa il semaforo, ritorna immediatamente, ma
   risveglia eventuali processi in attesa */

int binary_semaphore_V(int semid)
{
  struct sembuf operations[1];
  /* Utilizziamo il primo ed unico semaforo */
  operations[0].sem_num = 0;
  // L'op. incrementa il valore del semaforo
  operations[0].sem_op = 1;
  /* UNDO automatico all'uscita uscita  */
  operations[0].sem_flg = SEM_UNDO;

  return semop(semid,operations,1);
}
```

Problema Produttore / Consumatore

- Un processo P (produttore) deve continuamente inviare messaggi ad un altro processo C (consumatore) che li elabora nello stesso ordine in cui li ha ricevuti
- Viene utilizzato un buffer di scambio
  - P scrive i messaggi nel buffer
  - C legge i messaggi dal buffer



Produttore / Consumatore: Sincronizzazioni Necessarie

- I due processi devono essere opportunamente sincronizzati contro queste eventualità:
  - C legge un messaggio senza che P ne abbia depositato alcuno
  - P sovrascrive un messaggio senza che C sia riuscito a leggerlo
  - C legge lo stesso messaggio più di una volta

- Essenzialmente bisogna sincronizzarsi su due eventi a cui associamo due diversi semafori binari
  - DEPOSITATO:
    - ad 1 se e solo se un messaggio è stato depositato ed è prelevabile
  - PRELEVATO:
    - ad 1 se e solo se il buffer è vuoto e pronto ad accogliere un nuovo messaggio

Un Produttore / Un Consumatore

```
concurrent program COOPERAZIONE;
type     messaggio=…;
var      M: messaggio;
         DEPOSITATO, PRENOTATO:semaforo_binario;
```

```
concurrent procedure PROD
loop begin
    <produci un messaggio in M>
    P(PRELEVATO);
    BUFFER ← M;
    V(DEPOSITATO);
end
```

```
concurrent procedure CONS
loop begin
    P(DEPOSITATO);
    M ← BUFFER;
    V(PRELEVATO);
    <consuma il messaggio in M>
end
```

```
begin
    INIZ_SEM(PRELEVATO,1); INIZ_SEM(DEPOSITATO,0);
    cobegin PROD || CONS coend
end
```

1producer1consumer1buffer.c (1)

```
…include omessi…
#define FILENAME "1producer1consumer1buffer.c"
#define RANGE    16
#define TIME     30
#define DTIME     3

#define PRELEVATO    0
#define DEPOSITATO   1

void releaseAll(int,int);
int main(int argc, char **argv) {
  int *buffer;
  int M;
  int bufferid, semid;
  int delay, ptime=TIME;
  int p=0;

  pid_t pid;
  pid_t cons_pid;
```

1producer1consumer1buffer.c (2)

```
…
 struct sembuf sb;
 union semun {
    int val;  /* value for SETVAL */
    // buffer for IPC_STAT, IPC_SET
    struct semid_ds *buf;
    // array for GETALL, SETALL
    unsigned short int *array;
    // buffer for IPC_INFO
    struct seminfo *__buf;
 } su;
 sb.sem_flg=0;
…
```

1producer1consumer1buffer.c (3)

```
…/* allocate shared memory segments */
if ((bufferid=shmget(ftok(FILENAME,'B'),1,IPC_CREAT|0666))==-1){
  perror("shmget() for buffer");
  releaseAll(bufferid,semid); exit(-1);
}

/* allocate semaphores */
if ((semid=semget(ftok(FILENAME,'S'),2,IPC_CREAT|0666))==-1){
  perror("semget()");
  releaseAll(bufferid,semid); exit(-4);
}

/* initialize semaphores and shared segments */
su.val=0;
if (semctl(semid,DEPOSITATO,SETVAL,su)==-1) {
  perror("semctl() to initialize DEPOSITATO"); exit(-22);
}
su.val=1;
if (semctl(semid,PRELEVATO,SETVAL,su)==-1) {
  perror("semctl() to initialize PRELEVATO");  exit(-23);
}…
```

1producer1consumer1buffer.c (4)

```
/* fork producer  */
 pid = fork();

 switch(pid) {
 case -1: //fork fallita
   exit(-5);
   break;
 case 0:  /* PRODUCER */ …codice producer a seguire
   return 0;
 }

 /* fork consumer */
 pid = cons_pid = fork();

 switch(pid) {
 case -1: //fork fallita
   exit(-5);
   break;
 case 0: /* CONSUMER */ …codice consumer a seguire
   return 0;
 }
 …
```

1producer1consumer1buffer.c (5)

```
… // transitorio finale
  wait(NULL);  // wait producer ...
  // wait that consumer finishes ...
  sb.sem_num=PRELEVATO; sb.sem_op=-1;
  if (semop(semid, &sb,1)==-1) {// P(PRELEVATO)
    perror("semop() waiting for producers");
    exit(-19);
  }

  kill(cons_pid,SIGKILL); // ... and Kill consumer
  wait(NULL);

  /* dealloca risorse */
  releaseAll(bufferid,semid);
  return 0;
}
```

1producer1consumer1buffer.c (6)

```
case 0: /* PRODUCER */
   if ((int)(buffer=shmat(bufferid,NULL,0)) == -1) {
     perror("producer: shmat() for buffer"); exit(-7);   }
   /* PRODUCING */
   srandom(time(NULL));
   // produci un messaggio in M
   while (ptime>0) {
     sb.sem_num=PRELEVATO; sb.sem_op=-1;
     if (semop(semid, &sb,1)==-1) {  /* P(PRELEVATO) */
       perror("semop() producer P(PRELEVATO)"); exit(-9);  }
     M = (int)(random() % RANGE);

     buffer[0] = M;                  /* PRODUCI */

     printf("Producer produces: %d\n",M);

     sb.sem_num=DEPOSITATO; sb.sem_op= 1;
     if (semop(semid, &sb,1)==-1) {  /* V(DEPOSITATO) */
        perror("semop() producer V(DEPOSITATO)"); exit(-12); }
     delay = (int)(random() % DTIME) / 2 + 1;
     sleep(delay);
     ptime -=delay;
   }
   printf("Producer exits\n");
   shmdt(buffer);
   return 0;
"
```

1producer1consumer1buffer.c (7)

```
case 0: /* CONSUMER */
   if ((int)(buffer=shmat(bufferid,NULL,0)) == -1) {
     perror("consumer: shmat() fo buffer"); exit(-7);  }

   /* CONSUMING */
   srandom(time(NULL));
   // consuma un messaggio in M
   while (1) {
     sb.sem_num=DEPOSITATO; sb.sem_op=-1;
     if (semop(semid, &sb,1)==-1) {  /* P(DEPOSITATO) */
      perror("semop() consumer P(DEPOSITATO)"); exit(-9); }

     M = buffer[0];                    /* CONSUMA */

     printf("Consumer consumes: %d\n",M);

     sb.sem_num=PRELEVATO; sb.sem_op= 1;
     if (semop(semid, &sb,1)==-1) {  /* V(PRELEVATO) */
      perror("semop() consumer V(PRELEVATO)"); exit(-12); }
     delay = (int)(random() % DTIME) / 2 + 2;
     sleep(delay);
   }
   printf("Consumer exits\n");
   shmdt(buffer);
   return 0;
```

…

Esempio di Esecuzione

```
$ ./1producer1consumer1buffer
Producer produces: 1
Consumer consumes: 1
Producer produces: 15
Consumer consumes: 15
Producer produces: 13
Consumer consumes: 13
Producer produces: 14
Consumer consumes: 14
Producer produces: 6
…
Consumer consumes: 13
Producer produces: 7
Consumer consumes: 7
Producer produces: 4
Producer exits
Consumer consumes: 4
$
```

Esercizi

Esercizio: Realizzare un programma che risolva il problema di cooperazione con molti produttori e molti consumatori ed un buffer capace di contenere un solo messaggio.

Esercizio: Con riferimento all'esercizio precedente verificare che se un processo viene ucciso durante la sua sezione critica si verifica uno stallo. Risolvere il problema utilizzando `SEM_UNDO` con la chiamata `semop()`. Ripetere l'esperimento di prima e verificare che la nuova versione "resiste" alla terminazione anomala di alcuni processi produttori o consumatori.

Esercizio: Eseguire e stampare il prodotto di due matrici in parallelo: ciascun elemento della matrice risultante è calcolato da un processo dedicato.