Semafori

Un semaforo è un contatore che viene impiegato per consentire l'accesso a risorse condivise da più processi.

Per ottenere una risorsa condivisa un processo deve compiere le seguente azioni:

1. leggere il valore del semaforo associato alla risorsa

2. se il valore del semafore è positivo (>0), allora il processo può utilizzare la risorsa, decrementando il valore del semaforo di 1

3. se il valore del semafore è 0, il processo viene posto in stato attesa fino a quando il semaforo assumerà un valore positivo.
   Quando il processo viene riattivato, ritornerà al passo 1.

Quando un processo termina di utilizzare la risorsa condivisa, il semaforo viene incrementato di 1.

Se ci sono processi in attesa sul semaforo, questi verranno riattivati.

Una forma comune di semaforo è il semaforo "binario", il cui valore è inizializzato a 1. In genere, comunque, i semafori possono essere inizializzati con qualsiasi valore positivo.

Le primitive `p` e `v`

I semafori sono gestiti mediante le primitive `p` e `v` (anche: `wait` e `signal`):

```
p(sem)  /* or wait(sem) */

     if( sem != 0 )
          decrement sem by 1;
     else
          wait until sem becomes non-zero;


v(sem)  /* or signal(sem) */

     if( queue of waiting processes not empty )
          restart first process in queue;
     else
          increment sem by 1;
```

Formalmente, i semafori godono della seguente proprietà "invariante":

```
     valore iniziale del semaforo +
     numero di operazioni v -
     numero di operazioni p completate ≥ 0
```

I semafori possono essere impiegati per assicurare la mutua esclusione, dove soltanto un processo può eseguire una particolare regione di codice in un particolare istante di tempo.

Esempio:

```
     p(sem);
     ..........
     ..........         /* sezione critica */
     ..........
     v(sem);
```

Assumendo che il valore iniziale di `sem` sia 1, soltanto un processo potrà eseguire il codice nella "sezione critica", essendo:

```
  num di operazioni p completate - num di operazioni v ≤ 1
```

Semafori in Unix System V

Unix System V implementa i semafori, introducendo alcune "complicazioni":

1. un semaforo non è semplicemente un singolo intero non-negativo, ma un **insieme** (array) di valori. Quando si crea un semaforo è necessario specificare il numero di tali valori.

2. la creazione di un semaforo (mediante `semget`) è indipendente dalla sua inizializzazione (mediante `semctl`). Questo non consente di creare e inizializzare un semaforo in modo **atomico**.

3. come tutte le strutture SV-IPC, un semaforo continua a esistere anche quando non c'è nessun processo che lo usa.

Il kernel associa a ciascun semaforo la seguente struttura:

```
struct semid_ds {
  struct ipc_perm sem_perm;    /* info su proprietari
                                    permessi, etc */
  struct sem    *sem_base;     /* ptr a primo semaforo */
  ushort        sem_nsems;     /* numero semafori */
  time_t        sem_otime;     /* tempo ultima semop() */
  time_t        sem_ctime;     /* tempo ultima modifica */
};
```

Il campo `sem_base` è il puntatore all'inizio dell'array di semafori (strutture `sem`).

```
struct sem {
  ushort   semval;    /* valore del semaforo (≥ 0) */
  pid_t    sempid;    /* pid ultima operazione */
  ushort   semncnt;   /* # processi in attesa di
                           semval > current_value */
  ushort   semzcnt    /* # processi in attesa semval=0 */
};
```

La primitiva **semget**

La primitiva `semget` (analoga alla `msgget`) consente di "aprire" un semaforo.

Uso:

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

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

Il significato dei parametri è il seguente:

`key`	è la chiave del semaforo che si intende aprire
`nsems`	specifica il numero di semafori
`flag`	specifica i diritti di accesso (come in `msgget`)

Il valore di ritorno di `semget` è l'identificatore del semaforo aperto (oppure -1 in caso di errore).

Quando si crea un semaforo nuovo, alcuni campi della struttura `semid_ds` sono inizializzati come segue:

- nella struttura ipc_perm si scrivono informazioni relative al proprietario, diritti di accesso (contenuti in `flag`), etc.
- `sem_otime` (ultima operazione `semop`) è posto a 0
- `sem_ctime` (ultima modifica) è posto uguale al tempo corrente
- `sem_nsems` è posto uguale a `nsems`

Nel caso in cui si crei un semaforo nuovo, il parametro `nsems` deve essere specificato. Se, invece, si apre un semaforo già creato si può usare `nsems=0`.

La primitiva `semctl`

La `semctl` viene impiegata per svolgere varie funzioni:

Uso:

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

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

Il significato dei parametri è il seguente:

`semid` è l'identificatore del semaforo

semnum indica il numero (indice) del semaforo su cui realizzare l'operazione indicata dal parametro cmd (serve solo per alcuni comandi).
Può assumere valori compresi tra `0` e `nsems-1` (inclusi).

`cmd` specifica l'azione da eseguire, fra le 10 possibili (per esempio: `GETVAL`, `SETVAL`, `GETPID`, etc.)

`arg` è l'unione di tre campi, ciascuno corrispondente ad uno dei tre tipi di comandi disponibili

La unione `semun` è definita come segue:

```
union semun {
  int                 val;       /* per SETVAL */
  struct semid_ds     *buf;      /* per IPC_STAT e IPC_SET */
  ushort              *array;    /* per GETALL e SETALL */
}
```

Per tutti i comandi `GET` (eccetto `GETALL`) la funzione `semctl` restituisce il valore richiesto. Altrimenti ritorna 0.

I codici di comando per `semctl`

I comandi realizzati dalla primitiva `semctl` si dividono in tre gruppi:

1. Funzioni standard IPC

`IPC_STAT`	copia le informazioni riguardanti il semaforo (memorizzate in `semid_ds`) in `arg.buf`
`IPC_SET`	consente di modificare il valore di alcuni parametri associati al semaforo (user-id, group-id, modalità di accesso) E' consentita solo al proprietario della coda o al super-user.
`IPC_RMID`	rimuove il semaforo dal sistema (consentito solo al proprietario e il super-user)

2. Operazioni su singolo semaforo

`GETVAL`	restituisce il valore del semaforo (`semval`) avente indice `semnum`
`SETVAL`	scrive il valore `arg.val` nel semaforo `semnum` (campo `semval`)
`GETPID`	restituisce il campo `sempid` per `semnum`
`GETNCNT`	restituisce il campo `semncnt` per `semnum`
`GETZCNT`	restituisce il campo `semzcnt` per `semnum`

3. Operazioni che coinvolgono tutti i semafori dell'array

`GETALL`	restituisce tutti i valori del semaforo, memorizzandoli nell'area puntata da `arg.array`
`SETALL`	scrive nei campi `semval` di tutti i semafori i valori memorizzati in `*arg.array`

Esempio (inizializzazione di un semaforo)

```
/***
*
*    pv.h -- header file
*
***/

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

#include <errno.h>

extern int errno;

#define SEMPERM      0600
#define TRUE         1
#define FALSE   0



/***
*
*    initsem -- semaphore initialization
*
***/

#include "pv.h"

initsem( key_t semkey )
{
int status=0, semid;

if(semid=semget(semkey,1,SEMPERM|IPC_CREAT|ICP_EXCL)==-1){
  if(errno == EEXIST)     /* existing semaphore */
    semid = semget(semkey,1,0);
  }
else  /* if created */
  status = semctl(semid,0,SETVAL,1);


if( semid == -1 || status == -1 ) {
  perror("initsem failed");
  return(-1);
  }
else
  return(semid);  /* all okay */

} /* end of initsem */
```

La primitiva `semop`

La primitiva `semop` consente di realizzare atomicamente varie operazioni su un array di semafori.

Uso:

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

int semop(int semid,struct sembuf semoparray[],size_t nops)
```

Dove `semoparray` punta ad un array di operazioni semaforiche. La struttura `sembuf` è definita come segue:

```
struct sembuf {
  ushort sem_num;    /* indice in [0,nsem-1] */
  short  sem_op;     /* operazione */
  short  sem_flg;    /* IPC_NOWAIT, SEM_UNDO */
  };
```

`nops` specifica il numero di operazioni (elementi) dell'array.

L'operazione su ciascun elemento dell'insieme di semafori è specificata nel campo `sem_op`, mentre `sem_num` specifica l'elemento che interessa.

`sem_op` può essere negativo, positivo, o uguale a zero:

sem_op<0: è una generalizzazione all'operazione `p`.

sem_op>0: corrisponde all'operazione `v`:

sem_op=0: il processo attende fino a quando il valore del semaforo diventa 0.

La primitiva realizza le sue funzioni in modo atomico. Se non riesce a realizzare una funzione su un singolo semaforo, l'intera operazione sull'array fallisce.

La primitiva **semop** (2)

Descriviamo in dettaglio il comportamento della primitiva `semop`.

Caso 1: `sem_op<0`

Corrisponde al caso in cui un processo intende prendere possesso di una risorsa.

Se il valore del semaforo è maggiore o uguale a `abs(sem_op)`, cioè la risorsa è disponibile, si sottrae al semaforo `abs(sem_op)` (il risultato è sempre positivo).

Se il semaforo è minore di `abs(sem_op)`, cioè la risorsa non è disponibile:

a) Se `IPC_NOWAIT` è settato, ritorna immediatamente;
b) Se `IPC_NOWAIT` non è settato, il valore semncnt del semaforo viene incrementato, e il processo chiamante è sospeso finché una delle seguenti condizioni non si verifica:

  i) Il valore del semaforo diventa maggiore o uguale di `abs(sem_op)`, cioè qualche processo ha rilasciato la risorsa.
  In questo caso `semncnt` viene decrementato, e `abs(sem_op)` è sottratto dal semaforo.
  ii) Il semaforo è rimosso dal sistema (in questo caso la funzione ritorna il valore `ERMID`)
  iii) Il processo cattura un segnale e il signal handler provoca il ritorno. In questo caso `semncnt` viene decrementato

Caso 2: `sem_op>0`

Corrisponde al caso in cui un processo rilascia una risorsa.

Il valore di `sem_op` è aggiunto a quello del semaforo.

La primitiva `semop` (3)

Caso 3: `sem_op=0`

Il processo è posto in stato di attesa fino a quando il semaforo diventa uguale a 0.

Se il semaforo è 0, la funzione ritorna immediatamente.

Se il semaforo è diverso da 0:

a) Se `IPC_NOWAIT` è settato, ritorna immediatamente;
b) Se `IPC_NOWAIT` non è settato, il valore semzcnt del semaforo viene incrementato, e il processo chiamante è sospeso finché una delle seguenti condizioni non si verifica:

   i) Il valore del semaforo diventa uguale a 0. In questo caso `semzcnt` è decrementato.
   ii) Il semaforo è rimosso dal sistema (in questo caso la funzione ritorna un valore `ERMID`)
   iii) Il processo cattura un segnale e il signal handler provoca il ritorno. In questo caso `semzcnt` viene decrementato

Il flag `SEM_UNDO`

Un processo che termina mentre è in possesso di una risorsa (mediante un semaforo) costituisce un problema.

Quando si specifica il flag `SEM_UNDO` nel campo `sem_flg`, il kernel gestisce una variabile (`semadj`) che, opportunamente incrementata e decrementata, consente di risolvere questo problema.

Esempio (implementazione di `p` e `v`)

```
/***
*
*    p.c -- semaphore p operation
*
***/

#include "pv.h"

p( int semid )
{
struct sembuf p_buf;

p_buf.semnum = 0;
p_buf.sem_op = -1;
p_buf.sem_flg = SEM_UNDO;

if( semop( semid, &p_buf, 1 ) == -1 ) {
  perror("p(semid) failed");
  exit(1);
  }
else
  return(0);

} /* end of p */
```

```
/***
*
*    v.c -- semaphore v operation
*
***/

#include "pv.h"

v( int semid )
{
struct sembuf v_buf;

v_buf.semnum = 0;
v_buf.sem_op = 1;
v_buf.sem_flg = SEM_UNDO;

if( semop( semid, &p_buf, 1 ) == -1 ) {
  perror("v(semid) failed");
  exit(1);
  }
else
  return(0);

} /* end of v */
```

Esempio (mutua esclusione)

```
/***
*
*    testsem -- test semaphore routine
*
***/

#include "pv.h"

main()
{
key_t semkey = 0200;

if( fork() == 0 )
     handlesem(semkey);
if( fork() == 0 )
     handlesem(semkey);
if( fork() == 0 )
     handlesem(semkey);

} /* end of main */



/*--- handlesem ---*/

handlesem( key_t skey )
{
int semid, pid;

pid = getpid();

if( semid = initsem(skey) < 0 )
  exit(1);

printf("Process %d before critical section\n", pid);
p(semid)
printf("Process %d in critical section\n", pid);
sleep(10);   /** in real life do something useful **/
printf("Process %d leaving critical section\n", pid);
v(semid);
printf("Process %d leaving critical section\n", pid);

exit(0);

} /* end of handlesem */
```

Esempio (continua)

Un'esecuzione del programma precedente dà:

```
$ testsem

Process 799 before critical section
Process 800 before critical section
Process 801 before critical section

Process 799 in critical section
Process 799 leaving critical section
Process 799 exiting

Process 800 in critical section
Process 800 leaving critical section
Process 800 exiting

Process 801 in critical section
Process 801 leaving critical section
Process 801 exiting

$ _
```