SEMAFORI

Le chiamate di sistema dei semafori consentono ai processi di sincronizzare l'esecuzione facendo un insieme di operazioni atomiche su un insieme di semafori.
Tali chiamate sono una generalizzazione delle operazioni P e V di Dijkstra, nel senso che varie operazioni possono essere fatte simultaneamente e le operazioni di incremento e decremento possono essere fatte per valori superiori a 1. Il kernel esegue tutte le operazioni in modo atomico: nessun processo può accedere ai valori del semaforo finché sono state fatte tutte le operazioni. Se il kernel non esegue tutte le operazioni non ne fa nessuna.
Un semaforo nel System V UNIX consiste dei seguenti elementi:

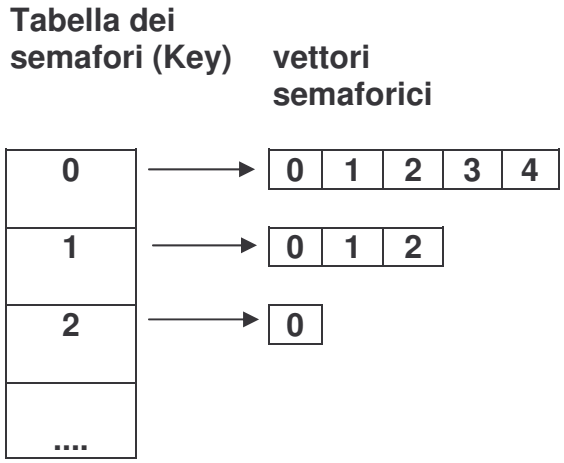
- **il valore del semaforo**
- **l'ID di processo dell'ultimo processo che ha manipolato il semaforo**
- **il numero di processi che attendono che il valore del semaforo aumenti**
- **il numero di processi in attesa che il valore del semaforo diventi 0**

Quando si fa uso dei semafori, per esempio per bloccare delle risorse, si gestiscono vettori (o insiemi) di semafori, in modo che, se serve bloccare più risorse in un "colpo solo", lo si può fare con un'unica istruzione. Nei nostri esempi ci limiteremo a un vettore composto di un solo semaforo.
Le chiamate di sistema sono simili a quelle dei messaggi e della memoria condivisa.
Per creare e ottenere l'accesso a un insieme (vettore) di semafori si usa la seguente:

```
id=semget(Key,cont,flag)
```

- **id è il descrittore dell'insieme di semafori (un intero)**
- **Key è, come per la memoria condivisa e per i messaggi, una chiave numerica**
- **cont è il numero di strutture semaforo nell'array, cioè il n. di semafori**
- **flag è un intero per assegnare i permessi e per altre cose particolari (come per la gestione dei messaggi e della memoria condivisa)**

Il kernel alloca un'entry che punta a un array di strutture semaforo con cont elementi:



I processi manipolano i semafori mediante la chiamata di sistema:

```
vecchioval=semop(id, oplist, cont)
```

- **id è il descrittore restituito da una semget**
- **oplist è un puntatore a una struttura di operazioni di semaforo. I campi sono: numero di semaforo su cui operare (cioè l'indice del vettore semafori corrispondente al semaforo che ci interessa), l'operazione, i flag. L'operazione è un numero che specifica come modificare il valore di un semaforo. C'è una miriade di possibilità, ma a noi ne interessano solo due: operazione = -1, che serve per una P di Dijkstra e operazione = 1, che serve per una V di Dijkstra.**
- **cont è la dimensione dell'array**
- **vecchioval è il valore dell'ultimo semaforo su cui si è operato nell'insieme prima dell'ultima operazione.**

Esiste un'ultima chiamata per manipolare i semafori, e serve, tra l'altro, alla loro inizializzazione (obbligatoria):

semctl (id, numero, cmd, arg)

- **id è il descrittore dell'insieme semafori**
- **numero fornisce il numero del semaforo dell'insieme da considerare**
- **cmd è un intero (di solito viene indicato come una costante) che specifica cosa fare di quei semafori. Ad es. IPC_RMID rimuove i semafori associati a id, SETALL assegna il valore di tutti i semafori secondo l'array arg.vettore, ecc.**
- **arg è una union complessa, ma, per i nostri scopi, è semplice impostarne il valore; noi lo faremo attraverso un vettore.**

Gli esempi che seguono mostrano come un processo (sem2) può bloccare una risorsa, ad esempio un file (file.dat), utilizzarla in modo che nessuno possa accedervi, per poi sbloccarla. Il blocco è fatto grazie a un semaforo. Un altro processo (sem3) che cerchi di accedere alla risorsa (il file) deve aspettare finché il semaforo diventa verde, cioè fin quando il primo processo non sblocca tale risorsa.
Il semaforo va prima inizializzato attraverso il programma sem1.
Per provare i programmi, partire prima con sem1, poi lanciare sem2, vedere quindi il contenuto del file, per poi far partire sem3. Quest'ultimo non potrà proseguire finché il semaforo diventa verde (cioè finché non si preme INVIO al sem2). Allo sblocco del file, sem3 potrà a sua volta mettere il semaforo a rosso e usare il file. Alla fine controllare il contenuto del file.

```
/*sem1*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY  75
int semid;

main()
{
      unsigned short initarray[1];

      semid=semget(SEMKEY,1,0777|IPC_CREAT);
      initarray[0]=1;
      semctl(semid,1,SETALL,initarray);
}
```

```
    /*sem2*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY  75
int semid;
struct sembuf psembuf;

main()
{     int fd;
      char ch;
      semid=semget(SEMKEY,1,0777);
      psembuf.sem_num=0;
      psembuf.sem_op=-1;
      psembuf.sem_flg=SEM_UNDO; /* SEM_UNDO rimette le cose a posto
                                   quando il processo termina */

      printf("Ora viene invocata la P() di Dijkstra. Attendere prego...\n");
      semop(semid,&psembuf,1); //invoco la P() di Dijkstra

      if ((fd=open("file,dat",1))==-1)
            fd=creat("file.dat");
      write(fd,"file aggiornato dal programma sem2\n",38);
      close(fd);
      printf("Effettuati gli aggiornamenti sul file -file.dat-\n");

      printf("Battere INVIO per continuare");

      ch=getchar();    // prima di sbloccare la risorsa file mettendo a verde
                  // il semaforo, aspetto che l'utente batta INVIO
      psembuf.sem_op=1;
      semop(semid,&psembuf,1); //invoco la V() di Dijkstra
      printf("Eseguita la V() di Dijkstra");
}


 /*sem3*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY  75
int semid;
struct sembuf psembuf;

main()
{
      int fd;
      semid=semget(SEMKEY,1,0777);
      psembuf.sem_num=0;
      psembuf.sem_op=-1;
      psembuf.sem_flg=SEM_UNDO;

      printf("Ora viene invocata la P() di dijkstra. Attendere prego ...\n");
      semop(semid,&psembuf,1); //invoco la P() di Dijkstra

      if ((fd=open("file,dat",1))==-1)
            fd=creat("file.dat");
      write(fd,"file aggiornato dal programma sem3\n",38);
      close(fd);
      printf("Effettuati gli aggiornamenti sul file -file.dat-\n");
      psembuf.sem_op=1;
      semop(semid,&psembuf,1);
}
```