CAPITOLO 6 - SINCRONIZZAZIONE TRA PROCESSI

Processi indipendenti usano primitive di sincronizzazione per coordinare le loro azioni e cooperare nella condivisione delle risorse. Il capitolo 1 ha introdotto il concetto di *semafori contatori* - il meccanismo di base per il coordinamento dei processi in PC-Xinu - ed ha fornito alcuni esempi che ne illustrano il loro uso. In uno di questi esempi, due processi sono coordinati in modo tale che il "consumatore" ricevesse ogni valore emesso dal "produttore". In un altro esempio, un insieme di processi usa un semaforo per ottenere l'accesso esclusivo a strutture dati che essi condividono.

I semafori sono inoltre le più semplici primitive, per il coordinamento tra processi, da capire ed implementare. Concettualmente, ogni semaforo *s*, consiste di un contatore di tipo intero. I processi chiamano la procedura *wait(s)* per decrementarne il valore e *signal(s)* per incrementarlo. Se tale valore diviene negativo quando un processo esegue *wait(s)*, questo viene sospeso. Un processo sospeso diverrà pronto per l'esecuzione quando una istruzione di *signal* viene chiamata. Se nessun processo chiama la procedura *signal* esso continuerà ad aspettare per sempre.

Come riesce esattamente la procedura *wait* a sospendere il suo chiamante? E' importante ricordare che sebbene si hanno processi eseguiti in pseudo-parallelismo, stiamo discutendo di un sistema in cui l'esecuzione avviene su un singolo processore. Un processo non può eseguire istruzioni in stato sospeso privando ad altri il servizio della CPU. Per esempio, l'attesa che implica il test di una locazione di memoria in un ciclo continuo è pericolosa - se la CPU spende tutto il suo tempo eseguendo un processo in stato di "waiting", nessun altro processo potrà chiamare una *signal* per sbloccarlo. Anche nei sistemi multiprocessore, le cosiddette tecniche di busy waiting possono interferire il procedere perché ogni processore è in contesa con gli altri, nell'uso della memoria o di un bus di sistema, per prelevare istruzioni o dati. Per minimizzare il sovraccarico del sistema, le primitive di sincronizzazione in PC-Xinu seguono il seguente principio:

> *I processi sospesi non eseguono istruzioni; quando tutti i processi utente sono in questo stato, il sistema non esegue codice.*

Se un sistema esegue codice quando tutti i processi utente sono in stato di waiting dipende piuttosto dall'architettura della macchina. Come molte altre, l'8088 include una istruzione di HALT per arrestare la CPU mentre tutti i processi sono in wait. Questo problema non lo prenderemo in considerazione e tratteremo il semplice caso di come evitare l'attesa attiva quando almeno un processo rimane pronto per l'esecuzione. Ricordiamo che PC-Xinu ha sempre un processo pronto - il processo NULL.

6.1 Tecniche di sincronizzazione a basso livello

Il capitolo precedente conteneva esempi di sincronizzazione tra processi in routine come *ready* e *resume*. Quando un processo esegue una di queste routine e ha bisogno di modificare strutture dati condivise come la tabella dei processi, esso deve essere sicuro che nessun altro tenti un accesso in concorrenza. Il coordinamento tra queste routine di sistema a basso livello implicano la disabilitazione degli interrupt e il privarsi dall'effettuare chiamate a *resched*. Perché non usiamo ancora questa soluzione? Disabilitare gli interrupt ha un generico effetto indesiderabile sul sistema: permette ad un solo processo l'esecuzione e ne limita le capacità di azione. Noi necessitiamo di primitive aventi scopi generali che coordino un sottoinsieme di processi senza:
- coinvolgere altri processi

- disabilitare gli interrupt dei dispositivi per lunghi periodi di tempo
- limitare le azioni dei processi in esecuzione.

Per esempio, dovrebbe essere possibile per un processo proibire cambiamenti a una struttura dati senza fermare quelli che non vogliono accedervi.

6.2 Implementazione di primitive di sincronizzazione di alto livello

L'implementazione di semafori contatori in PC-Xinu evita l'attesa attiva negando il servizio della CPU ai processi sospesi. Quando un processo è in tale stato su un semaforo, il sistema lo colloca in una lista associata al relativo semaforo. Naturalmente ognuno di essi ha la sua lista indipendente di processi. Per sospendere il processo corrente, *wait(s)* lo mette in coda sulla lista di *s* e chiama *resched* permettendo ad un altro processo di andare in esecuzione. *Signal(s)* controlla la lista associata ad *s* ogniqualvolta viene chiamata. Dopo aver controllato che la lista non sia vuota, *signal* reinizializza il processo riportandolo nella lista di processi pronti.

In quale stato dovrebbe essere un processo mentre è in attesa su un semaforo? Chiaramente non è nè corrente nè pronto perché non può usare la CPU tantomento essere eleggibile per essa. Lo stato *suspended*, introdotto nel Capitolo 5, non è sufficiente perché usato da procedure come *suspend* e *resume* che non hanno connessione con i semafori. Inoltre tali processi appaiono su delle liste al contrario dei processi suspended - *kill* necessità una distinzione tra i due casi. Ogniqualvolta gli stati di un processo non possono adeguatamente indicare quali operazioni devono essere messe alla luce, il progettista ne inventa uno nuovo. In questo caso chiameremo questo stato "waiting", e ci riferiremo ad esso nel codice con la costante simbolica PRWAIT. La figura 6.1 mostra l'espansione delle transizioni.

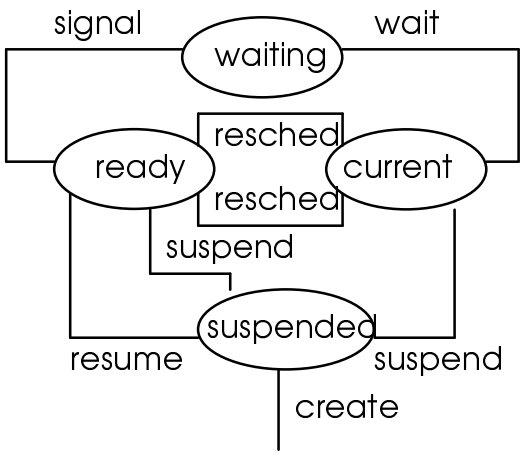

Figura 6.1 - Transizioni per lo stato "waiting"

6.2 Strutture dati per i semafori

In PC-Xinu, le informazioni sui semafori sono tenute in una tabella globale chiamata *semaph*. Ogni elemento inerente contiene un valore intero che funge da contatore, lo stato e i riferimenti della lista associata; la sua definizione è data in C dalla struttura *sentry*. Il file sem.h contiene i dettagli:

```
/* sem.h - isbadsem */
```

```
#ifndef NSEM
#define NSEM            45    /* numero di semafori, se non definito */
#endif

#define SFREE     '\01'       /* questo semaforo è libero */
#define SUSED     '\02'       /* questo semaforo è usato */

struct sentry {               /* elemento della tabella dei semafori */
      char  sstate;           /* può avere stato SFREE o SUSED */
      short semcnt;           /* contatore per questo semaforo */
      short sqhead;           /* indice per la testa della lista def. in q.h */
      short sqtail;           /* indice per la coda della lista def. in q.h */
};

extern struct sentry semaph[];
extern int nextsem;

#define isbadsem(s)     (s<0 || s>=NSEM)
```

Nella struttura *sentry*, il campo *semcnt* contiene il valore intero corrente del semaforo. La lista dei processi sospesi su un semaforo risiede nella struttura q; i campi di *sentry* - *sqhead* e *sqtail* - danno soltanto gli indici della testa e della coda. Il campo di stato, *sstate* dice se quel semaforo è correntemente libero o in uso.

Per tutto il sistema, i semafori sono identificati da un intero. Come per i processi, gli identificatori di semaforo sono dei valori significativi per connetterli con il corrispondente elemento della tabella:

> *I semafori sono identificati dal loro indice nella tabella dei semafori globale semaph.*

Le chiamate di sistema *wait* e *signal* implementano le operazioni di base sui semafori. *Wait(s)* decrementa il valore del semaforo *s*. Se esso rimane non negativo, *wait* ritorna al chiamante immediatamente. Altrimenti, mette in coda il processo chiamante sulla lista associata ad s, cambia lo stato del processo in PRWAIT, e chiama *resched* per eseguire uno tra i processi pronti. La lista è tenuta implementando l'algoritmo FIFO con inserimenti in coda e cancellazione in testa. Essenzialmente, un processo che esegue una *wait* su un semaforo con valore negativo volontariamente passa il controllo alla CPU dopo aver registrato il suo identificatore sulla lista dei processi sospesi.

Una volta messo in coda, un processo rimane lì (e quindi, non in esecuzione) fin quando non raggiunge la testa e qualche altro processo non esegue una signal su tale semaforo. Quando la chiamata a *signal* ripristina il processo nella lista di quelli pronti esso diviene eleggibile per l'uso della CPU ed eventualmente ricomincia l'esecuzione. Dal punto di vista del processo sospeso la sua ultima azione consisteva chiamare *ctxsw*. La chiamata a *ctxsw* ritorna a *resched*, la chiamata a *resched* ritorna a *wait*, e quella a *wait* ritorna ovunque era stata chiamata.

```
/* wait.c - wait */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#inlcude <sem.h>

/*------------------------------------------------------------------------
 *   wait  --  rende il processo corrente sospeso su un semaforo
 * ------------------------------------------------------------------------
 */
```

```
SYSCALL wait(sem)            /* SYSCALL == int */
      int   sem;
{
      int   ps;
      register struct sentry  *sptr;      /* sentry = elem. tab. dei semafori */
      register struct pentry  *pptr;      /* pentry = elem. tab. dei processi */

      disable(ps);                  /* disabilita interrupt */
      if (isbadsem(sem) || (sptr = &semaph[sem]) ->sstate == SFREE) {

/* #define isbadsem(s)  (s<0 || s>= NSEM)       "in sem.h" */
/* sptr punta all'indirizzo della tab. dei semafori di elem. sem "in sem.h"  */

            restore(ps);            /* riabilita interrupt */
            return(SYSERR);         /* SYSERR == -1      "in kernel.h"  */
      }
      if ( --sptr->semcnt < 0 ) {
            (pptr =  &proctab[currpid])->pstate = PRWAIT;
            pptr->psem = sem;
            enqueue(currpid, sptr->sqtail);     /* inserisce nella coda */
            resched();
      }
      restore(ps);
      return(OK);
}
```

Il codice di signal incrementa il contatore del semaforo e rende pronto per l'esecuzione il primo processo sospeso.

```
/* signal.c - signal */

#include <conf. h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/*------------------------------------------------------------------------
 *   signal  --  signal incrementa semaforo e rilascia un processo sospeso
 */------------------------------------------------------------------------
 */
SYSCALL signal(sem)
register int sem;
{
      register struct sentry  *sptr;
      int   ps;

      disable(ps);
      if (isbadsem(sem) || (sptr = &semaph[sem]->sstate == SFREE) {
            restore(ps);
            return(SYSERR);
      }
      if ( sptr->semcnt++ < 0 ) {
            ready(getfirst(sptr->sqhead));      /* rende pronto il primo processo
della coda */
            resched();
      }
      restore(ps);
      return(OK);
}
```

Sebbene può sembrare difficile da comprendere il motivo per cui *signal* rende un processo pronto anche se il valore del contatore rimane negativo ed il motivo per cui *wait* non mette in coda il processo sempre, la ragione è facile da capire ed implementare. *Wait* e *signal* non mutano le seguenti condizioni:

> *Il contatore non negativo di un semaforo significa che la coda è vuota; quando è negativo significa che la coda contiene n processi sospesi.*

Entrambe le procedure mutano il campo semcnt; wait lo decrementa ed aggiunge il processo corrente nella coda se il valore è negativo; signal lo incrementa e rimuove un processo dalla coda se essa non è vuota.

6.3 Creazione e cancellazione di un semaforo

Il bisogno dei semafori può nascere e scomparire con il progredire dell'esecuzione. Xinu permette ai processi la richiesta, l'uso ed il rilascio di semafori. I processi possono creare un arbitrario numero di semafori in ordine sparso, in quantità tale da non eccedere il valore definito (vedi sem.h e conf.h). Per minimizzare il costo di implementazione dei semafori, il sistema, durante l'inizializzazione, prealloca testa e coda della struttura q per ogni lista concatenata associata. Così solo un piccolo ammontare di lavoro necessita di essere svolto durante la creazione.

Le chiamate di sistema *screate* e *sdelete* allocano e rilasciano semafori. *Screate*, mostrata di seguito, prende il valore iniziale del semaforo come argomento e ritorna il relativo identificatore. Il metodo è semplice: si cerca un elemento libero della tabella semaph e si inizializza. *Screate* usa la procedura *newsem* per ricercare un elemento libero. *Screate* poi inizializza il contatore e ritorna l'indice del semaforo appena allocato.

```
/* screate.c - screate, newsem */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/* ------------------------------------------------------------------------
 *  screate  --  crea ed inizializza un semaforo, ritorna il suo id
 * ------------------------------------------------------------------------
 */
SYSCALL screate(count)
      int   count;                  /* valore iniziale (>= 0) */
{
      int   ps;
      int   sem;

      disable(ps);
      if (count<0 || (sem=newsem()) == SYSERR) {
            restore(ps);
            return(SYSERR);
      }
      semaph[sem]. semcnt = count;  /* viene inizializzato il contatore */
      /* sqhead e sqtail già inizializzati all'avvio del sistema */
      restore(ps);
      return(sem);
}

/* ------------------------------------------------------------------------
```

```
 *  newsem  --  alloca un semaforo non usato e ritorna il suo indice
 * ------------------------------------------------------------------------
 */
LOCAL       newsem();
{
      int   sem;
      int   i;

      for (i=0; i < NSEM; i++) {
            sem = nextsem--;
            if (nextsem < 0)
                  nextsem = NSEM-1;
            if (semaph[sem].state == SFREE) {
                  semaph[sem].sstate = SUSED;
                  return(sem);
            }
      }
      return(SYSERR);
}
```

Sdelete inverte le azioni di *screate*. Prende l'indice di un semaforo come argomento e rilascia l'elemento della tabella dei semafori per un ulteriore uso.

```
/* sdelete.c - sdelete */
#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/* ------------------------------------------------------------------------
 * sdelete -- cancella un semaforo rilasciando l'elemento corrispondente nella
tabella
 * ------------------------------------------------------------------------
 */
SYSCALL sdelete(sem)
      int   sem;
{
      int   ps;
      int   pid;
      struct sentry     *sptr;      /* indirizzo del semaforo da liberare */

      disable(ps);
      if ( isbadsem(sem) || semaph[sem].sstate==SFREE) {
            restore(ps);
            return(SYSERR);
      }
      sptr = &semaph[sem];
      sptr->sstate = SFREE;
      if (nonempty(sptr->sqhead)) {       /* libera processi sospesi */
            while ((pid=getfirst(sptr->sqhead)) != EMPTY)
            /* rimuove e ritorna il primo processo della lista */
                  ready(pid);
            resched();
      }
      restore(ps);
      return(OK);
}
```

Se alcuni processi rimangono in coda quando *sdelete* cerca di cancellare un semaforo, essa deve disfarsi di essi. Quando viene chiamata con un semaforo attivo, *sdelete* mette tutti i processi

sospesi nella lista dei pronti, permettendogli di riprendere l'esecuzione come se sul semaforo avvenisse una signal. Questa è soltanto una delle possibili soluzioni.

6.4 Ritornare il contatore del semaforo

E' utile poter ritornare in possesso del contatore del semaforo. Per esempio, la conoscenza di un contatore positivo su un semaforo può indicare che ci sono unità disponibili di una particolare risorsa. Il valore di un semaforo, contenuto nella componente *semcnt* della struttura è ritornato dalla routine *scount* mostrata qui:

```
/* scount.c - scount */

/* ------------------------------------------------------------------------
 *  scount  --  ritorna il contatore di un semaforo
 * ------------------------------------------------------------------------
 */
SYSCALL scount(sem)
int sem;
{
      extern struct sentry semaph[];
      int   ps;
      int   ct;

      disable(ps);
      if (isbadsem(sem) || semaph[sem].sstate == SFREE) {
            restore(ps);
            return(SYSERR);
      }
      ct = semaph[sem].semcnt;      /* ct contiene il valore restituito */
      restore(ps);
      return(ct);
}
```

6.5 Altre utility per i semafori

Due addizionali utility, *signaln* e *sreset*, sono usate nel software di input/output e anche disponibili ai programmi utente. *Signaln* è equivalente a chiamare *signal* un certo numero di volte, mentre *sreset* cancella un semaforo e ne ricrea un altro con un nuovo, iniziale valore. Queste routine sono state progettate per essere più efficienti della combinazione di altre.

```
/* signaln.c - signaln */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/* ------------------------------------------------------------------------
 * signaln  --  effettua signal su un semaforo n volte
 * ------------------------------------------------------------------------
 */
SYSCALL signaln(sem, count)
      int   sem;
```

```
      int   count;
{
      struct sentry *sptr;
      int   ps;

      disable(ps);
      if (isbadsem(sem) || semaph[sem].sstate==SFREE || count<=0) {
            restore(ps);
            return(SYSERR);
      }
      sptr = &semaph[sem];
      for ( ; count > 0; count--)
            if ((sptr->semcnt++) < 0)     /* se negativo sblocca il processo */
                  read(getfirst(sptr->sqhead));  /*in testa alla lista semaph */
      resched();
      restore(ps);
      return(OK);
}


/* sreset.c - sreset */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <q.h>
#include <sem.h>

/* ------------------------------------------------------------------------
 * sreset  --  reset the count and queue of a semaphore
 * ------------------------------------------------------------------------
 */
SYSCALL sreset(sem, count)
      int   sem;
      int   count;
{
      struct sentry *sptr;
      int   ps;
      int   pid;
      int   slist;

      disable(ps);
      if (isbadsem(sem) || count < 0 || semaph[sem].sstate==SFREE) {
            restore(ps);
            return(SYSERR);
      }
      sptr = &semaph[sem];
      slist = sptr->sqhead;
      while ((pid=getfirst(slist)) != EMPTY)    /* svuota la lista */
            ready(pid);
      sptr->semcnt = count;         /* reinizializza con un nuovo valore */
      resched();
      restore(ps);
      return(OK);
}
```