

Sistemi operativi

Lez. 10-11-12: primitive per la
concorrenza
i semafori

Disabilitazione interrupt

- Due processi possono trovarsi in sezione critica simultaneamente solo perché chi vi è entrato per primo è stato interrotto e la CPU è stata passata ad un altro processo
- La CPU passa da un processo ad un altro solo in seguito ad un interrupt
- Disabilitando gli interrupt prima di accedere alla sezione critica e riabilitandoli all'uscita, la CPU non può essere passata ad altri
 - un processo può leggere e aggiornare le variabili condivise senza essere interrotto da un altro processo

Disabilitazione interrupt

- Si può garantire l'accesso in mutua esclusione ad una sezione critica disabilitando gli interrupt
 - Utile per il kernel per aggiornare le sue variabili
 - Non è raccomandabile dare all'utente il potere di disabilitare gli interrupt
 - Istruzione privilegiata
 - L'utente potrebbe non riabilitarli più e non cedere più la CPU
 - Rallenta la risposta agli eventi
 - Inutile in un sistema multiprocessore
 - Non è strutturata e rende di difficile comprensione il codice

TSL

- La soluzione sw al problema della mutua esclusione è abbastanza complessa
 - Algoritmo di Peterson
- La soluzione hw mediante disabilitazione degli interrupt non è adeguata per programmi utente
 - Codice dal comportamento oscuro, scarsa efficienza, inutile nei sistemi multiprocessore
- Una soluzione migliore può essere ottenuta con un supporto MIRATO da parte del linguaggio macchina

Test and Set Lock

- I primi a notare questa necessità furono i progettisti dell'OS/360
- Aggiunsero nel linguaggio macchina del sistema l'istruzione

TEST AND SET LOCK (TSL)

TSL

- TSL opera nel seguente modo
 - Sintassi
 - TSL register, flag
 - Semantica
 - `register := flag /* copia in register il valore corrente di flag */`
 - `flag := 1 /* assegna 1 a flag */`

con la garanzia di indivisibilità o atomicità delle due operazioni

- sono eseguite come se si trattasse di un'unica istruzione

TSL

enter_region:

 tsl register,lock

 cmp register,#0

 jne enter_region

 ret

| copy lock to register and set lock to 1

| was lock zero?

| if it was non zero, lock was set, so loop

| return to caller; critical region entered

leave_region:

 move lock,#0

 ret

| store a 0 in lock

| return to caller

Osservazioni

- Svantaggi di soluzioni basate su istruzioni speciali
 - Busy waiting
 - Starvation possibile a causa della scelta del prossimo processo da eseguire quando uno lascia la sezione critica
 - Deadlock per inversione di priorità
 - se il processo in sezione critica P1 viene interrotto e la CPU assegnata ad un processo P2 con priorità superiore che cerca di entrare, non può a causa di P1 che però non può eseguire avendo priorità inferiore a P2, che è attivo in busy wait

Semafori

- I semafori sono delle variabili intere
 - condivise tra più processi
 - possono assumere come valore 0 e 1
 - in questo caso parliamo di semafori binari
 - oppure un intero ≥ 0
 - in questo caso parliamo di semafori generalizzati
- Le operazioni definite sui semafori sono due
 - P(sem) o Down(sem) o wait(sem)
 - la P sta per Proberen (provare)
 - V(sem) o Up(sem) o signal(sem)
 - la V sta per Verhogen (incrementare)
- A queste si aggiunge l'inizializzazione
 - semaforo = valore

Semafori binari

- Le operazioni `down(sem)` e `up(sem)` hanno la seguente semantica:

`down(sem) :`

```
if (sem == 0) then wait on sem;  
      else sem = 0;
```

`up(sem) :`

```
if (some process is waiting on sem)  
  then awake it  
  else sem = 1;
```

- Il sistema operativo ne garantisce l'atomicità
 - eseguite come se fossero un'unica istruzione macchina

Semafori binari

- I processi non attendono più in busy wait
- A ciascun semaforo è associata una coda in cui vengono posti i processi bloccati sul particolare semaforo
- La disciplina di gestione della coda non è specificata nella definizione dei semafori
 - solitamente FIFO
- I processi sono risvegliati dalla coda quando un altro processo esegue un'operazione di up sul semaforo su cui sono bloccati

Semafori binari

- Con i semafori binari il problema della mutua esclusione tra n processi può essere risolto molto facilmente
- Ogni processo esegue la sequenza

```
down (sem) /* enter_region */
```

```
Sezione critica
```

```
up (sem) /* leave_region */
```

- Il primo processo che riesce ad eseguire `down(sem)`, entra in regione critica

Semafori binari

- I semafori binari possono anche essere usati per sincronizzare due o più processi
 - Determinare una specifica sequenza di esecuzione
- Esempio

```
semaphore S1 = 1;
```

```
semaphore S2 = 0;
```

```
semaphore S3 = 0;
```

P1	P2	P3
down (S1)	down (s2)	down (s3)
:	:	:
up (S2)	up (S3)	up (S1)

I semafori generalizzati

- Le operazioni `down(sem)` e `up(sem)` hanno la seguente semantica

`down (sem) :`

```
if (sem == 0) then wait on sem;  
           else sem = sem - 1;
```

`up (sem) :`

```
if (some process is waiting on sem)  
   then awake it  
   else sem = sem + 1;
```

- Il sistema ne garantisce l'atomicità

Problemi implementativi

- Dal punto di vista della struttura dati un semaforo è:

```
typedef struct
{
    int valore;
    pcb *next; }
semaphore;
```

- Le primitive Up e Down vengono implementate come SVC

Problemi implementativi

- Le operazioni Down(sem) e Up(sem)
Down(sem) :

```
if (sem == 0) then waiting on sem ;  
else sem = sem - 1;
```

Up(sem) :

```
if (some process is waiting on sem)  
then awake it  
else sem = sem + 1;
```



come effettuo
questo test?

Problemi implementativi

Down (sem) :

```
sem.valore=sem.valore - 1;  
if (sem.valore < 0)  
    then {aggiungi un processo a sem.next ;  
          metti il processo in waiting;}
```

Up (sem) :

```
sem.valore=sem.valore + 1;  
if (sem.valore <=0)  
    then {rimuovi un processo da sem.next;  
          metti il processo in ready; }
```

Problemi implementativi

- Atomicità
 - Su sistemi uniprocessore: disabilita interrupt
 - Su sistemi multiprocessore disabilitare gli interrupt non basta, Up e Down devono essere eseguiti come sezione critica, ad esempio:

`LOCK(lockbit) → TSL o Peterson`

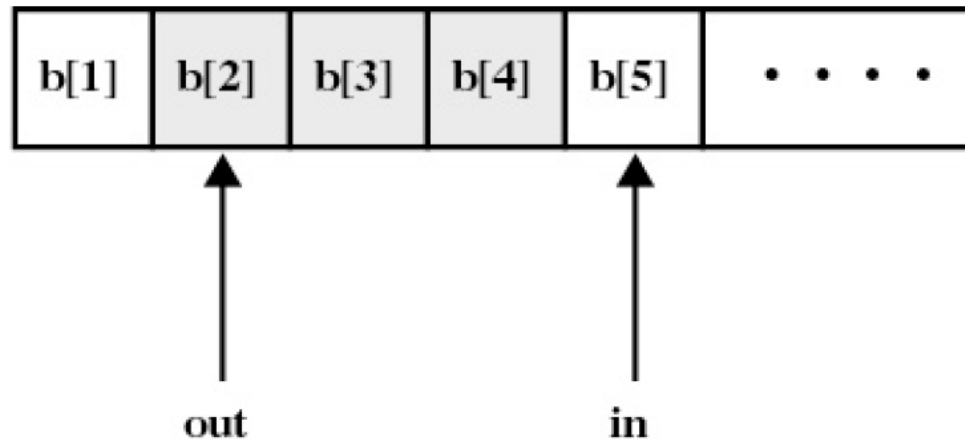
`Down(sem)`

`UNLOCK(lockbit)`

Produttore - Consumatore

- Il produttore è un processo che produce elementi che vengono consumati da un processo consumatore
- Produttore e consumatore possono eseguire concorrentemente grazie ad un buffer
 - il produttore vi inserisce i suoi elementi e il consumatore li preleva
- Il produttore non deve inserire elementi in celle del buffer già occupate
- Il consumatore non deve consumare elementi non ancora prodotti

Prod-Cons (buffer infinito)



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

Schema prod-cons con buffer infinito

```
Semaphore sem;
int buffer[infinito];
void prod (void)
{ int i=0;
  int item;
  while (TRUE) {
    produci(&item);
    buffer[i]=item;
    i = i+1;
    up(sem)
  }
```

```
void cons (void)
{ int i=0;
  int item;
  while (TRUE) {
    down(sem);
    item=buffer[i];
    i = i+1;
    consuma(&item);
  }
```

Domande

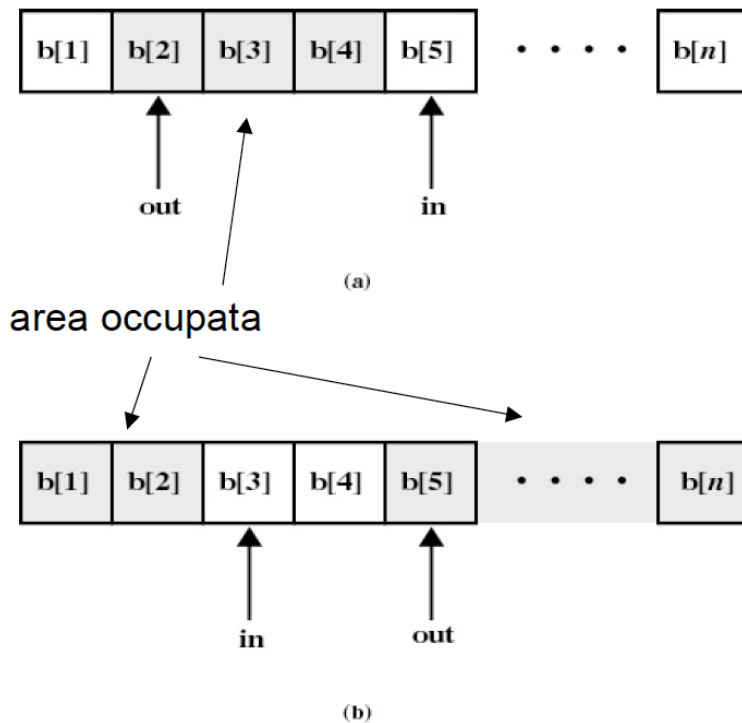
- Il semaforo utilizzato nel precedente programma, è un semaforo binario o un semaforo generalizzato?
- Quale problema si presenta nel momento in cui abbiamo a che fare con più processi produttori?

N prod, M cons, buffer infinito

```
Semaphore sem, mutex;
int buffer[infinito];
void prod (void)
{ int i=0;
  int item;
  while (TRUE) {
    produci(&item);
    down (mutex);
    buffer[i]=item;
    i = i+1;
    up (mutex)
    up(sem)
  }
```

```
void cons (void)
{ int i=0;
  int item;
  while (TRUE) {
    down(sem);
    down(mutex);
    item=buffer[i];
    i = i+1;
    up(mutex);
    consuma(&item);
  }
```

Caso generale



Modificare il caso con
buffer infinito al caso di
buffer circolare

Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

Producer-Consumer Problem

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
} . . .
```

/ number of slots in the buffer */*
/ semaphores are a special kind of int */*
/ controls access to critical region */*
/ counts empty buffer slots */*
/ counts full buffer slots */*

/ TRUE is the constant 1 */*
/ generate something to put in buffer */*
/ decrement empty count */*
/ enter critical region */*
/ put new item in buffer */*
/ leave critical region */*
/ increment count of full slots */*

Producer-Consumer Problem

```
void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item( );
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

/ infinite loop */*
/ decrement full count */*
/ enter critical region */*
/ take item from buffer */*
/ leave critical region */*
/ increment count of empty slots */*
/ do something with the item */*

Lettori - Scrittori

- Problema dei lettori e scrittori
- Un insieme di processi condivide un file dal quale alcuni possono solo leggere i dati, altri solo scriverli
- Più lettori possono leggere simultaneamente
- Un solo scrittore per volta può scrivere
- Quando uno scrittore scrive, nessun lettore può leggere
- Diverso da Prod/cons
 - i lettori non modificano il buffer

Esercizio

- Sviluppare una soluzione al problema nei casi:
- 1 scrittore, 1 lettore con accessi in mutua esclusione al file sia dello scrittore sia del lettore rispetto a questo
- 1 scrittore, n lettori
- m scrittori, n lettori
- Va definito chi ha priorità, lettori o scrittori

1 Lettore 1 Scrittore

```
Semaphore mutex = 1,  
        leggo =0 ;  
void write (void)  
{  
    while (TRUE) {  
        down(mutex) ;  
        write(item) ;  
        up(mutex) ;  
        up(leggo) ;  
    }  
}
```

```
void read (void)  
{  
    while (TRUE) {  
        down(leggo) ;  
        down(mutex) ;  
        read(item) ;  
        up(mutex) ;  
    }  
}
```

1 Scrittore n Lettori

- Gli scrittori aspettano che tutti i lettori abbiano terminato
- Gli scrittori scrivono in mutua esclusione
- I lettori devono sapere quanti sono
- Se c'è un solo lettore, deve aspettare che eventuali scrittori finiscano
- Lettori che arrivano quando altri stanno già leggendo procedono senza aspettare

Soluzione

```
Lettori_scrittori();
int numlett; /* contiamo i lettori */
Semaphore mutex = 1; /* mutua
esclusione lettori */
Semaphore r_w = 1; /* blocco
scrittori */

Scrittore {
down (r_w);
WRITE;
up (r_w)
}
```

```
Letture {
down(mutex); // lock su numlett
numlett += 1; // un lettore in più
if (numlett == 1)
    down(r_w); // synch w/ scrittori
up(mutex); // unlock numlett
Read;
down(mutex); // lock numlett
numlett -= 1; // un lettore in meno
if (numlett == 0)
    up(r_w); // up for grabs
up (mutex); // unlock numlett}
}
```

Considerazioni

- Che tipo di semafori abbiamo usato?
- Si potrebbe usare un semaforo generalizzato per contare i lettori?
- Cosa succede nei seguenti casi
 - Solo lettori presenti
 - Solo scrittori presenti
- Lettori e scrittori presenti ma
 - un lettore è arrivato primo
 - uno scrittore è arrivato primo
- I lettori continuano ad arrivare prima che l'ultimo finisca
- In coda su `r_w` ci sono sia lettori che scrittori