Sistemi operativi

Lez. 6/7: primitive per la concorrenza e semafori

Supporto alla concorrenza

- L'algoritmo di Peterson consente una soluzione del problema della concorrenza che non richiede alcun presupposto particolare, ma presenta alcune criticità:
  - Non immediata comprensione
  - Estensione al caso n>2 processi, non immediata

Supporto hardware

- Per facilitare la soluzione di questo problema si può ricorrere all'uso appropriato di alcune istruzioni hardware:
  - Disabilitazione interrupt
  - Istruzioni atomiche

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
- Se un processo disabilita gli interrupt prima di accedere ad una sezione critica e li riabilita all'uscita, la CPU non può, nel frattempo, essere utilizzata da altri processi

Disabilitazione interrupt: criticità

- Si può garantire l'accesso in mutua esclusione ad una sezione critica disabilitando gli interrupt
  - 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

Istruzioni dedicate

- Si tratta di istruzioni che eseguono 2 operazioni atomicamente su registri o operandi di memoria, queste istruzioni possono essere:
  - Testare il valore di una variabile e settare il suo valore (Test & Set)
  - Scambiare il contenuto di due variabili (XCGH)

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:esegue le seguenti istruzioni come se si trattasse di un'unica istruzione:
    - Register ← flag
    - flag ← 1
- TSL può essere usata per implementare le primitive di accesso alla sezione critica `enter_region` e `leave_region`

TSL

```
enter_region:
        tsl register,lock       | copy lock to register and set lock to 1
        cmp register,#0         | was lock zero?
        jne enter_region        | if it was non zero, lock was set, so loop
        ret                     | return to caller; critical region entered

leave_region:
        move lock,#0            | store a 0 in lock
        ret                     | return to caller
```

Intel x86

XCHG—Exchange Register/Memory with Register

Opcode	Instruction	Clocks	Description
90+*rw*	XCHG AX,*r16*	2	Exchange word register with AX
90+*rw*	XCHG *r16*,AX	2	Exchange word register with AX
90+*rd*	XCHG EAX,*r32*	2	Exchange dword register with EAX
90+*rd*	XCHG *r32*,EAX	2	Exchange dword register with EAX
86 /*r*	XCHG *r/m8,r8*	3	Exchange byte register with EA byte
86 /*r*	XCHG *r8,r/m8*	3	Exchange byte register with EA byte
87 /*r*	XCHG *r/m16,r16*	3	Exchange word register with EA word
87 /*r*	XCHG *r16,r/m16*	3	Exchange word register with EA word
87 /*r*	XCHG *r/m32,r32*	3	Exchange dword register with EA dword
87 /*r*	XCHG *r32,r/m32*	3	Exchange dword register with EA dword

Operation

temp ← DEST
DEST ← SRC
SRC ← temp

Description

The XCHG instruction exchanges two operands. The operands can be in either order. If a memory operand is involved, the LOCK# signal is asserted for the duration of the exchange, regardless of the presence or absence of the LOCK prefix or of the value of the IOPL.

Flags Affected

None.

Intel x86

19.1.1. LOCK Prefix and the LOCK# Signal

The LOCK prefix and its bus signal only should be used to prevent other bus masters from interrupting a data movement operation. The LOCK prefix can be used with the following Pentium processor instructions when they modify memory. An invalid-opcode exception results from using the LOCK prefix before any other instruction, or with these instructions when no write operation is made to memory (i.e., when the destination operand is in a register).

- Bit test and change: the BTS, BTR, and BTC instructions.
- Exchange: the XCHG, XADD, CMPXCHG, and CMPXCHG8B instructions (no LOCK prefix is needed for the XCHG instruction).
- One-operand arithmetic and logical: the INC, DEC, NOT, NEG instructions.
- Two-operand arithmetic and logical: the ADD, ADC, SUB, SBB, AND, OR, and XOR instructions.

A locked instruction is *guaranteed* to lock only the area of memory defined by the destination operand, but may lock a larger memory area. For example, typical 8086 and 80286 configurations lock the entire physical memory space.

Semaphores (shared memory used for signalling between multiple processors) should be accessed using identical address and length. For example, if one processor accesses a semaphore using word access, other processors should not access the semaphore using byte access.

The integrity of the lock is not affected by the alignment of the memory field. The LOCK# signal is asserted for as many bus cycles as necessary to update the entire operand.

Procedura C

- L'istruzione:

```
xchg (addr, 1)
```

svolge le seguenti operazioni

- `%eax = 1`
- `xchgl %eax, addr`
- `return %eax`

spin_lock

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
spin_lock(struct spinlock *lk)
{
// The xchg is atomic and return the old value of lk.
// It also serializes, so that reads after acquire are not
// reordered before it.
while (xchg(&lk->locked, 1) != 0)
        asm volatile ("pause");

        }
```

spin_unlock

```
// Release the lock.
void
spin_unlock(struct spinlock *lk)
{
        xchg(&lk->locked, 0);
}
// Mutual exclusion lock.
struct spinlock {
        unsigned locked;       // Is the lock held?
};
```

Osservazioni

- Spin-locks vanno utilizzati per proteggere operazioni brevi: incremento di un contatore, allocazione di uno spazio di memoria, ecc. In questi casi l'operazione di spin_lock non spreca troppo tempo di CPU, nel caso di lock occupato
- Nel caso di operazioni più lunghe (I/O) è opportuno individuare meccanismi più efficienti

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 essere mandato in esecuzione avendo priorità inferiore a P2, che è a sua volta in esecuzione in busy waiting

Busy Waiting è indispensabile?

- Busy waiting è un meccanismo molto potente che però abbiamo visto crea diversi problemi
- L'ideale sarebbe poter disporre di un meccanismo che consenta ad un thread di verificare lo stato di un lock e bloccarsi sino a che questi non sia disponibile
- Si introducono a questo scopo due funzioni:
  - `Sleep() e wakeup()`

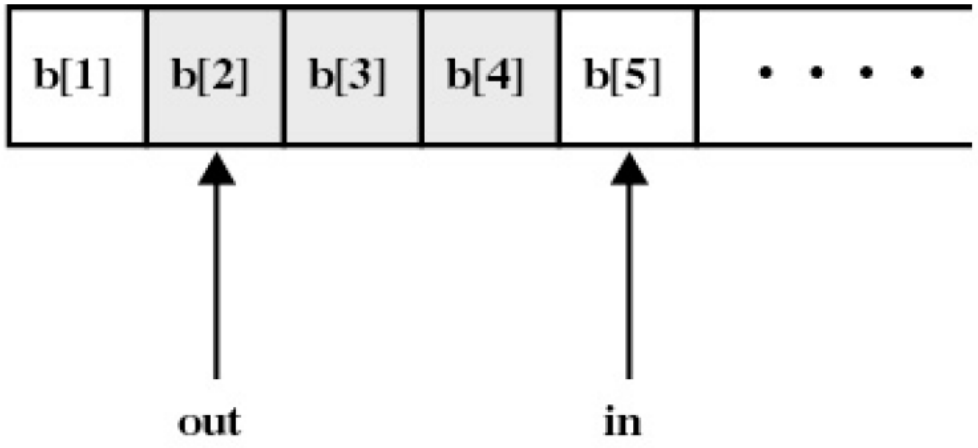
sleep / wakeup

- `sleep()`: mette in waiting il processo che la esegue
- `wakeup()`: risveglia un processo che si è bloccato, se nessun processo è sospeso la wakeup non crea alcun effetto

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

Produttore- Consumatore: con busy waiting

```
int N 10
int buf[N];
int in = 0, out = 0, cnt = 0;
```

```
Task producer {
  int item;
  while (1) {

    while(cnt == N);

  item = makeitem();
  buf[in] = item;
  in = (in + 1) % N;

  lock(lock);
  cnt++;
  unlock(0);
    }
                }
```

```
Task consumer {
  item_t item;
  while (1) {

    while (cnt == 0);

    item = buf[out];
    out = (out + 1) % N;

    lock(lock);
    cnt--;
    unlock(lock);
    consume(item);
  }
}
```

Produttore - Consumatore: con Sleep() e Wakeup

```
int N 10
int buf[N];
int in = 0, out = 0, cnt = 0;
```

```
Task producer {
  int item;
  while (1) {
    if (cnt == N)
      sleep();
  item = mkitem();
  buf[in] = item;
  in = (in + 1) % N;
  lock(lock);cnt++;unlock(0);
  if (cnt == 1)
    wakeup(consumer);
  }
}
```

```
Task consumer {
  item_t item;
  while (TRUE) {
    if (cnt == 0)
      sleep();
    item = buf[out];
    out = (out + 1) % N;
    lock(lock);cnt--;unlock(lock);
    if (cnt == N-1)
      wakeup(producer);

    consume(item);
  }
}
```

Any Race Condition ???

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 (counting semaphore)
- Sono utilizzati per:
  - Mutua esclusione (inizializzati ad 1)
  - Attesa di eventi (inizializzati a 0)
  - Contare risorse (inizializzati a n)

I semafori

- 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 waiting
- 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

Esercizio

Quale delle seguenti stringhe non può essere stampata dai seguenti processi assumendo che i valori iniziali dei semafori generalizzati siano sem =2, sem1= 1, sem2 = 1:

A ::	B ::	C: ….
while true {	while true {	while true {
down (sem)	**down (sem1)**	**down (sem2)**
write (ab)	**write (bb)**	**write (cb)**
up (sem1) }	**up (sem2) }**	**up (sem1) }**

a) abbbcbcbcb
b) bbabbbcbbb
c) cbabbbbbbb
d) cbabbbbbab

IMPLEMENTAZIONE

Struttura dati

- 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

Up & down

- 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?

Down()

```
void down ( semaphore sem)
{
  disable interrupts;
  if (sem.valore > 0) {
   sem.valore = sem.valore - 1;
   enable interrupts;
   return; }
  else {
  current_thread in waiting queue;
  enable interrupts;
  call scheduler; }
  }
```

up

```
Up(semaphore sem)
{
  disable interrupt
    if (sem.valore > 0)
          sem.valore = sem.valore + 1;
   else {rimuovi primo processo da sem.next;
          metti il processo in ready; }
  enable interrupt;
  return/call scheduler
}
```

Sistemi Multiprocessore

- La disabilitazione degli interrupt non è più sufficiente a garantire l'atomicità di `down()` e `up()` qualora le stesse siano eseguite su processori diversi
- È necessario individuare nuove soluzioni

Semaforo su sistemi multiprocessore

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

```
typedef  struct
         { int valore;
           int lock; /* initially 0 */
           pcb *next; }
         semaphore;
```

down multiproc

```
void down ( semaphore sem)
{
  disable interrupts;
  spin_lock (sem.lock)
  if (sem.valore > 0)
   { sem.valore = sem.valore - 1;
     spin_unlock (sem.lock);
     enable interrupts;
     return; }
  current_thread in waiting queue
  spin_unlock (sem.lock);
  enable interrupts;
  call scheduler;
  }
```

up

```
Up(semaphore sem)
{
  disable interrupt
  spin_lock (sem.lock);
  if (sem.valore > 0)
          sem.valore ++;
  else {rimuovi primo processo da sem.next;
            metti il processo in ready; }
  spin_unlock(sem.lock);
  enable interrupt;
  return/call scheduler
}
```

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

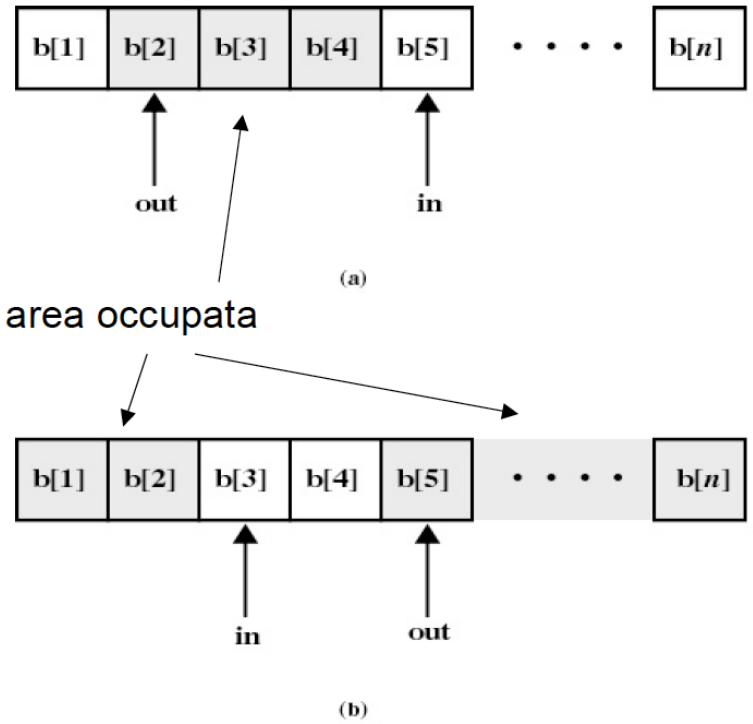


Figure 5.15 Finite Circular Buffer for the Producer/Consumer Problem

Modificare il caso con buffer infinito al caso di buffer circolare

Producer-Consumer Problem

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                          /* decrement empty count */
          down(&mutex);                          /* enter critical region */
          insert_item(item);                     /* put new item in buffer */
          up(&mutex);                            /* leave critical region */
          up(&full);                             /* increment count of full slots */
     }
}. . .
```

Producer-Consumer Problem

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

      while (TRUE) {                    /* infinite loop */
            down(&full);                /* decrement full count */
            down(&mutex);               /* enter critical region */
            item = remove_item( );      /* take item from buffer */
            up(&mutex);                 /* leave critical region */
            up(&empty);                 /* increment count of empty slots */
            consume_item(item);         /* 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 e priorità dello scrittore nell'accesso alla risorsa
- 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 scrivono in mutua esclusione
- Lettori che arrivano quando altri stanno già leggendo accedono alla risorsa senza aspettare
- I lettori sanno quanti sono
- Accede alla risorsa il primo processo che arriva

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)
}
```

```
Lettore {
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