Meccanismi di sincronizzazione:

Semafori e Monitor

Introduzione

- Nelle prossime lezioni vedremo alcuni meccanismi dei sistemi operativi e dei linguaggi di programmazione sviluppati per facilitare la scrittura di programmi concorrenti. Inizieremo a vedere i *semafori*
- Semafori
  - il nome indica chiaramente che si tratta di un paradigma per la sincronizzazione simile ai semafori stradali
- I semafori sono stati introdotti da Dijkstra nel 1965

Semafori - Definizione

- Prinicipio base
  - due o più processi possono cooperare attraverso semplici segnali, in modo tale che un processo possa essere bloccato in specifici punti del suo programma finchè non riceve un segnale da un altro processo
- Definizione

  È un tipo di dato astratto per il quale sono definite due operazioni
  - V (*verhogen*, incrementare) chiamata anche *up*
    viene invocata per inviare un segnale, quale il verificarsi di un evento o il rilascio di una risorsa
  - P (*proberen*, testare) chiamata anche *down*
    viene invocata per attendere il segnale

Semafori - Descrizione informale

- Un semaforo può essere visto come una *variabile* condivisa S tra i processi cooperanti
- S assume come valori numeri interi;
  viene inizializzata ad un valore non negativo
- Operazione P(S)/down(S)):
  - attendi finché il valore di *S* è maggiore di 0;
  - quindi decrementa *S*
- Operazione V(S)/up(S):
  - incrementa il valore di *S*
- Le azioni P e V devono essere atomiche

Semafori - Invariante

- Siano
  - n_p il numero di azioni $P/down$ completate
  - n_v il numero di azioni V/up completate
  - $init$ il valore iniziale del semaforo
    (cioè numero di risorse disponibili per i processi)
- allora vale la seguente proprietà invariante: $n_p \leq n_v + init$
- Se $init = 0$ il numero di attese dell'evento non deve essere superiore al numero di volte che l'evento si è verificato ($n_p \leq n_v$)
- Se $init > 0$ il numero di richieste soddisfatte (n_p) non deve essere superiore al numero iniziale di risorse ($init$) + il numero di risorse restituite (n_v)

Esempio: Sezione Critica per n processi

- Variabili condivise:
  - **var** *mutex* : *semaphore*
  - inizialmente *mutex* = 1
- Processo P_i

while (TRUE) {
  down(*mutex*);
    sezione critica
  up(*mutex*);
    sezione non critica
}

Esempio: Produttore-Consumatore con semafori

```
#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 */
     }
}
```



```
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 */
     }
}
```

Esempio: Sincronizzazione tra due processi

- Variabili condivise:
  - **var** *sync* : *semaphore*
  - inizialmente $sync = 0$
- Processo P_1 Processo P_2

⋮	⋮
S_1;	down(sync);
up(sync);	S_2;
⋮	⋮

- S_2 viene eseguito solo dopo S_1.

Implementazione dei semafori a livello Kernel

- L'implementazione della definizione classica dei semafori è basata su busy waiting:

```
procedure down(S)
 atomic(
  while (S=<0) no-op;
  S:=S-1
 )

procedure V(S)
 atomic(
  S:=S+1
 )
```


- Se i semafori sono implementati a livello Kernel è possibile limitare l'utilizzazione di busy waiting
- Per questo motivo:
  - L'operazione down deve *sospendere* il processo che la invoca
  - L'operazione up deve *svegliare* uno dei processi sospesi
- Per ogni semaforo
  - Il S.O. deve mantenere una struttura dati contenente l'insieme dei processi sospesi
  - quando un processo deve essere svegliato, si seleziona uno dei processi sospesi

- Semafori FIFO
  - politica first-in first-out: il processo che è stato sospeso più a lungo viene risvegliato per primo
  - la struttura dati utilizzata dal S.O. è una coda

Implementazione dei semafori a livello Kernel

Struttura dati per semafori

```
type semaphore = record
                     value: integer;
                     L: list of process;
                 end;
```

Assumiamo due operazioni fornite dal sistema operativo:

- *sleep()*: sospende il processo che la chiama (rilascia la CPU e va in stato *waiting*)
- *wakeup*(*Pid*): pone in stato di *ready* il processo *P*.

Implementazione dei semafori (Cont.)

- Le operazioni sui semafori sono definite come segue:

```
down/P(S): S.value := S.value − 1;
           if S.value < 0
             then begin
                      aggiungi questo processo a S.L;
                      sleep();
                  end;
up/V(S):   S.value := S.value + 1;
           if S.value ≤ 0
             then begin
                      togli un processo Pid da S.L;
                      wakeup(Pid);
                  end;
```


Implementazione dei semafori (Cont.)

- In questa implementazione *value* può avere valori negativi: indica quanti processi sono in attesa su quel semaforo
- le due operazioni *down (wait)* e *up (signal)* devono essere *atomiche* fino a prima della *sleep* e *wakeup:* problema di sezione critica, da risolvere come visto prima:
  - disabilitazione degli interrupt: semplice, ma inadatto a sistemi con molti processori
  - uso di istruzioni speciali (test-and-set)
  - ciclo busy-wait (spinlock): generale, e sufficientemente efficiente (le due sezioni critiche sono molto brevi)

Implementazione dei semafori (Cont.)

- In un sistema multiprocessore è necessario utilizzare una delle tecniche per risolvere il problema della sezione critica viste in precedenza (Peterson, Fornaio, ecc)
- In generale infatti vorremmo il seguente schema

down/P(S):
– enter CS –
$S.value := S.value - 1$;
if $S.value < 0$
then begin
aggiungi questo processo a $S.L$;
sleep();
end;
– exit CS –

up/V(S):
– enter CS
$S.value := S.value + 1$;
if $S.value \leq 0$
then begin
togli un processo *Pid* da $S.L$;
wakeup(*Pid*);
end;
– exit CS –

Riassumendo

- Utilizzando queste tecniche non abbiamo eliminato completamente la busy waiting
- Tuttavia abbiamo limitato busy-waiting alle sezioni critiche delle operazioni P e V, queste sezioni critiche sono molto brevi
- Senza semafori: potenziali busy-waiting di lunga durata, meno frequenti
- Con semafori: busy-waiting di breve durata, più frequenti

Semafori binari (Mutex)

- I mutex sono semafori con due soli possibili valori: *bloccato* o *non bloccato*
- Utili per implementare mutua esclusione, sincronizzazione, . . .
- due primitive: *mutex_lock* e *mutex_unlock*.
- Semplici da implementare, anche in user space (p.e. per thread). Esempio:

Osservazione: Memoria condivisa?

Implementare queste funzioni richiede una qualche memoria condivisa.

- A livello kernel: strutture come quelle usate dai semafori possono essere mantenute nel kernel, e quindi accessibili da tutti i processi (via le apposite system call)
- A livello utente:
  - all'interno dello stesso processo: adatto per i thread
  - tra processi diversi: spesso i S.O. offrono la possibilità di condividere segmenti di memoria tra processi diversi (*shared memory*)
  - alla peggio: file su disco

Deadlock con Semafori

- **Deadlock (stallo):** due o più processi sono in attesa indefinita di eventi che possono essere causati solo dai processi stessi in attesa.

- L'uso dei semafori può portare a deadlock. Esempio: siano S e Q due semafori inizializzati a 1

P_0	P_1
down(*S*);	*down*(*Q*);
down(*Q*);	*down*(*S*);
⋮	⋮
up(*S*);	*up*(*Q*);
up(*Q*);	*up*(*S*);

- Programmare con i semafori è molto delicato e prono ad errori, difficilissimi da debuggare. Come in assembler, solo peggio, perché qui gli errori sono race condition e malfunzionamenti non riproducibili.

Meccanismi di sincronizzazione

Monitor

Monitor

- Un *monitor* è un tipo di dato astratto che fornisce funzionalità di mutua esclusione
  - collezione di dati privati e funzioni/procedure per accedervi.
  - i processi possono chiamare le procedure ma non accedere alle variabili locali.
  - *un solo* processo alla volta può eseguire codice di un monitor
- Il programmatore raccoglie quindi i dati condivisi *e tutte le sezioni critiche relative* in un monitor; questo risolve il problema della mutua esclusione
- Implementati dal compilatore con dei costrutti per mutua esclusione (p.e.: inserisce automaticamente `lock_mutex` e `unlock_mutex` all'inizio e fine di ogni procedura)

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .
    .
    .
    end;
end monitor;
```


Monitor: Controllo del flusso di controllo

Per sospendere e riprendere i processi, ci sono le variabili *condition*, simili agli eventi, con le operazioni

- *wait(c):* il processo che la esegue si blocca sulla condizione c.
- *signal(c):* uno dei processi in attesa su c viene risvegliato.

  A questo punto, chi va in esecuzione nel monitor? Due varianti:

  - chi esegue la *signal(c)* si sospende automaticamente (*monitor di Hoare*)
  - la *signal(c)* deve essere l'ultima istruzione di una procedura (così il processo lascia il monitor) (*monitor di Brinch-Hansen*)
  - i processi risvegliati possono provare ad entrare nel monitor solo dopo che il processo attuale lo ha lasciato

  Il successivo processo ad entrare viene scelto dallo scheduler di sistema

- i *signal* su una condizione senza processi in attesa vengono persi

Produttore-consumatore con monitor

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count – 1;
        if count = N – 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```


Monitor (cont.)

- I monitor semplificano molto la gestione della mutua esclusione (meno possibilità di errori)
- Veri costrutti, non funzioni di libreria ⇒ bisogna modificare i compilatori.
- Implementati (in certe misure) in veri linguaggi.
  Esempio: i metodi `synchronized` di Java.
  - solo un metodo `synchronized` di una classe può essere eseguito alla volta.
  - Java non ha variabili *condition*, ma ha *wait* and *notify* (+ o − come *sleep* e *wakeup*).
- Un problema che rimane (sia con i monitor che con i semafori): è necessario avere *memoria condivisa* ⇒ questi costrutti non sono applicabili a sistemi distribuiti (reti di calcolatori) senza memoria fisica condivisa.