Capitolo 7: Sincronizzazione

- Il problema della sincronizzazione.
- Il problema della sezione critica.
- Hardware per la sincronizzazione.
- Semafori.
- Problemi classici di sincronizzazione.
- Monitor (cenni).

Il problema della sincronizzazione

- L'accesso concorrente a dati condivisi può portare alla loro inconsistenza.
- Per mantenere la consistenza dei dati sono necessari dei meccanismi che assicurino l'esecuzione ordinaria di processi cooperanti che condividano uno spazio di indirizzi (codice e dati).
- Es.: in un sistema multithread, più thread cooperano mentre sono eseguiti in modo asincrono e condividono dati.
  - Possiamo esemplificare un paradigma di cooperazione come nel modello del produttore/consumatore.

Codice dei thread

Le chiamate del produttore:

```
while (count == BUFFER_SIZE)
    ; // non fare nulla
// aggiungi un elemento al buffer
++count;
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
```

Le chiamate del consumatore:

```
while (count == 0)
    ; // non fare nulla
// elimina l'elemento dal buffer
--count;
item =  buffer[out];
out = (out + 1) % BUFFER_SIZE;
```

Corse critiche

- ++count può essere implementata nel modo seguente:
  register1 = count
  register1 = register1 + 1
  count = register1
- count- - può essere implementata nel modo seguente:
  register2 = count
  register2 = register2 - 1
  count = register2
- Si consideri il seguente ordine di esecuzione. Supponiamo count=5:

  S0: produttore esegue register1 = count {register1 = 5}
  S1: produttore esegue register1 = register1 + 1 {register1 = 6}
  S2: consumatore esegue register2 = count {register2 = 5}
  S3: consumatore esegue register2 = register2 - 1 {register2 = 4}
  S4: produttore esegue count = register1 {count = 6 }
  S5: consumatore esegue count = register2 {count = 4}
- Il risultato dell'esecuzione non è universale ma dipende dall'ordine con cui si è dato accesso ai dati.

Il problema della sezione critica

- Primo metodo per garantire la regolamentazione degli accessi è dichiarare come **critica** la sezione di codice ad accesso condiviso.
- In un sistema costituito dagli n thread $\{T_0, T_1, \ldots, T_{n-1}\}$, ogni thread potrà avere un segmento di codice determinato come critico.
- Soluzione al problema della sezione critica:
  1. Mutua esclusione – se il thread T_i sta eseguendo la sua sezione critica, nessun altro thread può eseguire la propria sezione critica.
  2. Progresso – se nessun thread sta eseguendo la sua sezione critica e alcuni thread vogliono entrare nella loro, allora solo i thread che non stanno eseguendo la loro sezione non critica possono partecipare alla decisione su chi sarà il prossimo a entrare nella propria sezione critica, e questa scelta non può essere ritardata indefinitamente.
  3. Attesa limitata *(bounded waiting)* – esiste un limite al numero di volte in cui gli altri thread possono entrare nelle loro sezioni critiche dopo che un thread ha fatto richiesta di entrare nella propria e prima che tale richiesta sia esaudita. Prevenzione della starvation di un singolo thread.
     - Assumiamo che ciascun thread sia in esecuzione a velocità non nulla.
     - Non si può fare alcuna ipotesi sulle velocità *relative* degli *n* thread.

Soluzione software

- Si presenta complessa e spesso inefficiente anche nel caso di sistemi semplici (pochi thread).
- Tutte le soluzioni software sono generalmente basate su di un algoritimo di ordinamento. Generalmente allo scopo si adopera l'algoritmo di Bakery.
- Si impone l'accesso alle sezioni critiche dei processi secondo uno stringente paradigma di tipo FCFS.
- **ALGORITMO DI BAKERY**
  - Si ipotizzi l'accesso alla sezione critica per n generici thread.
  - Prima di fare ingresso alla sezione critica, ciascun thread riceve un token numerico. Il thread possessore del token di valore più basso accede alla propria sezione critica.
  - Se due thread *Ti and Tj* ricevono il medesimo token, se *i < j,* allora *Ti* riceve la precedenza*;* in caso contrario la riceve *Tj.*
  - Lo schema di generazione algoritmica dei token genera sempre una sequenza crescente, *p.es.:* 1,2,3,3,3,3,4,5...

Hardware per la sincronizzazione

- Molti sistemi forniscono istruzioni hardware per la risoluzione del problema della sezione critica.
- Monoprocessori: si possono disabilitare gli interrupt durante l'accesso alle sezioni critiche.
  - La sequenza di istruzioni corrente sarà eseguita in ordine senza interruzioni.
- Di solito troppo inefficiente sui sistemi multiprocessore (i comandi per la disabilitazione degli interrupt devono essere passati a tutti i processori).
  - Ritardo nell'accesso alle sezioni critiche e nell'uscita da esse.
  - I sistemi operativi che lo usano sono carenti in scalabilità quando vengono aggiunte CPU.
    - I costi computazionali ed i ritardi aumentano al crescere delle CPU
- I calcolatori moderni forniscono **istruzioni hardware speciali** che funzionano in modo **atomico**.
    - Atomico = non-interrompibile.
  - Controllare e modificare il contenuto di una parola di memoria.
  - Scambiare il contenuto di due parole.

Semafori

- Sebbene efficaci, le soluzioni hardware sono di uso complesso per i programmatori di applicazioni.
- Strumenti poco complessi di sincronizzazione che impongono iterativamente un blocco durante un'attesa in modo attivo ***(spin lock)***.
- Semaforo *S* – variabile intera.
- Dopo l'inizializzazione si può accedere ad un semaforo S solo tramite due operazioni: acquire() e release().
- Si tratta di due operazioni indivisibili (le modifiche al valore intero di S sono permesse da un solo thread per volta):

```
acquire(S) {
  while S <= 0
        ; // nessuna operazione
  S--;
}
release(S) {
  S++;
}
```

- Nella acquire(S) il test sul valore positivo di S e la sua possibile modifica devono essere eseguiti senza interruzioni.

Impiego

- Semaforo generalizzato – il valore può variare in un dominio senza restrizioni.
- Semaforo binario – il valore può essere solo 0 o 1; può essere più facile da implementare.
  - Noto anche come mutex lock perchè consente di gestire la mutua esclusione.

```
Semaphore S; // assumo S inizializzato a 1

acquire(S);
criticalSection();
release(S);
```

- Possiamo adoperare semafori per controllare l'accesso a sezioni critiche.
- Si può adoperare un semaforo generalizzato S per controllare l'accesso ad una risorsa con un numero finito di istanze.
  - S viene inizializzato al numero di istanze della risorsa disponibili.
  - Un thread che desidera accedere alla risorsa esegue una acquire() su S (decrementando il contatore).
  - Un thread che rilascia la risorsa esegue una release() (incrementando il counter).
  - Quando S=0, tutte le istanze sono allocate e i thread che richiedano la risorsa resteranno in attesa del rilascio di un esemplare di essa.

Implementazione di semafori (1/2)

- Semafori come quelli descritti (spinlock) hanno lo svantaggio del *busy waiting*.
  - Mentre un processo è nella propria sezione critica, ogni altro processo che prova ad accedere alla propria deve ciclare continuamente sprecando cicli di CPU che potrebbero essere sfruttati da altri processi.
  - Gli spinlock non obbligano a cambi di contesto pertanto tornano utili quando si prevede che i blocchi vengano tenuti per brevi periodi (sistemi multiprocessore).

```
acquire(S){
    S--;
    if (S < 0) {
        add the process to list;
        block;
    }
}

release(S){
    S++;
    if (S <= 0) {
        remove a process P from list;
        wakeup(P);
    }
}
```

- Per superare il busy waiting si ridefiniscono acquire() e release().
  - Una acquire() che trova una variabile semaforica negativa mette il thread in una coda di attesa associata al semaforo.
  - Il microscheduler seleziona un altro processo da eseguire.
  - Un processo bloccato viene risvegliato da un wake-up a seguito di una release() di un altro processo sul semaforo.
  - Dopo il wake-up il thread bloccato viene rimosso dalla coda del semaforo e riposizionato nella coda di ready.
- I semafori dovranno così essere definiti come coppia variabile semaforica (intero) e coda di attesa.

Implementazione di semafori (2/2)

- Il semaforo può così assumere anche valori negativi.
  - Il valore assoluto rappresenta il numero di processi in coda di attesa.
- La lista d'attesa è implementata mediante due puntatori alla testa ed alla fine di una coda FIFO di PCB.
- Deve garantire che due processi non possano eseguire una acquire() ed una release() sullo stesso semaforo nello stesso momento. Questa situazione genera un problema di sezione critica.
  - Soluzione:
    - Ambiente monoprocessore: inibire gli interrupt durante l'esecuzione di acquire() e release().
    - Ambiente multiprocessore: adoperare istruzioni hardware speciali o adottare soluzioni software per la sezione critica.
- acquire() e release() non evitano il busy waiting ma lo dirottano nelle rispettive sezioni critiche dei programmi applicativi.
  - Queste sezioni sono brevi pertanto l'attesa attiva avviene raramente e quando capita è solo per brevi periodi.
  - Le applicazioni invece possono impiegare molto tempo nelle sezioni critiche ed in tal caso l'attesa attiva è estremamente svantaggiosa.

Stalli e starvation

- Stallo (deadlock) – due o più processi aspettano un evento che può essere causato solo da uno dei processi in attesa (nel caso dei semafori l'evento in questione è una release()).
- Consideriamo un sistema costituito da due processi *P* ciascuno dei quali accede a due semafori S e Q valorizzati a 1.

P_0	P_1
acquire(S);	acquire(Q);
acquire(Q);	acquire(S);
.	.
.	.
release(S);	release(Q);
release(Q);	release(S);

- Blocco indefinito (starvation) – i processi attendono indefinitamente all'interno del semaforo. Un processo può non essere mai rimosso dalla coda del semaforo nella quale è sospeso.
  - Può verificarsi se la coda semaforica di attesa è organizzata secondo il criterio LIFO.

Monitor

- Benché i semafori forniscano un meccanismo conveniente ed efficace per la sincronizzazione, il loro utilizzo non corretto comporta errori di temporizzazione difficilmente individuabili.
  - Essi avvengono in modo asincrono solo in determinate sequenze di esecuzione.
- Un monitor è un costrutto in un linguaggio ad alto livello che provvede alla sincronizzazione di thread.
  - Il tipo monitor è un tipo di dato astratto che incapsula metodi pubblici per l'esecuzione di operazioni su dati privati.
  - Un tipo monitor presenta operazioni definite dall'utente su cui è fornita la mutua esclusione: solo un thread per volta può essere attivo in un monitor.
  - Il tipo monitor contiene:
    - La dichiarazione di variabili mediante le quali si definisce lo stato dell'istanza.
    - Le funzioni/procedure che operano sulle variabili stesse.

```
monitor monitor-name
{
    shared variable declarations

    public entry p1(…) {
        …
    }

    public entry p2(…) {
        …
    }

    …

    public entry pN(…) {
        …
    }

    {
        initialization code
    }
}
```

Le variabili di tipo condition

- I thread non possono accedere direttamente all'implementazione interna di un monitor.
- Solo le procedure definite internamente al monitor possono accedere alle variabili locali o a quelle condivise.
- Il costrutto del monitor proibisce l'accesso concorrente alle procedure interne. Il programmatore non deve codificare la sincronizzazione esplicitamente.
- Le variabili di tipo *condition* (x e y nell'esempio) e i metodi *wait* e *signal* implementano la sincronizzazione;
- Un thread che invoca x.wait è sospeso finchè un altro thread non invoca x.signal.