Corso di Informatica

Modulo T5

C1– Mutua esclusione

Prerequisiti

- Mutua esclusione
- Risorse seriali e parallele
- Risorse permanenti e consumabili
- Stati di un processo
- Stati di un *thread*
- Descrittore di processo

Introduzione

Ci interessiamo ora di come possano comunicare due o più *thread*.

Per fare ciò, sarà necessario richiamare alcuni concetti di base dei sistemi operativi, rimandando al corso relativo eventuali approfondimenti.

Nella descrizione, qualora non sia necessario differenziare tra processi e *thread*, privilegeremo il termine *thread*.

Thread asincroni

Fin qui si è fatto riferimento a **thread asincroni**, ossia *thread* nei quali valgono le seguenti considerazioni:

- ciascun *thread* contiene tutti i dati di cui necessita per avanzare regolarmente;
- ciascun *thread* avanza in modo indipendente dallo stato di altri *thread* paralleli.

Comunicazione fra *thread*

In realtà, molto spesso, ci troviamo in un ambiente *multithreading* in cui i vari *thread* possono influenzarsi reciprocamente, ossia l'avanzamento di un *thread* può condizionare l'avanzamento di uno o più altri *thread*.

Per studiare questo aspetto, ricordiamo che due *thread* possono comunicare interagendo secondo due modalità:

- **competizione**, o **interazione indiretta** (in questa Unità)
- **cooperazione**, o **interazione diretta** (nella prossima Unità)

Normalmente, è il nucleo del S.O. a gestire le interazioni tra i processi. Nella programmazione Java, ad esempio, il compito di gestire le interazioni fra *thread* è delegato alla **JVM**.

Competizione

Partiamo con un esempio.

Abbiamo una sola stampante, che deve servire la richiesta di due processi di stampa P1 ed P2.

La stampante può stampare un solo lavoro alla volta, quindi P1 ed P2 sono in **competizione** tra loro.

Le richieste di P1 e P2 giungono in modo asincrono.

Un processo di stampa, una volta iniziato, deve terminare il servizio, prima di passare al successivo.

Problema: garantire che uno dei due processi ottenga l'accesso esclusivo alla stampante ed esegua prima dell'altro.

Competizione
- mutua esclusione

Il problema della **competizione** fra *thread* prevede di gestire la **mutua esclusione** e può essere posto come segue,

Dati due *thread* T e T' ed una risorsa R seriale e non prerilasciabile (almeno per un dato tempo), occorre garantire che:

- in ogni istante, R sia libera o assegnata ad uno solo dei due *thread*;
- ciascuno dei *thread* ottenga in un tempo finito l'uso di R

Competizione
- esempi

- nell'esempio iniziale della stampante, è evidente che quando questa è detenuta da uno dei due processi, il suo uso è inibito all'altro processo. La stampante è quindi una risorsa serializzata, ossia usabile in mutua esclusione;
- un web server che voglia contare il numero di pagine che ha servito ai vari client richiedenti, deve gestire il conteggio in modo seriale
- l'arrivo di due chiamate **push**(x, S) e **push**(y, S) concorrenti sullo stesso stack va serializzato
- l'accesso alla coda dei pronti, gestita secondo il modello round-robin, da parte di più thread deve essere serializzato.

Codice rientrante
- definizione

In alcuni casi è possibile evitare la competizione fra processi scrivendo **codice rientrante**.

In Informatica, il codice di un programma o di una routine è detto **codice rientrante** se è progettato in modo che una singola copia del codice in memoria possa essere condivisa ed eseguita contemporaneamente da processi separati.

Codice rientrante
- esempi

Le due funzioni del seguente esempio sono rientranti.

```
int f (int i)
{ return i + 2; }

int g (int i)
{ return f(i) + 2; }
```

Funzioni rientranti

La f() è rientrante poiché non modificando il valore di i, può essere eseguita in parallelo da più *thread*.

La g() è rientrante poiché è rientrante f().

Codice rientrante
- esempi

In quest'altro esempio, entrambe le funzioni non sono rientranti poiché il valore in uscita dipende dall'ordine di esecuzione della funzione da parte di più *thread* paralleli.

```
int f()
{ var = var + 2;
  return var; }

int g()
{ return f() + 2; }
```

Funzioni non rientranti

Il valore di *var* dipenderà dall'ordine di esecuzione dei *thread*

Codice rientrante
- thread safe call

Un altro esempio è offerto dalla routine di libreria (chiamate al sistema) che devono avere codice rientrante, potendo essere istanziate contemporaneamente da più *thread*.

Questa caratteristica, detta *thread safe call*, serve a garantire che se più *thread* istanziano una stessa chiamata ad una funzione di sistema, che usa variabili di sistema relative al processo, questa deve essere in grado di mantenere distinti i valori da restituire a ciascun *thread* chiamante, per non causare problemi nella lettura/scrittura dei dati di sistema.

La mutua esclusione

Se si ha codice non rientrante, la mutua esclusione può essere realizzata in base:

- alla priorità:
  - il processo a priorità maggiore acquisisce la risorsa. Se la risorsa è posseduta da un processo a priorità inferiore, avviene il **prerilascio**.
- all'istante di arrivo:
  - l'assegnazione segue la cronologia degli eventi.
- all'uso di semafori
  - si deve affrontare il problema della **sincronizzazione** fra i *thread*

Ci dedichiamo a quest'ultimo aspetto

Istruzioni atomiche

L'istruzione

var = var + 2

in bytecode diventa qualcosa del tipo seguente:

load var R1 // carica in R1 il valore di var
add R1 2 // aggiungi 2 a R1
store R1 var // salva il valore di R1 in var

Ora, se il *thread* che esegue il blocco delle suddette istruzioni fosse interrotto, detto blocco non verrebbe eseguito in maniera "atomica".

La locuzione **istruzione atomica** indica un'istruzione che non può essere interrotta perché considerata indivisibile.

Istruzioni atomiche

Consideriamo infatti due thread **Thread1** e **Thread2** che eseguono il blocco precedente. Se l'esecuzione di **Thread1** e di **Thread2** avviene in sequenza, il valore in *var* risulta corretto e prevedibile.

Sequenze di interleaving

Thread1	var	Thread2
load var R1	2	
add R1 2	4	
store var R1	4	
	4	**load** var R1
	6	**add** R1 2
	6	**store** var R1

Il valore di *var* è prevedibile

Istruzioni atomiche

Se, invece, l'esecuzione di **Thread1** e di **Thread2** avviene in concorrenza, la "corsa" fra essi può far perdere il valore corretto di *var*.

Sequenze di interleaving

Thread1	var	Thread2
load var R1	2	
add R1 2	4	
	2	**load** var R2
	4	**add** R1 2
store var R1	4	
	4	**store** var R1

Il valore di *var* non è prevedibile

Sincronizzazione

Se un codice non è rientrante, è necessario che venga serializzato, per fare in modo che i vari *thread* concorrenti lo eseguano uno alla volta: ciò si può fare mediante la **sincronizzazione**.

```
int var = 1;
int f()
{ var = var + 2;
  return var; }

int g()
{ return var + 2; }
```

Come visto in precedenza, questi sono esempi di codice non rientrante, poiché l'incremento della variabile *var* dipende dalla tempistica di esecuzione dei *thread* che lo eseguono, e quindi impredicibile.

Il risultato di un'operazione dipende dall'ordine con essa viene eseguita dai vari thread

Sincronizzazione
- sezione critica

Dato un processo P ed una risorsa R, indichiamo con la locuzione **sezione critica di P su R** il segmento di programma in cui P detiene R in modo esclusivo. Ne consegue che:

- in ogni istante, un solo processo può trovarsi in una sua sezione critica su R
- la sezione critica di P su R deve essere considerata **indivisibile**.

```
do {    ...
        /* inizio sezione critica */
        …
        …
        /* fine sezione critica */
    } while (1)
```

Compito indivisibile

Sincronizzazione
- semafori

Serializzare quindi i vari *thread*, significa quindi farli avanzare in sequenza secondo tempi ben precisi, ossia farli diventare **thread sincroni**.

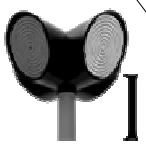
La tecnica della programmazione concorrente, consente di creare *thread* sincroni, mediante l'operazione di **sincronizzazione**, realizzata con i **semafori**.

Un utilizzo intuitivo molto semplice dei semafori si ha per garantire la mutua esclusione nell'accesso a una risorsa: in tal caso basta usare un **semaforo binario**.

> Se un problema di mutua esclusione non è gestito in modo corretto, sincronizzando opportunamente i vari *thread*, si ottengono risultati imprevedibili, con conseguenze a volte disastrose.

Sincronizzazione
- semafori

In informatica, un **Semaforo** è un tipo di dato astratto (*Abstract Data Type*) gestito da un sistema operativo multitasking per sincronizzare l'accesso a risorse condivise tra processi o *thread*, che chiamiamo con un termine unico **task**.



Un semaforo **S** può immaginarsi composto da:

- una **variabile intera**;
- la sua **interfaccia**;
- una **coda di processi**: ad ogni semaforo è associata una coda contenente i descrittori di processo di tutti i processi che concorrono per l'uso della risorsa regolata dal semaforo stesso.

Sincronizzazione
- semafori - valore

La variabile intera contenuta in **S** indica il numero di unità della risorsa che sono disponibili.

Si può avere:

- 0 o 1 per **semafori binari** (sono i classici **mutex**, cosiddetti perché consentono di gestire la mutua esclusione);
- un intero per **semafori a conteggio**

Sincronizzazione
- semafori - operazioni

L'interfaccia di un semaforo **S** sono le operazioni ammesse su esso. Possiamo avere:

- **Inizializzazione**: si assegna un valore iniziale al semaforo **S**
- **wait(S)**: il semaforo **S** viene decrementato. Se, dopo il decremento, **S** ha un valore negativo, il *task* T viene sospeso e accodato, in attesa di essere riattivato da un altro *task*.

void wait (**Intero** S)
{ **if** (S = = 0)
poni T *in attesa passiva di una signal (S);*
else S - -;
}

Un *task* T chiede di utilizzare una risorsa. Se S non è disponibile, il *task* T viene messo in attesa, per essere risvegliato solo quando gli sarà assegnata la risorsa richiesta.

Sincronizzazione
- semafori - operazioni

- **signal(S)**: il semaforo **S** viene incrementato. Se ci sono *task* in coda, il primo di essi viene tolto dalla coda e posto in stato di pronto.

void signal (**Intero** S)

{ **if** (*esiste un task* T1 *in stato di attesa di* **signal(S)**)

risveglia T1*;*

else S ++;

}

Un *task* rilascia l'unità di cui non ha più bisogno, e per la quale altri *task* potrebbero già essere in attesa.

Gestione della mutua esclusione

A questo punto è chiaro che per gestire il problema dell'accesso esclusivo ad una risorsa (mutua esclusione) sarà sufficiente un algoritmo di questo tipo:

GestisciAccesso (**Risorsa** R)
Inizio
Intero S=1;
wait(S); // accesso alla risorsa
utilizza risorsa;
signal (S); // rilascio della risorsa
Fine;

Sincronizzazione in Java
- metodi synchronized

La serializzazione delle operazioni in **Java** è un problema che viene risolto molto semplicemente dichiarando i metodi con la clausola **synchronized**:

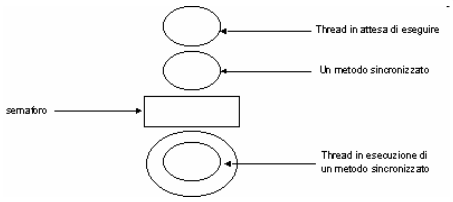
Ad esempio, per una classe *Stack* si hanno i metodi:

synchronized void push (**int** x) { ... }
synchronized int pop () { ... }

In tal modo la **JVM** garantisce che all'oggetto *Stack s* acceda solo un *thread* alla volta.

Sincronizzazione in Java
- metodi synchronized

Quando un metodo è definito **synchronized**, all'oggetto di quella classe viene associato un semaforo.



Quando un *thread* accede ad un oggetto sincronizzato, sbarra l'accesso a tutti gli altri (*lock*) e ogni nuovo *thread* viene posto in attesa (sospeso). Quando un *thread* finisce l'accesso, viene riattivato il primo in attesa.

Attenzione: la sincronizzazione ha un impatto *notevole* sull'efficienza.

Argomenti

- *Thread* asincroni
- Comunicazione fra *thread*
- Competizione
  - mutua esclusione
  - esempi
- Codice rientrante
  - definizione
  - esempi
  - *thread safe call*
- La mutua esclusione
- Istruzioni atomiche
- Sincronizzazione
  - sezione critica
  - semafori
  - semafori – valore
  - semafori - operazioni
- Gestione della mutua esclusione
- Sincronizzazione in Java
  - metodi synchronized

Altre fonti di informazione

- P.Camagni, R.Nicolassy – Java: interfacce grafiche e programmazione concorrente, ed. Hoepli Informatica
- **http://java**.sun.com/docs/books/tutorial/essential/threads/index.html