Sistemi Operativi

PROVA SCRITTA IN ITINERE – 09 giugno 2014

Esercizio 1. (punti 18)

In un sistema eseguono un thread di tipo `Ta` e due thread di tipo `Tb`. I thread vanno in esecuzione in modo alternato, prima il thread `Ta`, poi i due thread `Tb` (i due thread `Tb` vanno in esecuzione in "parallelo"). L'esecuzione di `Ta` e dei due thread `Tb` corrisponde ad un ciclo. I thread terminano dopo N cicli (con `N` letto sulla riga degli argomenti all'avvio del programma). Scrivere la soluzione Java che modella il funzionamento descritto utilizzando i **semafori** per risolvere i problemi di sincronizzazione tra thread.

Esercizio 2. (punti 12)

In un sistema sono presenti 4 processi (P_1, P_2, P_3, P_4) e risorse di 5 tipi diversi (A, B, C, D, E). All'istante t_0, le risorse disponibili sono le seguenti:

Disponibili				
A	**B**	**C**	**D**	**E**
0	x	1	y	2

L'allocazione ed il fabbisogno massimo di risorse per processo sono dati in tabella:

Processi	**Massimo**					**Allocate**				
	A	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**
P_1	3	0	3	3	2	1	0	0	0	1
P_2	1	1	1	2	1	0	1	1	1	0
P_3	2	2	2	1	1	2	0	1	1	1
P_4	3	2	2	1	4	3	1	0	1	0

Utilizzando l'***algoritmo del banchiere*** determinare:

- il minimo valore di *x* e di *y* per il quale lo stato di allocazione delle risorse sopra riportato è sicuro. Determinare anche una sequenza sicura di esecuzione;
- supponendo di attribuire ad *x* ed *y* i valori ottenuti al punto precedente, verificare se la richiesta r_4=(0,1,1,0,4) da parte del processo P_4 all'istante t_0 è ammissibile.

Soluzione

Esercizio 1.

La soluzione utilizza un semaforo semA per il thread A inizializzato a 2, e due semafori semb1 e semB2 per i thread B, entrambi inizializzati a 0.
L'inizializzazione a 2 di semA e la acquire(2) impediscono ad A di eseguire due volte di seguito richiedendo due release dai thread B. Il fatto di usare due distinti semafori per i thread B permette di evitare ad uno solo di essi di eseguire due volte di seguito prevenendo la starvation di un thread B (come invece potrebbe accadere nel caso un solo semaforo fosse usato per B).
La gestione del conteggio può essere realizzata localmente ai thread senza ricorrere a variabili condivise dato che ogni thread esegue un conteggio indipendente dagli altri.

```
package compito_20140609;

import java.util.concurrent.Semaphore;

public class Principale {
	public static void main( String args[] ) {
		int N = 0;
		if ( args.length > 0 ) {
			N = Integer.parseInt( args[0] );
			if ( N <= 0 ) {
				System.out.println( "The number must be > 0!" );
				System.exit( 1 );
			}
			System.out.println( "The number of runs is " + N );
			System.out.flush();

			Semaphore semA = new Semaphore(2);
			Semaphore semB1 = new Semaphore(0);
			Semaphore semB2 = new Semaphore(0);
			// one thread of type A
			ThreadA ta = new ThreadA(semA,semB1,semB2,N);
			ta.start();
			// two threads of type B
			ThreadB tb0 = new ThreadB(0,semA,semB1,N);
			tb0.start();
			ThreadB tb1 = new ThreadB(1,semA,semB2,N);
			tb1.start();
		}
	}

	public static void doWork( String t ) {
		long sleeptime = (long)(Math.random()*2000);
		System.out.println( "esegue thread " + t );
		System.out.flush();
		try {
			Thread.sleep(sleeptime);
		}
		catch( InterruptedException ie ) {}
	}
}

public class ThreadA extends Thread {
	private Semaphore semA;
	private Semaphore semB1;
	private Semaphore semB2;
	private int max;
```

```
	public ThreadA( Semaphore sA, Semaphore sB1, Semaphore sB2, int N ) {
		semA = sA;
		semB1 = sB1;
		semB2 = sB2;
		max = N;
	}

	public void run() {
		// local counter
		int count = 0;
		while ( count < max ) {
			try {
				semA.acquire(2);
				Principale.doWork( "Ta" );
				count ++;
				semB1.release();
				semB2.release();
			}
			catch(InterruptedException ie) {}
		}
		System.out.println("ThreadA termina! " + count);
		System.out.flush();
	}
}

public class ThreadB extends Thread {
	private Semaphore semA;
	private Semaphore semB;
	private int max;
	private int id;

	public ThreadB( int i, Semaphore sA, Semaphore sB, int N ) {
		semA = sA;
		semB = sB;
		max = N;
		id = i;
	}

	public void run() {
		int count = 0;
		while ( count < max ) {
			try {
				semB.acquire(1);
				Principale.doWork( "Tb" + id );
				count ++;
				semA.release();
			}
			catch(InterruptedException ie) {}
		}
		System.out.println("ThreadB" + id + " termina! " + count);
		System.out.flush();
	}
}
```

Esercizio 2.

Lo stato iniziale è caratterizzato dalle seguenti strutture dati per massima richiesta, allocazione e necessità:

Processi	Massimo					Allocate					Necessità				
	A	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**
P_1	3	0	3	3	2	1	0	0	0	1	2	0	3	3	1
P_2	1	1	1	2	1	0	1	1	1	0	1	0	0	1	1
P_3	2	2	2	1	1	2	0	1	1	1	0	2	1	0	0
P_4	3	2	2	1	4	3	1	0	1	0	0	1	2	0	4

Affinché lo stato sia sicuro è necessario identificare una sequenza sicura:

- Il primo processo della sequenza può essere solo il processo P_3, per il quale risulta la necessità (0,2,1,0,0) <= (0,x,1,y,2) a condizione che x >=2;
- Ipotizzando x=2, P_3 può terminare rilasciando le risorse allocate. Le disponibili diventano: (2,2,2,y+1,3). L'unico processo la cui necessità può essere soddisfatta è P_2, per il quale vale: (1,0,0,1,1) <= (2,x,2,y+1,3), che è soddisfatta anche per x=0 ed y=0;
- P_2 può terminare rilasciando le risorse allocate che diventano: (2,x+1,3,y+2,3). L'unico processo la cui richiesta può essere soddisfatta è P_1, per il quale risulta: (2,0,3,3,1) <= (2,x+1,3,y+2,3) a condizione che y>=1;
- Ipotizzando y=1, P_1 può terminare rilasciando le risorse allocate. Le disponibili diventano: (3,x+1,3,y+2,4). Con questa disponibilità anche P_4, può terminare dato che: (0,1,2,0,4)<=(3,x+1,3,y+2,4) a condizione che x>=0 ed y>=0.

Combinando i vincoli precedenti si ottiene che i valori minimi di x ed y per i quali lo stato è sicuro sono: x=2 ed y=1. Pertanto il numero di risorse disponibili minimo allo stato t_0 affinché lo stato sia sicuro è: (0,2,1,1,2).

Con queste risorse la sequenza sicura è: <P_3, P_2, P_1, P_4>

Un richiesta di risorse da parte di P_4, r_4=(0,1,1,0,4) non può essere soddisfatta dato che tale richiesta è minore uguale della necessità corrente: r_4=(0,1,1,0,4)<=(0,1,2,0,4), ma eccede la disponibilità della risorsa E: r_4=(0,1,1,0,4) > (1,2,1,1,2) sulla risorsa E. Il processo P_4 deve pertanto aspettare.