Sistemi Operativi

PROVA SCRITTA IN ITINERE – 19 dicembre 2011

Esercizio 1. (punti 20)

In un sistema eseguono `N` thread $T_0, \ldots, T_{N-1}$ (con `N` letto dalla riga degli argomenti). I thread devono eseguire secondo l'ordine $T_0, T_1, T_2, \ldots, T_{N-1}$, in modo circolare. L'esecuzione in ordine degli `N` thread rappresenta un ciclo di esecuzione. Ad ogni esecuzione un thread T_i decrementa un numero intero condiviso della quantità `i*10`. Nel sistema è presente anche un thread `Counter` che esegue in modo asincrono rispetto agli altri (è cioè fuori dalla catena di esecuzione circolare). Il thread `Counter` incrementa di uno in modo continuo il numero intero condiviso con gli altri thread.

L'esecuzione termina quando il numero di cicli eseguiti è uguale a `N` (cioè ogni thread esegue `N` volte). Quando questo accade tutti i thread devono uscire dai rispettivi metodi `run()` e terminare. Il thread T_{N-1} che esce dall'ultimo ciclo di esecuzione deve anche determinare l'uscita dal metodo `run()` del thread `Counter`.

Scrivere il programma Java che realizza lo schema di esecuzione descritto utilizzando i **semafori Java** per risolvere problemi di sincronizzazione tra thread.

Esercizio 2. (punti 10)

Descrivere l'algoritmo di rilevazione dello stallo nel caso di istanza multipla per tipo di risorsa.

Soluzione

Esercizio 1.

```
package compito_20111219;
import java.util.concurrent.Semaphore;

public class Principale {
	public static void main( String args[] ) {
		if ( args.length < 1 ) {
			System.out.println( "Usage: java Principale <num_thread>" );
			System.exit( 1 );
		}
		int N = Integer.parseInt( args[0] );
		Number number = new Number();
		Semaphore sem[] = new Semaphore[ N ];
		sem[ 0 ] = new Semaphore( 1 );
		for ( int i=1; i<N; i++ )
			sem[ i ] = new Semaphore( 0 );
		Counter counter = new Counter( number );
		counter.start();
		for ( int i=0; i<N; i++ ) {
			Worker wk = new Worker( i, sem, N, number );
			wk.start();
		}
	}
}

public class Number {
	private int value;
	private boolean itera;
	private Semaphore mutex;

	public Number() {
		value = 0;
		itera = true;
		mutex = new Semaphore( 1 );
	}

	public void changeValue( int v ) {
		try {
			mutex.acquire();
			value += v;
			System.out.println( "value: " + value );
			System.out.flush();
			mutex.release();
		}
		catch( InterruptedException ie ) {}
	}

	public boolean itera() {
		return itera;
	}

	public void termina() {
		itera = false;
	}

	public int getValue() {
		return value;
	}
}
```

```
public class Counter extends Thread {
	private Number number;

	public Counter( Number n ) {
		number = n;
	}

	public void run() {
		while ( number.itera() ) {
			number.changeValue( 1 );
		}
		System.out.println( "valore di counter: " + number.getValue() );
	}
}

public class Worker extends Thread {
	private int id;
	private Semaphore[] sem;
	private Number number;
	private int N;

	public Worker( int id, Semaphore[] s, int nthread, Number n ) {
		this.id = id;
		sem = s;
		N = nthread;
		number = n;
	}

	public void run() {
		int count = 0;
		while ( count < N ) {
			try {
				sem[ id ].acquire();
			}
			catch( InterruptedException ie ) {}
			number.changeValue( -id*10 );
			sem[ (id+1)%N ].release();
			count ++;
		}
		System.out.println( "termina thread " + id + " count " + count );
		System.out.flush();
		if ( id == N-1 )
			number.termina();
	}
}
```

Esercizio 2.

L'algoritmo di rilevazione dello stallo è usato nelle tecniche di rilevazione e recupero dello stallo per verificare se un insieme di processi è bloccato in modo indefinito in modo da poter eventualmente recuperare da questa situazione. Ipotizzando di avere *n* processi ed *m* tipi di risorsa, l'algoritmo usa le seguenti strutture dati:

- *available*[m]: array di *m* elementi; il generico elemento *available*[i] indica il numero di istanze disponibili per il tipo di risorsa *i*;
- *allocation*[n][m]: matrice con *n* righe ed *m* colonne; il generico elemento *allocation*[i][j] indica il numero di istanze del tipo di risorsa *j* allocate al processo *i*;
- *request*[n][m]: matrice con *n* righe ed *m* colonne; il generico elemento *request*[i][j] indica il numero di ulteriori istanze della risorsa *j* richieste dal processo *i*;

L'algoritmo si sviluppa in 4 passi, come segue:

1. *Inizializzazione*: *work* è un vettore di lunghezza *m* inizializzato come *work = available*. Il vettore *finish[n]* è inizializzato in modo che *finish[i] = true* se *allocation[i][] = 0* (riga *i* della matrice *allocation*), *finish[i] = false* altrimenti.
2. Cerca *i* tale che:
   finish[i] = false
   request[i][] <= work
   se tale *i* non esiste vai al passo 4.
3. *work = work + allocation[i][]*
   finish[i] = true
   vai al passo 2.
4. se *finish[i] == false* per qualche *i* allora il sistema è in stallo ed i processi in stallo sono quelli per i quali *finish[i] == false*.

L'algoritmo ha complessità mn^2 e può essere invocato ad ogni richiesta di una risorsa da parte di un processo (essendo questa la condizione che può chiudere un ciclo di stallo) o quando la percentuale d'uso della CPU scende sotto un certo valore (es., 40%).