SISTEMI OPERATIVI IIN, IEL, IDT, SAUT

seconda prova in itinere - 23.06.2009

Nome: _____ Cognome: _____ CDL:_____

Esercizio 1. (punti 20)

In un sistema sono presenti 3 istanze di una risorsa di tipo A e 2 istanze di una risorsa di tipo B. Tre Thread T_1, T_2 e T_3, eseguono un ciclo continuo in cui per svolgere il loro lavoro necessitano rispettivamente delle seguenti risorse: 2A e 1B per T_1; 1A e 1B per T_2; 1A e 2B per T_3.

Una volta verificata la disponibilità delle risorse, i thread le acquisiscono chiamando i metodi `public ObjectA prendiA()` e `public ObjectB prendiB()`, e le utilizzano usando le chiamate ai metodi statici della classe `Work`, `public void useAB(ObjectA A, ObjectB B)`, `public void use2AB(ObjectA A, ObjectA A1, ObjectB B)` e `public void useA2B(ObjectA A, ObjectB B, ObjectB B1)` (la classe `Work` ed i suoi metodi sono supposti dati).

Il numero di esecuzioni che cumula quelle di T_1, T_2 e T_3 deve essere di 10 dopo di che il programma deve terminare.

La soluzione dovrà utilizzare i semafori *Java* per risolvere i problemi di sincronizzazione tra thread.

Esercizio 2. (punti 10)

Utilizzando i dati del problema precedente per numero di thread, disponibilità delle risorse e richiesta massima di risorse dei thread, verificare:

- se lo stato iniziale dei processi è sicuro (si considera come stato iniziale quello in cui nessuna risorsa è allocata ai processi);
- se sono sicuri gli stati in cui i thread si possono portare considerando separatamente le richieste di T_1, T_2 e T_3 a partire dallo stato iniziale.

Soluzione

Esercizio 1.

Nel seguito si ipotizza che i metodi `prendiA()` e `prendiB()` siano ancora metodi statici della classe Work. Il testo per gli indirizzi di Elettronica, Telecomunicazioni e Specialistica in Automazione non richiedevano il conteggio di 10 esecuzioni. In quel caso la soluzione può essere desunta da quella presentata omettendo la gestione del conteggio.

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

public class Principale {
	public static final int N = 10;

	public static void main( String[] args ) {
		Conta count = new Conta( N );
		Semaphore sA = new Semaphore( 3 );
		Semaphore sB = new Semaphore( 2 );
		Thread1 t1 = new Thread1( sA, sB, count );
		t1.start();
		Thread2 t2 = new Thread2( sA, sB, count );
		t2.start();
		Thread3 t3 = new Thread3( sA, sB, count );
		t3.start();
		System.out.println( "main termina" );
	}
}

class Conta {
	private Semaphore mutex;
	private int count;
	private int max;

	public Conta( int n ) {
		mutex = new Semaphore( 1 );
		count = 0;
		max = n;
	}

	public boolean itera() {
		boolean itera = true;
		try {
			mutex.acquire();
			if ( count < max ) {
				count ++;
			}
			else {
				itera = false;
			}
			mutex.release();
		}
		catch (InterruptedException e) {
		}
		return itera;
	}
}

class Thread1 extends Thread {
	private Semaphore sA;
	private Semaphore sB;
	private Conta c;
```

```
	public Thread1( Semaphore sA, Semaphore sB, Conta c ) {
		this.sA = sA;
		this.sB = sB;
		this.c = c;
	}

	public void run() {
		while ( c.itera() ) {
			// verifica la disponibilità delle risorse
			try {
				sA.acquire( 2 );
				sB.acquire();
			}
			catch (InterruptedException e) {
			}

			// prende le risorse e le usa
			ObjectA A = Work.prendiA();
			ObjectA A1 = Work.prendiA();
			ObjectB B = Work.prendiB();
			Work.use2AB( A, A1, B );

			// rlascia la disponibilità delle risorse
			sA.release( 2 );
			sB.release();

			System.out.println( "Thread 1 ha eseguito" );
		}
	}
}

class Thread2 extends Thread {
	private Semaphore sA;
	private Semaphore sB;
	private Conta c;

	public Thread2( Semaphore sA, Semaphore sB, Conta c ) {
		this.sA = sA;
		this.sB = sB;
		this.c = c;
	}

	public void run() {
		while ( c.itera() ) {
			// verifica la disponibilità delle risorse
			try {
				sA.acquire();
				sB.acquire();
			}
			catch (InterruptedException e) {
			}
			// prende le risorse e le usa
			ObjectA A = Work.prendiA();
			ObjectB B = Work.prendiB();
			Work.useAB( A, B );

			// rlascia la disponibilità delle risorse
			sA.release();
			sB.release();

			System.out.println( "Thread 2 ha eseguito" );
		}
```

```
	}
}

class Thread3 extends Thread {
	private Semaphore sA;
	private Semaphore sB;
	private Conta c;

	public Thread3( Semaphore sA, Semaphore sB, Conta c ) {
		this.sA = sA;
		this.sB = sB;
		this.c = c;
	}

	public void run() {
		while ( c.itera() ) {
			// verifica la disponibilità delle risorse
			try {
				sA.acquire();
				sB.acquire( 2 );
			}
			catch (InterruptedException e) {
			}
			// prende le risorse e le usa
			Object A = Work.prendiA();
			Object B = Work.prendiB();
			Object B1 = Work.prendiB();
			Work.useA2B( A, B, B1 );

			// rlascia la disponibilità delle risorse
			sA.release();
			sB.release( 2 );

			System.out.println( "Thread 3 ha eseguito" );
		}
	}
}
```

Esercizio 2.

	Allocate	*Massimo*	*Need*	*Disponibili*
	A B	A B	A B	A B
Thread 1	0 0	2 1	2 1	3 2
Thread 2	0 0	1 1	1 1	
Thread 3	0 0	1 2	1 2	

- lo stato iniziale è sicuro dato che applicando l'algoritmo del banchiere è possibile trovare la sequenza sicura: $<T_1 , T_2 , T_3>$.
- La richiesta iniziale di T_1 è: $r_1 = (2,1)$ che soddisfa i limiti su *Need* e *Disponibili*. Ipotizzando di allocare queste risorse si passa nello stato:

	Allocate	*Massimo*	*Need*	*Disponibili*
	A B	A B	A B	A B
Thread 1	2 1	2 1	0 0	1 1
Thread 2	0 0	1 1	1 1	
Thread 3	0 0	1 2	1 2	

che ammette come sequenza sicura ancora $<T_1 , T_2 , T_3>$.

In modo simile si prova che anche gli stati raggiunti a partire dallo stato iniziale a seguito delle richieste di T_2 e T_3 sono sicuri.