PROGETTO JAVA

PRIMA PARTE

La prima parte del progetto consiste nella implementazione del meccanismo di sincronizzazione tipico dei monitor (mutua esclusione e variabili condition) che, con riferimento alla primitiva signal, utilizzi la semantica signal-and urgent.
Per l'implementazione del meccanismo devono essere utilizzati esclusivamente i costrutti di sincronizzazione offerti da Java 1.4.2 (e cioè i metodi e/o i blocchi **`synchronized`** e i metodi **`wait()`**, **`notify()`** e **`notifyAll()`** offerti dalla classe **`Object`**).

In altri termini, si chiede di implementare un meccanismo simile a quello direttamente offerto dalle versioni più recenti di java ma con una diversa semantica.
In particolare, il meccanismo direttamente offerto dalle versioni più recenti di java consente di utilizzare oggetti di tipo **`Lock`** per programmare esplicitamente la mutua esclusione fra le esecuzioni dei metodi di accesso ad un oggetto condiviso fra più threads. Inoltre, oggetti di tipo **`Lock`** consentono anche di dichiarare variabili di tipo **`Condition`** su cui eseguire le operazioni primitive **`c.await()`**, **`c.signal()`** e **`c.signalAll()`** usate rispettivamente per bloccare un thread sulla variabile **`Condition c`**, per svegliare un thread bloccato sulla variabile **`Condition c`**, o per svegliare tutti i thread sospesi su tale variabile. In questo caso però la semantica offerta da java è la signal-and-continue.
Nel progetto, si chiede di fornire un analogo meccanismo cambiando però la semantica offerta, secondo, le specifiche indicate di seguito.

REQUISITO N.1: implementare un meccanismo di mutua esclusione da utilizzare per programmare esplicitamente la mutua esclusione fra le esecuzioni dei metodi di accesso ad un oggetto condiviso fra più threads. Cioè, a differenza della mutua esclusione implicita offerta dai blocchi synchronized, la garanzia di mutua esclusione è demandata al programmatore e cioè programmata esplicitamente. In particolare: implementare la classe **`FairLock`** che offra i due metodi, **`lock()`** ed **`unlock()`** da usare per programmare esplicitamente gli accessi esclusivi ad un oggetto condiviso.

REQUISITO N.2: a differenza del *wait set* di un oggetto (contenente i thread sospesi per mutua esclusione nell'accesso ad un blocco synchronized) che, come noto, non prevede nessun tipo di gestione, i thread che si sospendono nel tentativo di accedere ad un oggetto condiviso eseguendo il metodo **`mutex.lock()`** (dove **`mutex`** rappresenta un'istanza della classe **`FairLock`** utilizzata per garantire la mutua esclusione negli accessi all'oggetto condiviso) devono essere risvegliati in ordine FIFO.

REQUISITO N.3: implementare la classe **`Condition`** che offre i metodi **`await()`** e **`signal()`** usati, rispettivamente, per sospendere un thread su una variabile **`Condition`** e per risvegliare il primo che si è sospeso (se c'è) fra i thread presenti sulla variabile **`Condition`**. Al solito, se nessun thread è sospeso sulla variabile **`Condition`**, il metodo **`signal()`** non esegue nessuna operazione. Quindi la **`Condition`** deve fornire una coda FIFO di thread sospesi. La semantica della signal deve essere la signal.and.urgent. Per questo motivo, non è prevista l'implementazione del metodo **`signalAll()`**.

REQUISITO N.4: Come noto dalla teoria dei monitor, una variabile **`Condition`** è sempre locale ad un monitor e le operazioni primitive **`await()`** e **`signal()`** possono essere quindi utilizzate solo all'interno dei metodi di accesso ad un monitor. Ciò implica che ogni istanza della classe **`Condition`** deve essere intrinsecamente legata ad un lock (oggetto della classe **`FairLock`**). Per questo motivo la classe **`FairLock`** deve offrire, oltre ai metodi **`lock()`** ed **`unlock()`** prima visti, anche il metodo **`newCondition()`**. In questo modo, se ad esempio **`mutex`** è un istanza

della classe **`FairLock`**, l'operazione **`mutex.newCondition()`**restituisce un'istanza di **`Condition`** legata allo specifico oggetto **`mutex`** di tipo **`Lock`**.

SECONDA PARTE
Scegliere uno fra i vari problemi di sincronizzazione visti anche a lezione e che preveda soluzioni diverse a seconda della semantica della signal utilizzata.
Quindi, risolvere il problema utilizzando il meccanismo implementato nella prima parte del progetto.
Successivamente, risolvere lo stesso problema utilizzando però gli strumenti di sincronizzazione offerti dalle versioni più recenti di java (**`Lock`** , variabili **`Condition`** e i metodi **`await()`**,**`signal()`** e **`signalAll()`**) che utilizzano la semantica signal and urgent, e verificare come deve essere modificata la soluzione rispetto a quella che utilizza il meccanismo di sincronizzazione implementato nel progetto.

VALUTAZUIONE
Per consentire la valutazione del progetto, devono essere consegnati:

- tutti i file sorgente (.java)
- tutti i file oggetto (.class)
- una relazione (file di tipo testo).

La relazione deve sinteticamente, ma chiaramente, riportare come è stato strutturato il progetto, indicando quali problemi sono stati affrontati, con quali soluzioni sono stati risolti e, se del caso, dare una giustificazione della soluzione scelta rispetto ad altre possibili soluzioni. La relazione è importante per capire meglio come il progetto è stato svolto.

I parametri utilizzati per la valutazione del progetto sono i seguenti:
1. Correttezza della soluzione
2. Chiarezza della relazione
3. Casi di test eseguiti
4. Struttura del progetto