Université Paris 7
M2 II
Programmation Répartie

Exam

Lundi 25 mars–15h à 16h

Aucun document n'est autorisé. Les 3 exercices sont indépendants. Le barème est donné à titre indicatifs.

Exercice 1.— (6 points). On suppose que l'on a utilisé la classe ReentrantReadWriteLock pour implementer ReadWriteLock. Vous trouverez dans ANNEXE à la fin de ces pages un bref rappel sur cette classe.

On a 5 threads (nommés A,B,C,X,Y) qui utilisent un verrou V ReadWriteLock crée par `ReentrantReadWriteLock()`. Les 3 threads A, B et C ont pour programme:

```
V.ReadLock.lock(); CODE1; V.ReadLock.unlock();
```

et les 2 threads X et Y ont pour programme:

```
V.WriteLock.lock(); CODE2; V.WriteLock.unlock();
```

1. A un instant donné, combien de threads au maximum peuvent être en train d'exécuter CODE1?
2. A un instant donné, combien de threads au maximum peuvent être en train d'exécuter CODE2?
3. A un instant donné, combien de threads au maximum peuvent être en train d'exécuter CODE1 et CODE2?
4. La thread A exécute maintenant

   ```
   while (true) {V.ReadLock.lock(); CODEA; V.ReadLock.unlock();}
   ```

   La thread X exécute maintenant

   ```
   while (true) {V.WriteLock.lock(); CODEX; V.WriteLock.unlock();}
   ```

   Le code des autres threads ne changent pas.

   On exécutent les 5 threads.

   (a) Est-il possible que CODE1 ne soit jamais exécuté?

   (b) Est-ill possible que CODEA ne soit jamais exécuté?

(c) Est-il possible que CODE2 ne soit jamais exécuté?

(d) Est-il possible que CODEX ne soit jamais exécuté?

(e) si CODEA est exécuté, est-il possible que CODE1 ne soit pas exécuté?

(f) si CODEX est exécuté, est-il possible que CODE2 ne soit pas exécuté?

Exercice 2.— (8 points). *Pour cet exercice, pour chaque question, vous rappellerez la définition du terme en italique (1 à 3 lignes) et justifierez vos réponses en quelques lignes (1 à 6 lignes).*

On réalise une implémentation pour les fonctions `scan` et `update` du snapshot de la manière suivante: On ajoute au tableau un registre `R`. Pour réaliser *update* on incrémente `R` de 1 et on met à jour le tableau. Pour réaliser le *scan* on lit `R`, on lit toute les valeurs du tableau et on relit `R`. Si la valeur de `R` a changé on recommence, sinon on retourne les valeurs du tableau qu'on vient juste de lire.

1. Est-ce que l'implémentation proposée est une *implémentation linéarisable (atomique)* du snapshot?
2. Cette implémentation est-elle *wait-free*?
3. Cette implémentation est-elle *non-blocking*?
4. Cette implémentation est-elle *obstruction-free*?

Exercice 3.— (6 points). On considère un objet atomique Test qui a une opération `Boolean t()`. Sa spécification séquentielle est : la première opération `t` retourne `true` les autres retournent `false`.

- Est-il possible de faire du consensus avec des objets de type Test et des registres pour 2 threads? (si oui donnez l'algorithme si non faites une preuve)
- Est-il possible de faire une implémentation wait-free de Test avec des registres ? (si oui donnez l'algorithme si non faites une preuve)
- Est-il possible de faire une implémentation obstruction-free de Test avec des registres ? (si oui donnez l'algorithme, si non faites une preuve)

ANNEXE

public class ReentrantReadWriteLock extends Object implements ReadWriteLock, Serializable

Constructor Summary

- ReentrantReadWriteLock()

  Creates a new ReentrantReadWriteLock with default (nonfair) ordering properties.

- ReentrantReadWriteLock(boolean fair)

  Creates a new ReentrantReadWriteLock with the given fairness policy.

Acquisition order

This class does not impose a reader or writer preference ordering for lock access. However, it does support an optional fairness policy.

- *Non-fair mode (default)*

  When constructed as non-fair (the default), the order of entry to the read and write lock is unspecified, subject to reentrancy constraints. A nonfair lock that is continously contended may indefinitely postpone one or more reader or writer threads, but will normally have higher throughput than a fair lock.

- *Fair mode*

  When constructed as fair, threads contend for entry using an approximately arrival-order policy. When the currently held lock is released either the longest-waiting single writer thread will be assigned the write lock, or if there is a group of reader threads waiting longer than all waiting writer threads, that group will be assigned the read lock. A thread that tries to acquire a fair read lock (non-reentrantly) will block if either the write lock is held, or there is a waiting writer thread. The thread will not acquire the read lock until after the oldest currently waiting writer thread has acquired and released the write lock. Of course, if a waiting writer abandons its wait, leaving one or more reader threads as the longest waiters in the queue with the write lock free, then those readers will be assigned the read lock.

  A thread that tries to acquire a fair write lock (non-reentrantly) will block unless both the read lock and write lock are free (which implies there are no waiting threads). (Note that the non-blocking ReentrantReadWriteLock.ReadLock.tryLock() and ReentrantReadWriteLock.WriteLock.tryLock() methods do not honor this fair setting and will acquire the lock if it is possible, regardless of waiting threads.)