Université Paris 7
M2 II
Programmation Répartie

Exam

Lundi 24 mars–15h30 à 16h30

Aucun document n'est autorisé sauf les documents manuscrits de votre main. Les 4 exercices sont indépendants. Le barème est donné à titre indicatif.

Exercice 1.— (4 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:

```
while (true) {V.ReadLock.lock(); CODE1; V.ReadLock.unlock();}
```

et les 2 threads X et Y ont pour programme:

```
while (true) {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. Est-il possible que CODE1 ne soit jamais exécuté?
5. Est-il possible que CODE2 ne soit jamais exécuté?

Exercice 2.— (5 points) On suppose que l'on dispose de registre SRSW régulier booléen et on veut implementer des registres SRSW régulier pouvant contenir des entiers. Basiquement on utilise un tableau de registres booléens tel que si on veut écrire la valeur i on écrit vrai dans le $i$eme élément du tableau. On lit le registre en faisant une lecture des différentes éléments du tableau en commençant à l'indice 0 jusqu'à trouver une valeur vrai. Ecrire les deux méthodes écrire et lire. Justifiez en quelques lignes que votre implémentation est correcte

Exercice 3.— (5 points) On suppose que vous avez réalisé une implémentation atomique et wai-free pour les fonctions `scan` et `update` du snapshot. Par ailleurs, vous avez réalisé une fonction `collect` qui fait une lecture séquentielle des éléments du tableau.

```
public interface Snapshot<T> {
    public void update(T v);
    public T[] scan();
    public T[] collect();
}
```

On suppose que le tableau est de taille n ($n > 2$) et est initialisé avec -1. On suppose que la thread i exécute:

```
update(i); col=collect();sc=scan();
```

On notera col_i (resp. sc_i)la valeur de la variable col (resp. sc) à la fin de l'exécution de la thread i.

Si X et X' sont deux tableaux, $X \subseteq X'$ si et seulement si ($\forall i \;\; X[i] \neq -1 \Rightarrow X[i] = X'[i]$)

Soient deux threads i et j:

1. A-t-on $col_i \subseteq col_j$ ou $col_j \subseteq col_i$?
2. A-t-on $sc_i \subseteq sc_j$ ou $sc_j \subseteq sc_i$?
3. A-t-on $col_i \subseteq sc_i$?
4. Garde-t-on une implémentation atomique et wait free si on implémente le `scan()` en faisant des `collect` jusqu'à ce que deux `collect` retourne le même tableau ?

Exercice 4.— (6 points) Dans cet exercice tous les objets sont atomiques et wait-free. On considère l'objet `SymetryBreaking`. Cet objet n'a qu'une seule méthode $sb()$ qui retourne 0 ou 1. Sa spécification séquentielle est : le premier appel de $sb()$ retourne 0 les autres retournent 1.

1. En utilisant des registres atomiques et des objets `SymetryBreaking` réaliser du consensus pour 2 processus. (On rappelle que le consensus est un objet avec une méthode $propose(int\ v)$ qui retourne un entier et dont la spécification est : tous les $propose$ retournent le paramètre du premier $propose$)
2. Montrer qu'en utilisant des objets `SymetryBreaking` et des registres atomiques, on ne peut pas réaliser du consensus pour 3 processus.
3. Est-il possible de réaliser un objet `SymetryBreaking` à l'aide de registres atomiques et de `CompareAndSet`? (On rappelle que cet objet n'a qu'une méthode $cas(int\ v, int\ w)$ qui retourne un booléen et dont la spécification séquentielle est: si la valeur de l'objet est égale à v alors elle devient w et la méthode retourne $true$ sinon elle retourne $false$ et l'objet ne change pas de valeur.)
4. Est-il possible de réaliser un `CompareAndSet` à l'aide de registres atomiques et d'objets `SymetryBreaking`?

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.)