POLYTECH / INFO 4ème année

07/01/2020
Durée : 1h30, Documents écrits autorisés (sauf livres)
Le barème donné est indicatif.

Examen de Programmation Concurrente

Note : Dans le sujet, et dans vos réponses, on s'autorisera à utiliser indifféremment l'opération P() ou acquire() (resp. V() ou release()) pour les sémaphores. Par ailleurs, on supposera que les programmes s'exécutent bien, on ne cherchera pas à gérer les cas d'erreurs.

Questions de connaissance générale (4 pts).

a) Dans un système à multi-programmation, quelles situations amènent un processus à être commuté ? citez au moins 2 cas.
b) Quels types d'isolation sont garantis par les processus ?
c) La clause *synchronized* interdit-elle les commutations de contexte ?
d) Quelle est la principale différence entre un appel à *wait()* dans un moniteur, et un appel à *acquire()* (ou P()) sur un sémaphore ? (5 lignes max)
e) Quelle est la caractéristique d'un thread de type démon ? (3 lignes max)
f) Pour quelle raison une exécution multi-threadée n'est pas systématiquement plus efficace qu'une exécution mono-threadé ? (5 lignes max)
g) Qu'est ce qui caractérise une variable *volatile* ? (2 lignes max)
d) Les moniteurs sont-ils ré-entrants ?

Exercice 1 (1 pt).

Les deux classes suivantes sont-elles équivalentes d'un point de vue comportement concurrent? Si la réponse est non, préciser.

```
class Count1 {
    int count;
    static int scount;

     synchronized void inc(){
          count++;
     }

     static synchronized void sinc(){
        scount++;
     }
 }
```

```
class Count2 {
 int count;
 static int scount;

 void inc(){
     synchronized(this){ count++;}
 }

 static void sinc() {
     synchronized(this.getClass()){
       scount++;
     }
 }
}
```

Exercice 2 (1,5 pt).

En quoi le comportement des trois méthodes suivantes diffère? (max 10 lignes)

```
public void M1 () {
   while(!cond()) ;
   System.out.println("done");
}
```

```
public void M2 () {
   while(!cond())
      Thread.yield();
   System.out.println("done");
}
```

```
public synchronized void M3 () {
 while(!cond()) {
   try {
       wait();
   }catch(InterruptedException e) {}
 }
 System.out.println("done");
}
```

Exercice 3 (1,5 pt).

On considère la classe suivante, dans laquelle la méthode *getInst()* crée une instance de la classe *A* seulement si aucun instance n'existe déjà. Cette méthode doit pouvoir être invoquée par des threads concurrents, il faut donc que la classe *A* soit thread-safe. Dans les versions suivantes, quelles sont les versions thread-safe ? justifiez vos réponses (en 3 ou 4 lignes), qu'elles soient positives ou négatives.

```
public class A {    // version 1
private static A inst;

public static synchronized A
getInst(){
   if (inst == null)
       inst = new A();
   return inst;
}
```

```
public class A {   // version 2
private static volatile A inst;

public static A getInst(){
   if (inst == null)
        inst = new A();
   return inst;
}
```

```
public class A {   // version 3
private static volatile A inst;

public static A getInst(){
   if (inst == null)
       synchronized(this) {
         if (inst == null)
              inst = new A();
       }
   }
   return inst;
}
```

```
public class A {   // version 4
private static A inst;

public static A getInst(){
   if (inst == null)
      synchronized(this) {
       if (inst == null)
           inst = new A();
      }
   return inst;
}
```

Exercice 4 (1 pt).

La classe suivante n'est pas thread-safe à cause d'erreurs dans sa programmation. Identifiez et corrigez ces erreurs.

```
public class Stack {
   LinkedList list = new LinkedList();
   public void push(Object o) {
      synchronized(list) {
         list.addLast(o);
         notify();
      }
   }
   public Object pop() throws InterruptedException {
      synchronized(list) {
        while (list.isEmpty())
          wait();
        return list.removeLast();
      }
   }
}
```

Exercice 5 (9 pts).
Soit un type de ressource **R** accédée par des threads à l'aide de trois opérations, notées **OP0**, **OP1** et **OP2**, dont les contraintes d'exécution concurrente sont données par le tableau suivant :

	OP_0	OP_1	OP_2
OP_0	oui	non	non
OP_1	non	oui	non
OP_2	non	non	non

*Un "**oui**" (respectivement "**non**") en position (i, j) du tableau signifie que l'exécution concurrente des opérations **Opi** et **Opj** est autorisée (respectivement interdite).*

On souhaite programmer une classe R gérant l'accès à une ressource de type R. Cette classe expose les méthodes *beginOP0, endOP0, beginOP1, endOP1, beginOP2, endOP2***.** Toute méthode *beginOPi()* est censée être appelée avant l'exécution de l'opération *OPi*, et toute méthode *endOPi()* appelée après l'exécution de l'opération *Opi* (i=0, 1 ou 2). Ces méthodes sont chargées de garantir les contraintes d'exclusion exprimées par le tableau précédent.

Question a. Après avoir donné les variables caractérisant l'état du système, donner le tableau de gardes et actions**.**

Question b. En déduire la solution directe en Java. Caractériser cette solution (ordonnancement, efficacité).

Question c*. Proposez une adaptation de la solution directe qui respecte un ordre FIFO sur l'accès à la ressource R.

Question d*. Proposez une adaptation de la solution directe qui donne la priorité aux opérations OP_2 puis OP_1 puis OP_0.

(*) : vous prendrez soin de remplacer les *notifyAll()* par des *notify(),* lorsque c'est possible.

Question e. On considère le programme suivant, qui :
- crée K ressources de type R (K > 1),
- crée et démarre N threads de classe *RThread*, qui vont manipuler les ressources précédemment créées,
- attend la terminaison des N threads, puis affiche le message « all done ».

Complétez la classe P pour ce faire.

```
class P {
   static final int K = ..;
   static final int N = ..;
   <to complete>

  public static void main(String args[]) {

      // creation de K ressources de type R
        for (int i=0; i<K; i++)
             ressources[i] = new R();

      // creation et lancement de N threads
      // qui vont manipuler les ressources
      <to complete>

      // program ending
      <to complete>
  }
```

```
class RThread implements Runnable {
   <to complete>

   public RThread(<to complete>){ <to
complete> }

   public void run(){
        …
   }
```

Question f. Proposez une solution simple permettant de borner à 5 secondes le temps d'attente durant lequel le thread principal attend la terminaison des N threads créés.

Question g. Le programme de la question e) met en jeu un ensemble de threads qui accèdent un ensemble de ressources de type R, au travers des opération *beginOPi(), endOPi()*. En l'absence de gestion particulière, la survenue d'inter-blocages est possible. Proposez un scénario illustrant un inter-blocage.