POLYTECH / RICM 4ème année

11/1/2019
Examen de Programmation Concurrente
Durée : 1h30, Documents écrits autorisés (sauf livres)
Le barème et le temps donné sont indicatifs.

Question 1. Shell (2 pts)

A) Les shells sont mis en œuvre sous la forme de programmes multi-processus. Serait-il envisageable de les mettre en œuvre sous la forme de programmes multi-threadés ? quelles seraient les conséquences de ce choix d'implantation ? (5 lignes max)

B) Les shells proposent un certain nombre de commandes intégrées appelées *built-in*, qui sont exécutées par le shell lui-même, comme vous l'avez vu en TP. On trouve parmi celles-ci : *cd, alias, login, bg, echo,...* Expliquez pourquoi *cd* ne peut être qu'une commande built-in, alors que *ls* n'en est pas une. (5 lignes max)

Question 2 – Interblocages (3 points)

Quatre threads numérotés de 1 à 4 partagent des ressources de 3 types différents (A, B ou C). Il existe 3 ressources de type A, 5 de type B et 2 de type C. Au cours de son exécution, chaque thread a besoin d'un nombre maximal de ressources donné par la table MAX. A un instant donné, les ressources détenues par les threads sont données par la table ALLOC.

MAX	**A**	**B**	**C**
P1	**1**	**2**	**2**
P2	**0**	**3**	**0**
P3	**3**	**2**	**1**
P4	**2**	**1**	**0**

ALLOC	**A**	**B**	**C**
P1	**1**	**2**	**1**
P2	**0**	**1**	**0**
P3	**1**	**0**	**0**
P4	**1**	**1**	**0**

a) On suppose que le système prévient les inter-blocages au travers de l'algorithme du banquier vu en cours. Pour chacune des requêtes d'allocation de ressources listée dans la table ci-après, précisez quelle est la décision prise par l'algorithme.

Requête x) P1 : +1C
Requête y) P2 : +1B +1C
Requête z) P3 : +2B

b) Quelle est la contrainte principale qui fait que l'algorithme du banquier n'est pas applicable dans toute application ? (1 à 2 lignes).

Question 3 - Sémaphores et moniteurs - (3 pts)

On considère les éléments de synchronisation de base fournis par Java au travers des directives *synchronized*, *wait* et *notify*, comme illustré ci-après. On souhaite produire une mise en œuvre équivalente (bien que non ré-entrante), entièrement basée sur l'usage de sémaphores. En utilisant uniquement la classe Semaphore comme outil de synchronisation, reprogrammer M1() et M2().

```
-----------------------------------------------------------------
  public class X {
    public synchronized void M1 (…){
        while (! <condition>)
          wait();
    }
    public synchronized void M2 (…){
        <action>
        notify();
    }
  }
-------------------------------------------------------------------------------------
  public class X {   // mise en oeuvre équivalente
    <à compléter>              // sans synchronized, wait et notify
    public void M1 (…){
        <à compléter>
    }
    public void M2 (…){
        <à compléter>
    }
  }
```

Question 4 – Programmation concurrente (12 points)

On considère la mise en œuvre d'un moniteur Java permettant de gérer la transmission de *jetons* d'un thread à un autre dans une application concurrente. Un jeton est une ressource qui peut être libre, ou bien acquise par un thread à la fois. Dans les systèmes simples de gestion de jeton, un jeton peut simplement être *acquis* et *libéré* par un thread. Dans des systèmes plus complexes, un jeton acquis par un thread peut être aussi *transmis* à un autre thread. Des mécanismes de suivi du parcours du jeton peuvent également être implantés. Nous allons considérer différentes stratégies de gestion de jetons dans cet exercice.

Comme point de départ, nous supposons qu'il n'y a qu'un jeton à gérer et que celui-ci est représenté par une instance de la classe *Token* dont une première version est donnée ci-après. Par ailleurs, les threads de notre application, que nous appelerons *acteurs*, exécutent tous le même code défini par la classe *Actor* donnée ci-après. Durant l'exécution de la méthode *run(),* les acteurs acquièrent puis libèrent le jeton un certain nombre de fois.

Question A. La classe *Token* comporte plusieurs erreurs. Précisez ces erreurs et proposez une correction.

Question B. Donnez le code du programme principal de l'application chargé de créer N threads acteurs (N étant passé en argument du programme). On souhaite que l'application se termine lorsque tous les acteurs créés ont fini leur exécution.

Question C. On veut fournir une version de la classe *Token* qui assure que le jeton est alloué aux acteurs dans l'ordre de leurs demandes. Définissez la classe *TokenFifo* assurant cette allocation FIFO.

```
public class Actor implements Runnable {

 int id;    // identification of the current actor
 Token tok; // reference to the token
    ...
 public Actor(int id, Token tok){
       this.id = id;
       this.tok = tok;
   }

 public void run(){
      ..
      tok.get();
      ...
      tok.release();
      ..
      tok.get();
      ...
      tok.release();
      ..
 }
}

public class Token {
  boolean free = true;
  public void get(){
     if (free)
        free = false;
     else wait();
  }
  public void release(){
    free = true;
    notify();
  }
}
```

Question D. On considère maintenant une politique d'allocation tenant compte de priorités. On suppose que les acteurs peuvent être associés à une priorité haute ou faible. Définissez la classe *TokenPrio* fournissant les méthodes *get*(), *getPrio()* et *release*(), la méthode *getPrio()* permettant de récupérer le jeton avec une priorité haute (*get()* restant associé à une faible priorité).

Question E. On souhaite permettre à un acteur qui libère le jeton de préciser quel est le prochain acteur qui pourra récupérer ce jeton (on supposera qu'initialement le jeton est libre pour l'acteur d'indice 0). Pour cela, on considère la classe *TokenNext* qui fournit la

méthode *get(int id*), permettant au thread d'indice *id* de demander le jeton, ainsi que la méthode *release(int next),* donnant le jeton au thread d'indice *next*. Fournir le tableau définissant les gardes et actions des méthodes *release(int next)* et *get(int id)*. Définir ensuite la solution directe.

Question F. On souhaite étendre la solution précédente en proposant la classe *TokenSync* dans laquelle l'acteur qui libère le jeton précise quel est le prochain acteur qui doit récupérer ce jeton *et* se met en attente jusqu'à la demande de prochain acteur (si celle-ci n'est pas déjà faite). Pour réaliser la classe *TokenSync*, on propose d'utiliser une classe *Rdv* qui permet de mettre en oeuvre un rendez-vous entre 2 threads. Tout thread arrivant au rendez-vous est censé appeler la méthode *arrival(), le* premier thread arrivé restant bloqué jusqu'à l'arrivée du 2ième. Proposez une réalisation de la classe *TokenSync* basée sur l'usage de la classe *Rdv*.

```
public class Rdv()
  ...
  public synchronized void arrival() {...}
}
```

Question G. (indépendante). Proposez une réalisation de la classe *Rdv* mettant en oeuvre un Rendez-vous entre deux threads (tel que décrit dans la question précédente). La solution sera basée sur l'usage de sémaphores Java.

Question H. On veut gérer le cas dans lequel l'acteur étant supposé récupérer le jeton n'arrive jamais. Pour cela, un thread *veilleur* est créé lors de l'initialisation du programme, et chargé de vérifier, après chaque libération du jeton, que ce dernier est bien récupéré au bout d'un délai fixé à 5 secondes. Si, au bout de ce temps, le jeton n'est toujours pas récupéré, le veilleur affiche un message d'erreur et provoque une sortie forcée du programme. On vous demande de (re)définir 1) le code du programme, 2) la classe *Veilleur* définissant le code du veilleur et 3) les méthodes modifiées de la classe *TokenNext*.

Question I. On considère maintenant qu'il y a N jetons gérés par notre application. Chaque acteur reçoit en argument de son constructeur les références de tous les jetons, comme illustré ci-après. Chaque acteur enchaîne des phases d'acquisition et des phases de libération des jetons, comme illustré ci-après.

```
public class Actor {

 int id; // identification of the current actor
 Token[] tokens; // token references
   ...
 public Actor(int id, Token[] tokens){
       this.id = id;
       this.tokens = tokens;
   }

 public void run(){
      ..
      tokens[x].get();
      tokens[y].get();
      ..
      tokens[x].release();
      tokens[y].release();
      ..
      tokens[z].get();
      tokens[y].get();
      ..
      tokens[z].release();
      tokens[y].release();
       ...

 }
}
```

A) Quel problème majeur est posé par cette version de l'application ?
B) Quelle solution proposeriez-vous pour gérer ce problème ?

```
 Semaphore mutex   = newSemaphore(1);
 Semaphore attente = new Semaphore(0);
 int nb_att = 0;
public void M1 (…) throws Exception {
     mutex.P();
     while (!<condition>)  { nb_att++ ; mutex.V();attente.P()(); mutex.P() ;}
     mutex.V();
 }
public void M2 (…) throws Exception {
     mutex.P();
     if (nb_att>0) { nb_att--; attente.V();}
     mutex.V();
 }
```

question de cours

sc interdit commutations

synchronized et mutex.P() avec mutex initialisé à 1 ....dire si équivalent

usage processus / threads, critères

rdv avec semaphore

collections synchronisees - iterateur renvoie exception si modif

interblocages

chercher erreur

Question 2 – Chercher l'erreur (3 pts). SEE 2013, PB ROND POINT