Université Bordeaux I
U.F.R. Mathématiques et Informatique

IUP MIAGe 3^{eme} année
Année 2001-2002

Mercredi 17 avril, 10 heures - 12 heures
Tous documents autorisés.

Programmation Parallèle
Alain Griffault

SUJET + CORRIGE

---

1 Ordonnancement et partage de ressources (6 points)

L'algorithme d'ordonnancement

Les sources suivantes correspondent à celles étudiées en cours et en travaux dirigés. L'ordonnancement des tâches est obtenu en mettant un sémaphore par arête du graphe.

Le fichier `Ordonnancement.java`

```
1   import java.util.*;
2
3   class OrdonnancementArc {
4       public static void main(String []args) {
5           System.out.println("DEBUT");
6           Graphe graphe = new Graphe();
7           int nbProcessus = graphe.getNombreSommets();
8           Semaphore[][] sem = new Semaphore[nbProcessus][nbProcessus];
9           for (int i=0; i<nbProcessus; i++)
10              for (int j=0; j<nbProcessus; j++)
11                  if (graphe.isArcInitial(i,j))
12                      sem[i][j] = new Semaphore(1);
13                  else if (graphe.isArc(i,j))
14                      sem[i][j] = new Semaphore(0);
15          Random alea = new Random();
16          Tache[] tache = new Tache[nbProcessus];
17          for (int i=0; i<nbProcessus; i++) {
18              tache[i] = new Tache(i,alea.nextInt(200),graphe,sem);
19              tache[i].start();
20          }
21          for (int i=0; i<nbProcessus; i++) {
22              try {tache[i].join();}
23              catch (InterruptedException e) {}
24          }
25          System.out.println("\nFIN ");
26      }
27  }
```

Le fichier `Semaphore.java`

```
1   class Semaphore {
2       private int n = 1;
3       Semaphore(int val){
```

```
4           if (val >= 0)
5               n = val;
6       }
7       synchronized void p() { // autre nom de wait()
8           n--;
9           if (n < 0)
10              try{wait();}
11              catch(InterruptedException e){}
12      }
13      synchronized void v() { // autre nom de signal()
14          n++;
15          if (n <= 0)
16              notify();
17      }
18  }
```

Le fichier `Tache.java`

```
1   class Tache extends Thread {
2       Semaphore[][] sem;
3       Graphe graphe;
4       int threadId;
5       int duree;
6
7       Tache(int id, int duree, Graphe graphe, Semaphore[][] sem) {
8           setName(Integer.toString(id));
9           threadId = id;
10          this.duree = duree;
11          this.graphe = graphe;
12          this.sem = sem;
13      }
14
15      public void run() {
16          // chaque tache est exécutée 10 fois
17          for (int occurence=0; occurence<10; occurence++){
18              // protocole de debut de tâche
19              for (int i=0; i<graphe.getNombreSommets(); i++)
20                  if (graphe.isArc(i,threadId))
21                      sem[i][threadId].p();
22              // corps de la tâche avec simulation de la durée
23              System.out.println("Tâche "+
24                                 currentThread().getName()+
25                                 " : début de l'occurence "+
26                                 occurence);
27              try{sleep(duree);}
28              catch(InterruptedException e){}
29              System.out.println("              ... fin de la tâche "+
30                                 currentThread().getName());
31              // protocole de fin de tâche
32              for (int i=0; i<graphe.getNombreSommets(); i++)
33                  if (graphe.isArc(threadId,i))
34                      sem[threadId][i].v();
35          }
36      }
37  }
```

L'ordonnancement avec ressources limitées

Dans une atelier de production, les tâches sont non seulement contraintes par un ordonnancement, mais également par le fait que les ressources (humaines, matérielles, ...) sont en nombres limités.

Une solution consiste à utiliser des sémaphores pour chaque type de ressources initialisés au nombre d'exemplaires disponibles. Chaque tâche doit alors acquérir pour chaque type de ressources le nombre d'exemplaires nécessaires avant d'exécuter son travail, puis libérer les ressources une fois celui-ci terminé.

Les différences entre l'ancien (<) et le nouveau (>) `Ordonnancement.java`

```
15a16,26
>       // creation aléatoire des ressources
>       int nbTypeRessource = 1+alea.nextInt(4);
>       int[][] resNecessaires = new int[nbProcessus][nbTypeRessource];
>       Semaphore[] resDisponibles = new Semaphore[nbTypeRessource];
>       for (int i=0; i<nbTypeRessource; i++) {
>           // le nombre de ressources "i" disponibles
>           resDisponibles[i] = new Semaphore(2+alea.nextInt(2));
>           for (int j=0; j<nbProcessus; j++)
>               // le nombre de ressources "i" nécessaires à la tâche "j"
>               resNecessaires[j][i] = alea.nextInt(2);
>       }
18c29,30
<           tache[i] = new Tache(i,alea.nextInt(200),graphe,sem);
---
>           tache[i] = new Tache(i,alea.nextInt(200),graphe,sem,
>                                resNecessaires[i],resDisponibles);
```

Question 1.1 (4 points) *Ajouter dans le source suivant de la classe* `Tache` *les instructions suffisantes afin que les ressources soient correctement gérées.*

Réponse à la question 1.1: Le fichier `Tache.java`

```
1   class Tache extends Thread {
2
3       Semaphore[][] sem;
4       Graphe graphe;
5       int threadId;
6       int duree;
7       int[] resNecessaires;
8       Semaphore[] resDisponibles;
9
10      Tache(int id, int duree, Graphe graphe, Semaphore[][] sem,
11            int[] resNecessaires, Semaphore[] resDisponibles) {
12          setName(Integer.toString(id));
13          threadId = id;
14          this.duree = duree;
15          this.graphe = graphe;
16          this.sem = sem;
17          this.resNecessaires = resNecessaires;
18          this.resDisponibles = resDisponibles;
19      }
20
21      public void run() {
22          // chaque tache est exécutée 10 fois
23          for (int occurence=0; occurence<10; occurence++){
```

```
24              // protocole de debut de tâche
25              // autorisation, PUIS acquisition, sinon INTERBLOCAGE
26              // on attend l'autorisation de commencer
27              for (int i=0; i<graphe.getNombreSommets(); i++)
28                  if (graphe.isArc(i,threadId))
29                      sem[i][threadId].p();
30              // on acquière les ressources nécessaires
31              for (int i=0; i<resNecessaires.length; i++)
32                  for (int j=0; j<resNecessaires[i]; j++)
33                      resDisponibles[i].p();
34              // corps de la tâche avec simulation de la durée
35              System.out.println("Tâche "+
36                                 currentThread().getName()+
37                                 " : début de l'occurence "+
38                                 occurence);
39              try{sleep(duree);}
40              catch(InterruptedException e){}
41              System.out.println("              ... fin de la tâche "+
42                                 currentThread().getName());
43              // protocole de fin de tâche
44              // libération et signalisation dans un ordre quelconque
45              // on libère les ressources
46              for (int i=0; i<resNecessaires.length; i++)
47                  for (int j=0; j<resNecessaires[i]; j++)
48                      resDisponibles[i].v();
49              // on prévient indirectement les autres tâches
50              for (int i=0; i<graphe.getNombreSommets(); i++)
51                  if (graphe.isArc(threadId,i))
52                      sem[threadId][i].v();
53          }
54      }
55  }
```

Question 1.2 (2 points) *Penser vous que votre solution puisse engendrer des inter-blocages ? Si oui, donner une exécution conduisant probablement à cette situation, sinon, expliquer pourquoi votre solution prévient les inter-blocages.*

Réponse à la question 1.2: Oui et Non. En fait la réponse est plus complexe.

Oui s'ils existent : d'une part une ressource R disponible en n exemplaires; d'autre part deux tâches T_1 et T_2 pouvant s'exécuter simultanémént et désirant respectivement $n_1 > 1$ et $n_2 > 1$ exemplaires de R, et tels que $n < n_1 + n_2 - 1$.

Exemple : Posons $n = n_1 = n_2 = 2$ et considérons l'exécution suivante :

1. T_1 prend un premier exemplaire de R
2. T_2 prend le second exemplaire de R
3. interblocage

Non si la condition précédente n'est pas vérifiée. Dans ce cas, les ressources étant ordonnées dans un tableau, et tous les processus respectant cet ordre pour acquérir les ressources, il ne peut y avoir d'interblocage.

2 Le contrôle d'un musée (14 points)

Le cadre général

L'objectif de l'étude est la réalisation d'un contrôleur pour la gestion des visiteurs d'un musée. Le musée est composé de salles et de portes. Des portes permettent le passage (dans les deux sens) soit d'une salle à une autre, soit d'une salle vers l'extérieur. Chaque salle à une capacité maximale de visiteurs simultanés, et le contrôleur est chargé de veiller que ce nombre ne soit jamais dépassé.

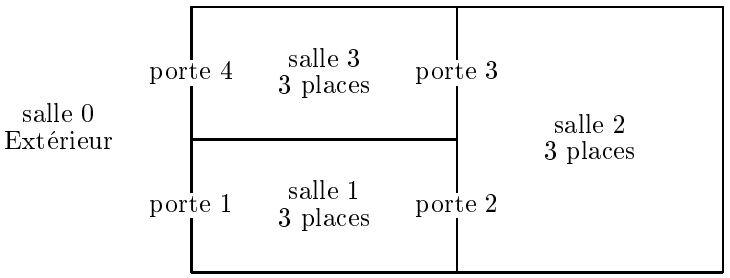

FIG. 1 – Un exemple de musée

Un modèle sans contrôleur

Les visiteurs sont implantés par des threads. Ils terminent lorsqu'ils ont visités toutes les salles du musée. L'implantation actuelle effectue un tirage aléatoire pour l'ordre de visite des salles. Il est donc possible que le programme ne termine pas. Dans toute la suite, ce fait ne sera pas considéré comme une erreur.

Le fichier `Visiteur.java`

```
1   import java.util.*;
2
3   class Visiteur extends Thread {
4       private static Random alea = new Random();
5       int salle;
6       int nbSalles;
7       boolean[] visitee;
8       boolean fini;
9
10      Visiteur(String who, int nb) {
11          setName(who);
12          nbSalles = nb;           // l'exterieur est codé comme une salle
13          salle = 0;               // le visiteur est initialement à l'extérieur
14          visitee = new boolean[nbSalles];
15          visitee[0] = true;       // l'extérieur est visité
16          for (int i=1; i<nbSalles; i++)
17              visitee[i] = false;  // aucune "vraie" salle n'est visitée
18      }
19
20      public void bouger() {
21          // il va au hasard dans une des deux salles voisines.
22          if (alea.nextBoolean())
23              salle = (salle+1)%nbSalles;
24          else
25              salle = (salle-1+nbSalles)%nbSalles;
26          visitee[salle] = true;
```

```
27          System.out.println(getName()+" dans la salle "+salle);
28          // il dort pendant les commentaires du guide
29          try {Thread.sleep(alea.nextInt(1000));}
30          catch (InterruptedException ie){}
31          // il a fini s'il a vu toutes les salles et si il est sorti du musée
32          if (salle == 0){
33              fini = true;
34              for (int i=0; i<nbSalles; i++)
35                  fini &= visitee[i];
36          }
37          else
38              fini = false;
39      }
40
41      public void run() {
42          while (!fini)
43              bouger();
44      }
45  }
```

Question 2.1 (2 points) *Ajouter dans le source suivant de la classe* `Musee` *les instructions suffisantes afin que le message* `FIN` *de la méthode* `main` *soit le dernier affiché lors de toute exécution.*

Réponse à la question 2.1: Le fichier `Musee.java`

```
1   import java.util.*;
2
3   class Musee {
4       public static void usage () {
5           System.err.println("syntaxe : java Musee NbSalles NbVisiteurs");
6       }
7
8       public static void main(String []args) {
9           boolean pasArgument = args.length != 2;
10          if (pasArgument)
11              usage();
12          else {
13              System.out.println("DEBUT");
14              int nbSalles = Integer.parseInt(args[0]);
15              int nbVisiteurs = Integer.parseInt(args[1]);
16              Visiteur[] visiteur = new Visiteur[nbVisiteurs];
17              for (int i=0; i<nbVisiteurs; i++)
18                  visiteur[i] = new Visiteur("Visiteur "+i+" ",nbSalles);
19              for (int i=0; i<nbVisiteurs; i++)
20                  visiteur[i].start();
21              try {
22                  for (int i=0; i<nbVisiteurs; i++)
23                      visiteur[i].join();
24              }
25              catch (InterruptedException ie){}
26              System.out.println("FIN");
27          }
28      }
29  }
```

Un premier modèle avec contrôleur

Dans ce modèle, chaque pièce du musée est limitée en capacité d'accueil, et un gardien contrôleur autorise ou bien fait attendre les visiteurs lorsqu'ils désirent changer de pièces.

Les différences entre l'ancien (<) et le nouveau (>) `Musee.java`

```
15a16,22
>           // on définit aléatoirement la capacité des pièces du musée
>           int[] capacite = new int[nbSalles];
>           capacite[0] = nbVisiteurs; // pour l'extérieur
>           Random alea = new Random();
>           for (int i=1; i<nbSalles; i++)
>               capacite[i] = alea.nextInt(nbVisiteurs/2)+1;
>           Controleur gardien = new Controleur(capacite);
18c25
<               visiteur[i] = new Visiteur("Visiteur "+i+" ",nbSalles);
---
>               visiteur[i] = new Visiteur("Visiteur "+i+" ",gardien,nbSalles);
```

Les différences entre l'ancien (<) et le nouveau (>) `Visiteur.java`

```
9,10c9,11
<
<     Visiteur(String who, int nb) {
---
>     Controleur controleur;
>
>     Visiteur(String who, Controleur gardien, int nb) {
11a13
>       controleur = gardien;
20a23
>       controleur.sortir(salle);
25a29
>       controleur.entrer(salle);
```

Question 2.2 (4 points) *Ajouter dans le source suivant de la classe* `Controleur` *les instructions suffisantes afin que les capacités d'accueil des pièces soient respectées.*

Réponse à la question 2.2: Le fichier `Controleur.java`

```
1   import java.util.*;
2
3   class Controleur {
4       int[] nbPlaces, nbVisiteurs;
5
6       public Controleur (int[] capacite) {
7           nbPlaces = capacite;
8           nbVisiteurs = new int[capacite.length];
9           nbVisiteurs[0] = nbPlaces[0];
10          for (int i=1; i<capacite.length; i++)
11              nbVisiteurs[i] = 0;
12      }
13
14      public synchronized void entrer(int salle){
15          while(nbVisiteurs[salle] == nbPlaces[salle]){
16              System.out.println(Thread.currentThread().getName()+ "attend");
17              try{wait();}
18              catch(InterruptedException ie){}
19          }
20          nbPlaces[salle]++;
21      }
22
23      public synchronized void sortir(int salle){
24          nbPlaces[salle]--;
25          notifyAll();
26      }
27
28  }
```

Question 2.3 (2 points) *Expliquer le défaut principal de cette solution. Indication : que deviennent les visiteurs lorsqu'ils attendent pour entrer dans une salle ?*

Réponse à la question 2.3: Ils ne sont ni dans la pièce d'où ils viennent, ni dans la pièce où ils désirent aller. Implicitement, les portes sont devenues des couloirs (tampons, buffers) de capacités infini.
Ce programme est donc assez peu réaliste.

Un second modèle avec contrôleur

Afin d'éviter le problème précédent, les visiteurs ne demandent plus les autorisations de `sortir` et d'`entrer` dans une salle, mais une seule autorisation plus générale de déplacement entre deux salles. Ils restent donc dans l'ancienne salle tant qu'ils ne peuvent entrer dans la nouvelle.

Les différences entre l'ancien (<) et le nouveau (>) `Visiteur.java`

```
23c23
<       controleur.sortir(salle);
---
>       int ancienneSalle = salle;
29c29
<       controleur.entrer(salle);
---
>       controleur.deplacer(ancienneSalle,salle);
```

Question 2.4 (3 points) *Ajouter dans le source suivant de la classe* `Controleur` *les instructions suffisantes afin que les capacités d'accueil des pièces soient respectées.*

Réponse à la question 2.4: Le fichier `Controleur.java`

```
1   import java.util.*;
2
3   class Controleur {
4       int[] nbPlaces, nbVisiteurs;
5
6       public Controleur (int[] capacite) {
7           nbPlaces = capacite;
8           nbVisiteurs = new int[capacite.length];
9           nbVisiteurs[0] = nbPlaces[0];
10          for (int i=1; i<capacite.length; i++)
11              nbVisiteurs[i] = 0;
12      }
13
14      public synchronized void deplacer(int from, int to){
15          while(nbVisiteurs[to] == nbPlaces[to]){
16              System.out.println(Thread.currentThread().getName()+ "attend");
17              try{wait();}
18              catch(InterruptedException ie){}
19          }
20          nbVisiteurs[from]--;
21          nbVisiteurs[to]++;
22          notifyAll();
23          // INTERBLOCAGE POSSIBLE !
24      }
25
26  }
```

Question 2.5 (3 points) *Expliquer le défaut principal de cette solution. Donner un exemple possible d'exécution conduisant à cette situation.*

Réponse à la question 2.5: Il y a risque d'inter-blocage. Dans certaines situations, le programme ne termine pas car certains visiteurs se gênent mutuellement.
Un exemple avec simplement deux visiteurs, et les salles 2 et 3 limitées à une place.

1. Le premier visiteur entre dans la salle 1.
2. Le second visiteur entre dans la salle 3.
3. Le premier entre dans la salle 2.
4. Le premier désire entrer dans la salle 3 $\rightarrow$ il est bloqué et endormi par le contrôleur.
5. Le second désire entrer dans la salle 2 $\rightarrow$ il est bloqué et endormi par le contrôleur.
6. C'est une situation d'inter-blocage.