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

IUP MIAGe 3eme année
Année 2001-2002

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

Programmation Parallèle
Alain Griffault

Numéro d'anonymat :

---

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`

```
import java.util.*;

class OrdonnancementArc {
    public static void main(String []args) {
        System.out.println("DEBUT");
        Graphe graphe = new Graphe();
        int nbProcessus = graphe.getNombreSommets();
        Semaphore[][] sem = new Semaphore[nbProcessus][nbProcessus];
        for (int i=0; i<nbProcessus; i++)
            for (int j=0; j<nbProcessus; j++)
                if (graphe.isArcInitial(i,j))
                    sem[i][j] = new Semaphore(1);
                else if (graphe.isArc(i,j))
                    sem[i][j] = new Semaphore(0);
        Random alea = new Random();
        Tache[] tache = new Tache[nbProcessus];
        for (int i=0; i<nbProcessus; i++) {
            tache[i] = new Tache(i,alea.nextInt(200),graphe,sem);
            tache[i].start();
        }
        for (int i=0; i<nbProcessus; i++) {
            try {tache[i].join();}
            catch (InterruptedException e) {}
        }
        System.out.println("\nFIN ");
    }
}
```

Le fichier `Semaphore.java`

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

```
        if (val >= 0)
            n = val;
    }
    synchronized void p() { // autre nom de wait()
        n--;
        if (n < 0)
            try{wait();}
            catch(InterruptedException e){}
    }
    synchronized void v() { // autre nom de signal()
        n++;
        if (n <= 0)
            notify();
    }
}
```

Le fichier `Tache.java`

```
class Tache extends Thread {
    Semaphore[][] sem;
    Graphe graphe;
    int threadId;
    int duree;

    Tache(int id, int duree, Graphe graphe, Semaphore[][] sem) {
        setName(Integer.toString(id));
        threadId = id;
        this.duree = duree;
        this.graphe = graphe;
        this.sem = sem;
    }

    public void run() {
        // chaque tache est exécutée 10 fois
        for (int occurence=0; occurence<10; occurence++){
            // protocole de debut de tâche
            for (int i=0; i<graphe.getNombreSommets(); i++)
                if (graphe.isArc(i,threadId))
                    sem[i][threadId].p();
            // corps de la tâche avec simulation de la durée
            System.out.println("Tâche "+
                               currentThread().getName()+
                               " : début de l'occurence "+
                               occurence);
            try{sleep(duree);}
            catch(InterruptedException e){}
            System.out.println("              ... fin de la tâche "+
                               currentThread().getName());
            // protocole de fin de tâche
            for (int i=0; i<graphe.getNombreSommets(); i++)
                if (graphe.isArc(threadId,i))
                    sem[threadId][i].v();
        }
    }
}
```

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.*

Le fichier `Tache.java`

```
class Tache extends Thread {

    Semaphore[][] sem;
    Graphe graphe;
    int threadId;
    int duree;
    int[] resNecessaires;
    Semaphore[] resDisponibles;

    Tache(int id, int duree, Graphe graphe, Semaphore[][] sem,
          int[] resNecessaires, Semaphore[] resDisponibles) {
        setName(Integer.toString(id));
        threadId = id;
        this.duree = duree;
        this.graphe = graphe;
        this.sem = sem;
        this.resNecessaires = resNecessaires;
        this.resDisponibles = resDisponibles;
    }

    public void run() {
        // chaque tache est exécutée 10 fois
        for (int occurence=0; occurence<10; occurence++){
```

```
            // protocole de debut de tâche

            for (int i=0; i<graphe.getNombreSommets(); i++)
                if (graphe.isArc(i,threadId))
                    sem[i][threadId].p();

            // corps de la tâche avec simulation de la durée
            System.out.println("Tâche "+
                               currentThread().getName()+
                               " : début de l'occurence "+
                               occurence);
            try{sleep(duree);}
            catch(InterruptedException e){}
            System.out.println("              ... fin de la tâche "+
                               currentThread().getName());
            // protocole de fin de tâche

            for (int i=0; i<graphe.getNombreSommets(); i++)
                if (graphe.isArc(threadId,i))
                    sem[threadId][i].v();

        }
    }
}
```

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.*

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.

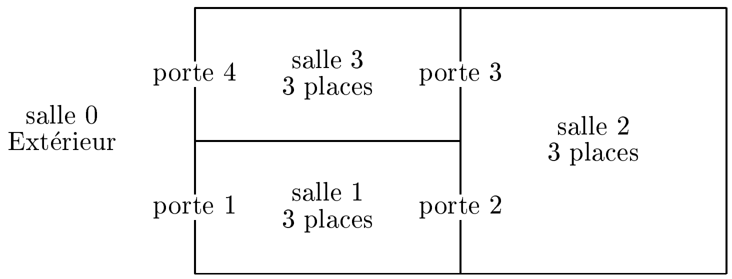

Fig. 1 – Un exemple de musée

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`

```
import java.util.*;

class Visiteur extends Thread {
    private static Random alea = new Random();
    int salle;
    int nbSalles;
    boolean[] visitee;
    boolean fini;

    Visiteur(String who, int nb) {
        setName(who);
        nbSalles = nb;           // l'exterieur est codé comme une salle
        salle = 0;               // le visiteur est initialement à l'extérieur
        visitee = new boolean[nbSalles];
        visitee[0] = true;       // l'extérieur est visité
        for (int i=1; i<nbSalles; i++)
            visitee[i] = false;  // aucune "vraie" salle n'est visitée
    }

    public void bouger() {
        // il va au hasard dans une des deux salles voisines.
        if (alea.nextBoolean())
            salle = (salle+1)%nbSalles;
        else
            salle = (salle-1+nbSalles)%nbSalles;
        visitee[salle] = true;
```

```
            System.out.println(getName()+" dans la salle "+salle);
            // il dort pendant les commentaires du guide
            try {Thread.sleep(alea.nextInt(1000));}
            catch (InterruptedException ie){}
            // il a fini s'il a vu toutes les salles et si il est sorti du musée
            if (salle == 0){
                fini = true;
                for (int i=0; i<nbSalles; i++)
                    fini &= visitee[i];
            }
            else
                fini = false;
        }

        public void run() {
            while (!fini)
                bouger();
        }
    }
```

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.*

Le fichier `Musee.java`

```
import java.util.*;

class Musee {
    public static void usage () {
        System.err.println("syntaxe : java Musee NbSalles NbVisiteurs");
    }

    public static void main(String []args) {
        boolean pasArgument = args.length != 2;
        if (pasArgument)
            usage();
        else {
            System.out.println("DEBUT");
            int nbSalles = Integer.parseInt(args[0]);
            int nbVisiteurs = Integer.parseInt(args[1]);
            Visiteur[] visiteur = new Visiteur[nbVisiteurs];
            for (int i=0; i<nbVisiteurs; i++)
                visiteur[i] = new Visiteur("Visiteur "+i+" ",nbSalles);
            for (int i=0; i<nbVisiteurs; i++)
                visiteur[i].start();







            System.out.println("FIN");
        }
    }
}
```

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.*

Le fichier `Controleur.java`

```java
import java.util.*;

class Controleur {
    int[] nbPlaces, nbVisiteurs;

    public Controleur (int[] capacite) {
        nbPlaces = capacite;
        nbVisiteurs = new int[capacite.length];
        nbVisiteurs[0] = nbPlaces[0];
        for (int i=1; i<capacite.length; i++)
            nbVisiteurs[i] = 0;
    }

    public synchronized void entrer(int salle){

                System.out.println(Thread.currentThread().getName()+ "attend");

    }

    public synchronized void sortir(int salle){
        // il est toujours possible de quitter une salle.

    }

}
```

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 ?*

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.*

Le fichier `Controleur.java`

```
import java.util.*;

class Controleur {
    int[] nbPlaces, nbVisiteurs;

    public Controleur (int[] capacite) {
        nbPlaces = capacite;
        nbVisiteurs = new int[capacite.length];
        nbVisiteurs[0] = nbPlaces[0];
        for (int i=1; i<capacite.length; i++)
            nbVisiteurs[i] = 0;
    }

    public synchronized void deplacer(int from, int to){

                System.out.println(Thread.currentThread().getName()+ "attend");

    }
}
```

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