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

IUP MIAGe 3^{eme} année
Année 2002-2003

Mardi 29 avril, 14 heures - 15 heures 30
Tous documents autorisés.

Programmation Parallèle
Alain Griffault

Numéro d'anonymat :

Partage de ressources (20 points)

L'objectif de l'exercice est de construire un ordonnanceur non bloquant de processus partageant plusieurs ressources communes.
Remarque : le nombre de lignes laissées vides dans les sources à complétés est $\geq$ au nombre nécessaire.

1 La première version

Les sources suivantes décrivent un contrôleur qui démarre plusieurs `Thread` processus. Chacun d'eux simule son exécution en effectuant différents types d'instructions (calcul, écriture, lecture au clavier).
Le fichier `Controle.java`

```
import java.util.*;

public  class Controle {

    Controle() {
        Random r = new Random();
        for (int i=0; i<r.nextInt(10); i++) { // nb aleatoire de processus
            Processus p = new Processus();
            p.start();
        }
    }

    public static void main(String []args) {
        new Controle();
    }
}
```

Le fichier `Processus.java`

```
import java.io.*;
import java.util.*;

public class Processus extends Thread {

    static int numero = 0;
    static Random r = new Random();
    int threadIdent;
    BufferedReader clavier;
```

```java

    public Processus() {
        threadIdent = numero++;
        clavier = new BufferedReader(new InputStreamReader(System.in));
    }

    public void run() {
        try {
            for(int i=0; i<r.nextInt(100); i++) // nb aleatoire d'instructions
                instruction();
        }
        catch(Exception e){}
        finally{}
    }

    private void instruction() throws Exception {
        switch(r.nextInt(3)){ // choix aleatoire d'une instruction
        case 0 : calcul(); break;
        case 1 : entreeClavier(); break;
        case 2 : ecritureCache(); break;
        default :
        }
    }

    private void calcul() throws Exception {
        Thread.sleep(10); // simule le chargement contexte + executions
    }

    private void entreeClavier() throws Exception {
        Thread.sleep(10); // simule le chargement contexe + interruption I/O
        System.out.print("? "+threadIdent+" : ");System.out.flush();
        String ligne = clavier.readLine();
    }

    private void ecritureCache() throws Exception {
        Thread.sleep(10); // simule le chargement contexte + executions
        Thread.sleep(50); // simule la sauvegarde en memoire cache
    }

}
```

Voici la trace d'une exécution possible :

```
[alain@localhost share]$ java Controle
? 1 : ? 2 : ? 0 : a
? 1 : b
c
d
? 2 : e
? 2 : f
? 2 : g
[alain@localhost share]$
```

Question 1.1 (3 points) *Expliquez l'exécution précédente.*

2 La seconde version

Les processus partagent des ressources disponibles en quantités limitées. Ils ne peuvent donc pas toujours faire ce qu'ils veulent. Le contrôleur va donc contraindre le comportement des processus à l'aide de sémaphores, dont voici l'implémentation classique.

Le fichier `Semaphore.java`

```java
class Semaphore {
    private int n = 1;
    Semaphore(int val){
        if (val >= 0)
            n = val;
    }

    synchronized void p() throws InterruptedException { // autre nom de wait()
        n--;
        if (n < 0)
            wait();
    }

    synchronized void v() { // autre nom de siganl()
        n++;
        notify();
    }
}
```

Les contraintes imposées au contrôleur sont les suivantes :

- les processus partagent le même processeur,
- les processus partagent le même clavier,
- les processus partagent les mêmes trois pages de cache mémoire,
- une instruction `calcul` nécessite le processeur,
- une instruction `entreeClavier` nécessite le processeur pour traiter l'instruction, **puis** le clavier,
- une instruction `ecritureCache` nécessite le processeur pour traiter l'instruction, **ainsi qu'une** page pendant la sauvegarde.

Question 2.1 (5 points) *Complétez le code des deux classes* `Controle` *et* `Processus` *afin que les spécifications précédentes soient respectées.*

Le fichier `Controle.java`

```java
import java.util.*;

public  class Controle {

    Semaphore cpu, keyboard, page;

    Controle() {
        Random r = new Random();





        for (int i=0; i<r.nextInt(10); i++) { // nb aleatoire de processus
            Processus p = new Processus(cpu, keyboard, page);
            p.start();
        }
    }

    public static void main(String []args) {
        new Controle();
    }
}
```

Le fichier `Processus.java`

```java
import java.io.*;
import java.util.*;

public class Processus extends Thread {

    static int numero = 0;
    static Random r = new Random();
    int threadIdent;
    BufferedReader clavier;

    Semaphore cpu, keyboard, page;

    public Processus(Semaphore cpu,
                     Semaphore kb,
                     Semaphore pg) {
        threadIdent = numero++;
        clavier = new BufferedReader(new InputStreamReader(System.in));
        this.cpu = cpu;
        keyboard = kb;
        page = pg;
    }

    public void run() {
        try {
```

```
            for(int i=0; i<r.nextInt(100); i++) // nb aleatoire d'instructions
                instruction();
        }
        catch(Exception e){}
        finally{}
    }

    private void instruction() throws Exception {
        switch(r.nextInt(3)){ // choix aleatoire d'une instruction
        case 0 : calcul(); break;
        case 1 : entreeClavier(); break;
        case 2 : ecritureCache(); break;
        default :
        }
    }

    private void calcul() throws Exception {

        Thread.sleep(10); // simule le chargement contexte + executions

    }

    private void entreeClavier() throws Exception {

        Thread.sleep(10); // simule le chargement contexe + interruption I/O

        System.out.print("? "+threadIdent+" : ");System.out.flush();
        String ligne = clavier.readLine();

    }

    private void ecritureCache() throws Exception {

        Thread.sleep(10); // simule le chargement contexte + executions

        Thread.sleep(50); // simule la sauvegarde en memoire cache

    }
}
```

Question 2.2 (3 points) *Votre solution fonctionne t-elle pour un nombre de processeurs supérieur à 1 ? Si oui, justifiez votre réponse ; si non, donnez une exécution possible mettant en évidence un comportement non désiré.*

3 La troisième version

Les contraintes du contrôleur sont modifiées (en gras dans le texte) pour devenir :
- les processus partagent les mêmes **trois** processeurs,
- les processus partagent le même clavier,
- les processus partagent les mêmes trois pages de cache mémoire,
- une instruction `calcul` nécessite le processeur,
- une instruction `entreeClavier` nécessite le processeur pour traiter l'instruction, puis le clavier,
- une instruction `ecritureCache` nécessite le processeur pour traiter l'instruction, ainsi **qu'un nombre aléatoire <4 de pages** pendant la sauvegarde.

Question 3.1 (5 points) *Complétez le code des deux classes* `Controle` *et* `Processus` *afin que les spécifications précédentes soient respectées.*

Le fichier `Controle.java`

```
import java.util.*;

public  class Controle {

    Semaphore cpu, keyboard, page;

    Controle() {
        Random r = new Random();





        for (int i=0; i<r.nextInt(10); i++) { // nb aleatoire de processus
            Processus p = new Processus(cpu, keyboard, page);
            p.start();
        }
    }

```

```
    public static void main(String []args) {
        new Controle();
    }
}
```

Le fichier `Processus.java`

```
import java.io.*;
import java.util.*;

public class Processus extends Thread {

    static int numero = 0;
    static Random r = new Random();
    int threadIdent;
    BufferedReader clavier;

    Semaphore cpu, keyboard, page;

    public Processus(Semaphore cpu,
                     Semaphore kb,
                     Semaphore pg) {
        threadIdent = numero++;
        clavier = new BufferedReader(new InputStreamReader(System.in));
        this.cpu = cpu;
        keyboard = kb;
        page = pg;
    }

    public void run() {
        try {
            for(int i=0; i<r.nextInt(100); i++) // nb aleatoire d'instructions
                instruction();
        }
        catch(Exception e){}
        finally{}
    }

    private void instruction() throws Exception {
        switch(r.nextInt(3)){ // choix aleatoire d'une instruction
        case 0 : calcul(); break;
        case 1 : entreeClavier(); break;
        case 2 : ecritureCache(); break;
        default :
        }
    }

    private void calcul() throws Exception {


        Thread.sleep(10); // simule le chargement contexte + executions


    }
```

```java
    private void entreeClavier() throws Exception {
        Thread.sleep(10); // simule le chargement contexe + interruption I/O
        System.out.print("? "+threadIdent+" : ");System.out.flush();
        String ligne = clavier.readLine();
    }

    private void ecritureCache() throws Exception {
        Thread.sleep(10); // simule le chargement contexte + executions
        Thread.sleep(50); // simule la sauvegarde en memoire cache
    }
}
```

Question 3.2 (4 points) *Votre solution est-elle exempte d'inter-blocage ? Si oui, justifiez votre réponse ; si non, donnez d'une part une exécution possible mettant en évidence un comportement non désiré ; d'autre part une idée de correction.*