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

SUJET + CORRIGE

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`

```
1   import java.util.*;
2
3   public  class Controle {
4
5       Controle() {
6           Random r = new Random();
7           for (int i=0; i<r.nextInt(10); i++) { // nb aleatoire de processus
8               Processus p = new Processus();
9               p.start();
10          }
11      }
12
13      public static void main(String []args) {
14          new Controle();
15      }
16  }
```

Le fichier `Processus.java`

```
1   import java.io.*;
2   import java.util.*;
3
4   public class Processus extends Thread {
5
6       static int numero = 0;
7       static Random r = new Random();
8       int threadIdent;
9       BufferedReader clavier;
```

```
10
11      public Processus() {
12          threadIdent = numero++;
13          clavier = new BufferedReader(new InputStreamReader(System.in));
14      }
15
16      public void run() {
17          try {
18              for(int i=0; i<r.nextInt(100); i++) // nb aleatoire d'instructions
19                  instruction();
20          }
21          catch(Exception e){}
22          finally{}
23      }
24
25      private void instruction() throws Exception {
26          switch(r.nextInt(3)){ // choix aleatoire d'une instruction
27          case 0 : calcul(); break;
28          case 1 : entreeClavier(); break;
29          case 2 : ecritureCache(); break;
30          default :
31          }
32      }
33
34      private void calcul() throws Exception {
35          Thread.sleep(10); // simule le chargement contexte + executions
36      }
37
38      private void entreeClavier() throws Exception {
39          Thread.sleep(10); // simule le chargement contexe + interruption I/O
40          System.out.print("? "+threadIdent+" : ");System.out.flush();
41          String ligne = clavier.readLine();
42      }
43
44      private void ecritureCache() throws Exception {
45          Thread.sleep(10); // simule le chargement contexte + executions
46          Thread.sleep(50); // simule la sauvegarde en memoire cache
47      }
48
49  }
```

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.*

Réponse à la question 1.1: La première ligne indique que les processus `1, 2` et `0` sont en attente d'une entrée au clavier. Vu l'ordre d'activation des processus, cela indique les le processus `0` a certainement effectuer plus d'instructions `calcul()` ou `ecritureCache()` que les deux autres.
L'entrée `a` a libéré le processus `1` qui aussitôt est repassé en attente.
Il est très difficile de savoir qui va lire l'entrée `b` de la seconde ligne.
Au total, `0` a effectué une instruction `entreeClavier()`, `1` en a effectué deux et `2` quatre.

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`

```
1   class Semaphore {
2       private int n = 1;
3       Semaphore(int val){
4           if (val >= 0)
5               n = val;
6       }
7
8       synchronized void p() throws InterruptedException { // autre nom de wait()
9           n--;
10          if (n < 0)
11              wait();
12      }
13
14      synchronized void v() { // autre nom de siganl()
15          n++;
16          notify();
17      }
18   }
```

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`

```
1  import java.util.*;
2
3  public  class Controle {
4
5      Semaphore cpu, keyboard, page;
6
7      Controle() {
8          Random r = new Random();
9          cpu = new Semaphore(1);
10         keyboard = new Semaphore(1);
11         page = new Semaphore(3);
12         for (int i=0; i<r.nextInt(10); i++) { // nb aleatoire de processus
13             Processus p = new Processus(cpu, keyboard, page);
14             p.start();
15         }
16     }
17
18     public static void main(String []args) {
19         new Controle();
20     }
21 }
```

Le fichier `Processus.java`

```
1  import java.io.*;
2  import java.util.*;
3
4  public class Processus extends Thread {
5
6      static int numero = 0;
7      static Random r = new Random();
8      int threadIdent;
9      BufferedReader clavier;
10
11     Semaphore cpu, keyboard, page;
12
13     public Processus(Semaphore cpu,
14                      Semaphore kb,
15                      Semaphore pg) {
16         threadIdent = numero++;
17         clavier = new BufferedReader(new InputStreamReader(System.in));
18         this.cpu = cpu;
19         keyboard = kb;
20         page = pg;
21     }
22
23     public void run() {
24         try {
25             for(int i=0; i<r.nextInt(100); i++) // nb aleatoire d'instructions
26                 instruction();
```

```
27          }
28          catch(Exception e){}
29          finally{}
30      }
31
32      private void instruction() throws Exception {
33          switch(r.nextInt(3)){ // choix aleatoire d'une instruction
34          case 0 : calcul(); break;
35          case 1 : entreeClavier(); break;
36          case 2 : ecritureCache(); break;
37          default :
38          }
39      }
40
41      private void calcul() throws Exception {
42          cpu.p();
43          Thread.sleep(10); // simule le chargement contexte + executions
44          cpu.v();
45      }
46
47      private void entreeClavier() throws Exception {
48          cpu.p();
49          Thread.sleep(10); // simule le chargement contexe + interruption I/O
50          cpu.v();
51          keyboard.p();
52          System.out.print("? "+threadIdent+" : ");System.out.flush();
53          String ligne = clavier.readLine();
54          keyboard.v();
55      }
56
57      private void ecritureCache() throws Exception {
58          cpu.p();
59          Thread.sleep(10); // simule le chargement contexte + executions
60          page.p();
61          Thread.sleep(50); // simule la sauvegarde en memoire cache
62          page.v();
63          cpu.v();
64      }
65
66  }
```

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.*

Réponse à la question 2.2: Oui, cette solution fonctionne pour un nombre quelconque de processeurs.
Pour les instructions `calcul()`, cela permet d'en effectuer plusieurs simultanément.
Pour les instructions `entreeClavier()`, cela ne change rien, car les `cpu` sont libérées avant l'acquisition du `clavier`.
Pour les instructions `ecritureCache()`, cela permet d'en effectuer plusieurs simultanément sous réserve d'un nombre suffisant de pages disponibles. Sur notre exemple trois processus peuvent donc écrire simultanément. Un quatrième devra attendre la libération d'une page avant de pouvoir écrire.

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`

```
1   import java.util.*;
2
3   public  class Controle {
4
5       Semaphore cpu, keyboard, page;
6
7       Controle() {
8           Random r = new Random();
9           cpu = new Semaphore(3);
10          keyboard = new Semaphore(1);
11          page = new Semaphore(3);
12          for (int i=0; i<r.nextInt(10); i++) { // nb aleatoire de processus
13              Processus p = new Processus(cpu, keyboard, page);
14              p.start();
15          }
16      }
17
18      public static void main(String []args) {
19          new Controle();
20      }
21  }
```

Le fichier `Processus.java`

```
1   import java.io.*;
2   import java.util.*;
3
4   public class Processus extends Thread {
5
6       static int numero = 0;
7       static Random r = new Random();
8       int threadIdent;
9       BufferedReader clavier;
10
11      Semaphore cpu, keyboard, page;
12
13      public Processus(Semaphore cpu,
14                       Semaphore kb,
15                       Semaphore pg) {
```

```
16          threadIdent = numero++;
17          clavier = new BufferedReader(new InputStreamReader(System.in));
18          this.cpu = cpu;
19          keyboard = kb;
20          page = pg;
21      }
22
23      public void run() {
24          try {
25              for(int i=0; i<r.nextInt(100); i++) // nb aleatoire d'instructions
26                  instruction();
27          }
28          catch(Exception e){}
29          finally{}
30      }
31
32      private void instruction() throws Exception {
33          switch(r.nextInt(3)){ // choix aleatoire d'une instruction
34          case 0 : calcul(); break;
35          case 1 : entreeClavier(); break;
36          case 2 : ecritureCache(); break;
37          default :
38          }
39      }
40
41      private void calcul() throws Exception {
42          cpu.p();
43          Thread.sleep(10); // simule le chargement contexte + executions
44          cpu.v();
45      }
46
47      private void entreeClavier() throws Exception {
48          cpu.p();
49          Thread.sleep(10); // simule le chargement contexe + interruption I/O
50          cpu.v();
51          keyboard.p();
52          System.out.print("? "+threadIdent+" : ");System.out.flush();
53          String ligne = clavier.readLine();
54          keyboard.v();
55      }
56
57      private void ecritureCache() throws Exception {
58          cpu.p();
59          Thread.sleep(10); // simule le chargement contexte + executions
60          int nbFD = r.nextInt(4);
61          for (int i=0; i<nbFD;i++)
62              page.p();
63          Thread.sleep(50); // simule la sauvegarde en memoire cache
64          for (int i=0; i<nbFD;i++)
65              page.v();
66          cpu.v();
67      }
68
```

```
69   }
```

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.*

Réponse à la question 3.2: Non, il y a un risque d'inter-blocage.
L'exécution suivante est en effet possible.

```
[alain@localhost share]$ java Controle
? 0 :
? 3 :
? 4 :
? 1 :
? 2 :
? 5 :
? 0 :
? 3 :
? 2 :
? 0 :
? 3 :
```

Sur cet exemple, il suffit qu'à un instant donné, un premier processus souhaite obtenir trois pages, un second souhaite également trois pages. Le premier en obtient deux, le second une, puis les deux se bloquent en attente d'une libération de page qui ne pourra jamais se produire.
La correction de ce problème est compliquée, plusieurs solutions :
- Modifier la classe sémaphore pour pouvoir retirer en une seule fois plusieurs ressources à condition qu'il y en ai suffisamment.
- Ordonner les pages (les différencier) et les requérir toujours dans le même ordre.