Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Programmation Concurrente

Introduction
Exclusion mutuelle

Jacques Supcik | 2013 | T-2a/T2d

Hes·so FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Moodle/Cyberlearn

`http://cyberlearn.hes-so.ch/course/view.php?id=3830`

1213_HES-SO-FR_IT2_Programmation concurrente (ProgConc)

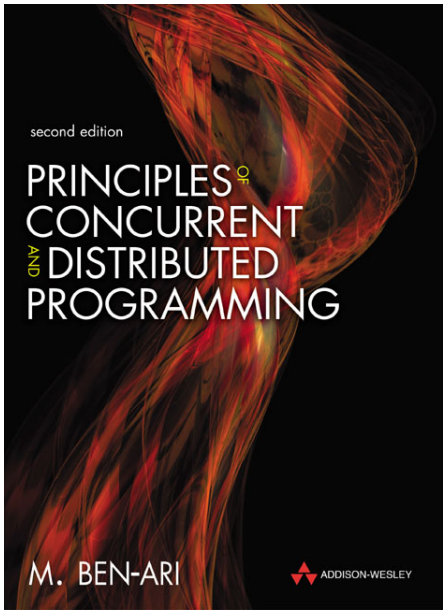
Enrolment Key: `Dijkstra4ever`

Motivation

- En 2013, la fréquence du «clock» ne peut plus beaucoup augmenter (problème de dissipation de la chaleur)
- Performance des algorithmes («web spider», opérations sur les matrices, «graphic rendering», …)

Littérature

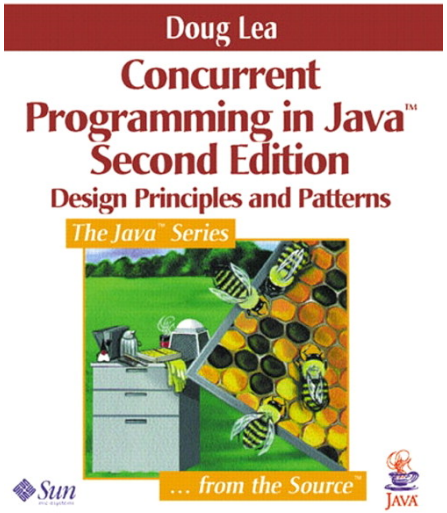
Principles of Concurrent And Distributed Programming – M. Ben-Ari



Edition	2, illustrated
Publisher	Pearson Education, 2006
ISBN	032131283X, 9780321312839
Length	361 pages

Littérature

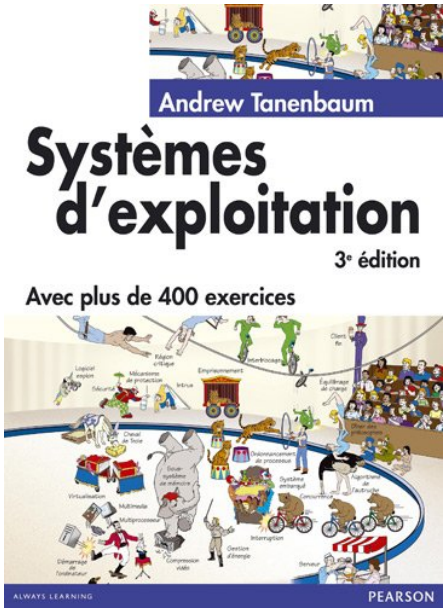
Concurrent Programming in Java: Design Principles and Pattern – Doug Lea



Edition	2, illustrated, reprint
Publisher	Addison-Wesley Professional, 2000
ISBN	0201310090, 9780201310092
Length	411 pages

Littérature

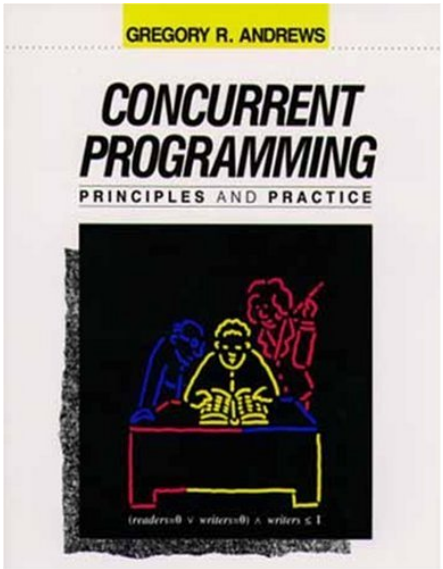
Systèmes d'exploitation 3ème Ed. – Andrew Tanenbaum



Edition	3, illustrated
Publisher	Pearson Education, 2008
ISBN	2744072990, 978-2744072994
Length	1070 pages

Littérature

Concurrent programming: principles and practice – Gregory R. Andrews



Publisher Benjamin/Cummings Pub. Co., 1991

ISBN 0805300864, 9780805300864

Length 637 pages

Définitions

- Un *programme concurrent* est composé de plusieurs processus (threads) qui *coopèrent* pour effectuer une tâche donnée.
- Un *processus* (thread) est un programme qui exécute une séquence d'instructions.

Définitions

- Une *action atomique* est une action qui ne peut pas être interrompue. En java, les opérations suivantes sont atomiques:
  - l'assignation de types primitifs (sauf les `long` et `double`).
  - L'assignation de références.
  - Les opérations de `java.concurrent.Atomic.*`
- L'*entrelacement* (interleaving) est l'exécution d'une séquence d'actions atomiques indépendamment des processus.

Définitions

- une *section critique* (ou *région critique*) est une partie d'un programme à partir de laquelle on accède à une ressource partagée.
- On empêche que plusieurs processus se trouvent simultanément dans la section critique à l'aide de techniques d'*exclusion mutuelle*.

Exclusion mutuelle – Première tentative

Programme principal

```
 1  package attempt1;
 2
 3  public class Test {
 4
 5    static int turn = 1;
 6
 7    public static void main(String[] args)
 8          throws InterruptedException {
 9      Thread p1 = new Process1();
10      Thread p2 = new Process2();
11      p1.start(); p2.start();
12      p1.join();  p2.join();
13      System.out.println(”Done”);
14    }
15  }
```

Exclusion mutuelle – Première tentative

Classe «Process1»

```
1  package attempt1;
2
3  public class Process1 extends Thread {
4
5    public void run() {
6      for (int i = 0; i <= 10; i++) {
7        // Entry protocol
8        while (Test.turn != 1) Thread.yield();
9        // Critical Section
10       System.out.printf("Proc␣1␣in␣crit.␣region␣(i=%d)\n", i);
11       try {
12         Thread.sleep((long) (100 + (Math.random() * 900)));
13       } catch (InterruptedException e) {
14         e.printStackTrace();
15       }
16       // Exit protocol
17       Test.turn = 2;
18     }
19   }
20 }
```

Exclusion mutuelle – Première tentative

Classe «Process2»

```
1  package attempt1;
2
3  public class Process2 extends Thread {
4
5    public void run() {
6      for (int i = 0; i <= 10; i++) {
7        // Entry protocol
8        while (Test.turn != 2) Thread.yield();
9        // Critical Section
10       System.out.printf("Proc␣2␣in␣crit.␣region␣(i=%d)\n", i);
11       try {
12         Thread.sleep((long) (100 + (Math.random() * 900)));
13       } catch (InterruptedException e) {
14         e.printStackTrace();
15       }
16       // Exit protocol
17       Test.turn = 1;
18     }
19   }
20 }
```

Exclusion mutuelle – Première tentative

Problème

Problèmes:

- Alternance stricte entre le processus 1 et le processus 2.
- Si un processus termine (volontairement ou à la suite d'un «bug»), alors l'autre processus n'a aucune chance de continuer.

Exclusion mutuelle – Deuxième tentative

Programme principal

```
 1  package attempt2;
 2
 3  public class Test {
 4
 5    static boolean inC1 = false;
 6    static boolean inC2 = false;
 7
 8    public static void main(String[] args)
 9        throws InterruptedException {
10      Thread p1 = new Process1();
11      Thread p2 = new Process2();
12      p1.start(); p2.start();
13      p1.join();  p2.join();
14      System.out.println(”Done”);
15    }
16  }
```

Exclusion mutuelle – Deuxième tentative

Classes «Process1» et «Process2»

```
1  class Process1 {
2
3    void run() {
4      for (int i = ...) {
5        // Entry protocol
6        while (Test.inC2)
7          yield();
8        Test.inC1 = true;
9        // Critical Section
10       printf("P1 in CR");
11       try {
12         Thread.sleep(...);
13       } catch ...
14       // Exit protocol
15       Test.inC1 = false;
16     }
17   }
18 }
```

```
class Process2 {

  void run() {
    for (int i = ...) {
      // Entry protocol
      while (Test.inC1)
        yield();
      Test.inC2 = true;
      // Critical Section
      printf("P2 in CR");
      try {
        Thread.sleep(...);
      } catch ...
      // Exit protocol
      Test.inC2 = false;
    }
  }
}
```

Exclusion mutuelle – Deuxième tentative

Problème

Problème:

- L'exclusion mutuelle n'est pas garantie! Cette solution n'est pas utilisable.

Exclusion mutuelle – Troisième tentative

Classes «Process1» et «Process2» (le programme principal reste inchangé)

```
1  class Process1 {
2
3    void run() {
4      for (int i = ...) {
5        // Entry protocol
6        Test.inC1 = true;
7        while (Test.inC2)
8          yield();
9        // Critical Section
10       printf(”P1␣in␣CR”);
11       try {
12         Thread.sleep(...);
13       } catch ...
14       // Exit protocol
15       Test.inC1 = false;
16     }
17   }
18 }
```

```
class Process2 {

  void run() {
    for (int i = ...) {
      // Entry protocol
      Test.inC2 = true;
      while (Test.inC1)
        yield();
      // Critical Section
      printf(”P2␣in␣CR”);
      try {
        Thread.sleep(...);
      } catch ...
      // Exit protocol
      Test.inC2 = false;
    }
  }
}
```

Exclusion mutuelle – Troisième tentative

Problème

Problème:

- Risque de «deadlock» $\rightarrow$ Inutilisable!

Exclusion mutuelle – Quatrième tentative

Classes «Process1» et «Process2»

```
1  class Process1 {
2    void run() {
3      for (int i = ...) {
4        // Entry protocol
5        Test.inC1 = true;
6        while (Test.inC2) {
7          Test.inC1 = false;
8          Thread.yield();
9          Test.inC1 = true;
10       }
11       // Critical Section
12       printf("P1 in CR");
13       try {
14         Thread.sleep(...);
15       } catch ...
16       // Exit protocol
17       Test.inC1 = false;
18     }
19   }
20 }
```

```
class Process2 {
  void run() {
    for (int i = ...) {
      // Entry protocol
      Test.inC2 = true;
      while (Test.inC1) {
        Test.inC2 = false;
        Thread.yield();
        Test.inC2 = true;
      }
      // Critical Section
      printf("P2 in CR");
      try {
        Thread.sleep(...);
      } catch ...
      // Exit protocol
      Test.inC2 = false;
    }
  }
}
```

Exclusion mutuelle – Quatrième tentative

Problème

Problème:

- Risque de «livelock» (mais très peu probable que le blocage dure pour toujours)

Exclusion mutuelle – Algorithme de Dekker

Programme principal

```
1  package attempt5; // Dekker's algorithm
2
3  public class Test {
4
5    static boolean enter1 = false;
6    static boolean enter2 = false;
7    static int turn = 1;
8
9     public static void main(String[] args)
10        throws InterruptedException {
11      Thread p1 = new Process1();
12      Thread p2 = new Process2();
13     p1.start(); p2.start();
14     p1.join();  p2.join();
15     System.out.println("Done");
16   }
17 }
```

Exclusion mutuelle – Algorithme de Dekker

Classes «Process1» et «Process2»

```
1  class Process1 {
2    void run() {
3      for (int i = ...) {
4        // Entry protocol
5        Test.enter1 = true;
6        while (Test.enter2) {
7          if (Test.turn =! 1) {
8            Test.enter1 = false;
9            while (test.turn == 2)
10             Thread.yield();
11           Test.enter1 = true;
12       }
13       // Critical Section
14       printf(”P1␣in␣CR”);
15       try {
16         Thread.sleep(...);
17       } catch ...
18       // Exit protocol
19       Test.turn = 2;
20       Test.enter1 = false;
21     }
22   }
23 }
```

```
class Process2 {
  void run() {
    for (int i = ...) {
      // Entry protocol
      Test.enter2 = true;
      while (Test.enter1) {
        if (Test.turn =! 2) {
          Test.enter2 = false;
          while (test.turn == 1)
            Thread.yield();
          Test.enter2 = true;
      }
      // Critical Section
      printf(”P2␣in␣CR”);
      try {
        Thread.sleep(...);
      } catch ...
      // Exit protocol
      Test.turn = 1;
      Test.enter2 = false;
    }
  }
}
```

Exclusion mutuelle - Algorithme de Peterson

Programme principal

```
1  package attempt6; // Peterson's algorithm
2
3  public class Test {
4
5    static boolean enter1 = false;
6    static boolean enter2 = false;
7    static int turn = 0;
8
9    public static void main(String[] args)
10       throws InterruptedException {
11     Thread p1 = new Process1();
12     Thread p2 = new Process2();
13     p1.start(); p2.start();
14     p1.join();  p2.join();
15     System.out.println("Done");
16   }
17 }
```

Exclusion mutuelle - Algorithme de Peterson

Classe «Process1» et «Process2»

```
1  class Process1 {
2    void run() {
3      for (int i = ...) {
4        // Entry protocol
5        Test.enter1 = true;
6        Test.turn = 2;
7        while (Test.enter2
8            && Test.turn == 2) {
9          Thread.yield();
10       }
11       // Critical Section
12       printf("P1 in CR");
13       try {
14         Thread.sleep(...);
15       } catch ...
16       // Exit protocol
17       Test.enter1 = false;
18     }
19   }
20 }
```

```
class Process2 {
  void run() {
    for (int i = ...) {
      // Entry protocol
      Test.enter2 = true;
      Test.turn = 1;
      while (Test.enter1
          && Test.turn == 1) {
        Thread.yield();
      }
      // Critical Section
      printf("P2 in CR");
      try {
        Thread.sleep(...);
      } catch ...
      // Exit protocol
      Test.enter2 = false;
    }
  }
}
```