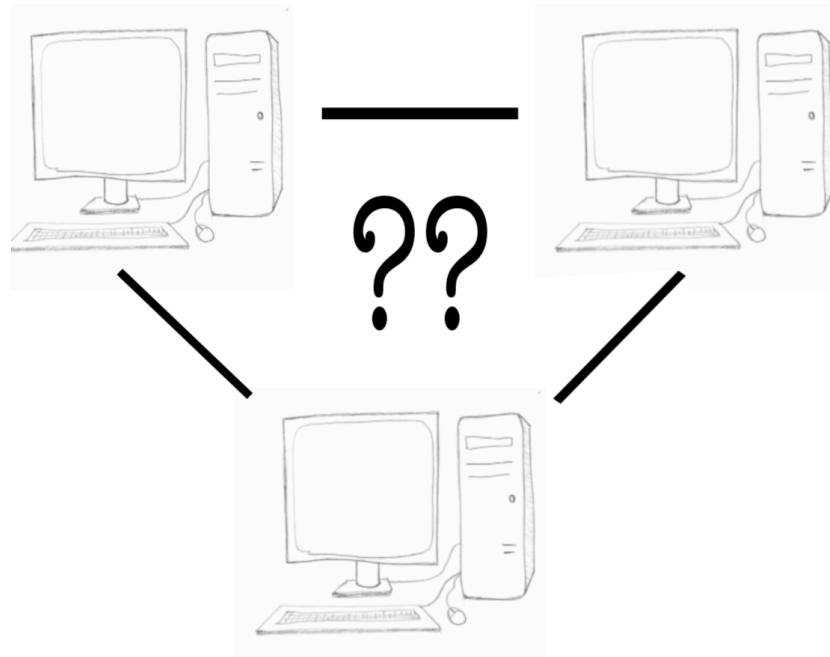


PROGRAMMATION RÉSEAU

Arnaud Sangnier
sangnier@irif.fr

La concurrence en Java - II



Les problèmes de la concurrence (1)

```
import java.net.*;
import java.io.*;
import java.lang.*;

public class Compteur{

    private int valeur;

    public Compteur(){
        valeur=0;
    }

    public int getValeur(){
        return valeur;
    }

    public void setValeur(int v){
        valeur=v;
    }

}
```

Les problèmes de la concurrence (2)

```
import java.net.*;
import java.io.*;
import java.lang.*;

public class CodeCompteur implements Runnable{

    private Compteur c;

    public CodeCompteur(Compteur _c){
        this.c=_c;
    }

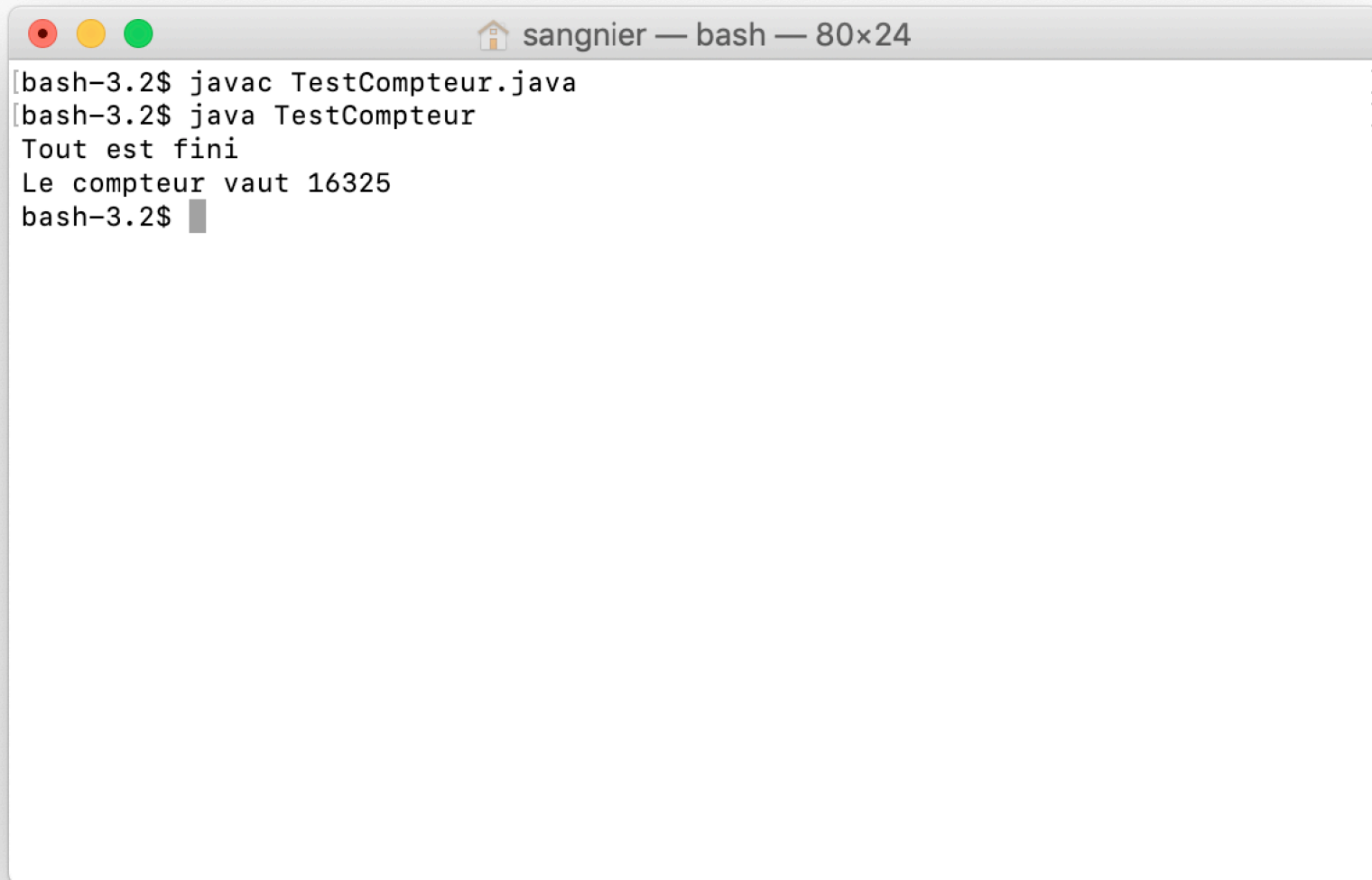
    public void run(){
        for(int i=0; i<1000; i++){
            c.setValeur(c.getValeur()+1);
        }
    }
}
```

Les problèmes de la concurrence (3)

```
import java.lang.*;
import java.io.*;

public class TestCompteur {
    public static void main(String[] args){
        try{
            Compteur c=new Compteur();
            CodeCompteur code=new CodeCompteur(c);
            Thread []t=new Thread[20];
            for(int i=0; i<20; i++){
                t[i]=new Thread(code);
            }
            for(int i=0; i<20; i++){
                t[i].start();
            }
            for(int i=0; i<20; i++){
                t[i].join();
            }
            System.out.println("Tout est fini");
            System.out.println("Le compteur vaut "+c.getValeur());
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Résultat de l'exécution

A screenshot of a macOS terminal window. The title bar shows a home icon, the text 'sangnier — bash — 80x24', and three window control buttons (red, yellow, green). The terminal content shows the compilation and execution of a Java program. The prompt is '[bash-3.2\$]'. The first command is 'javac TestCompteur.java' followed by a closing bracket. The second command is 'java TestCompteur' followed by a closing bracket. The output consists of two lines: 'Tout est fini' and 'Le compteur vaut 16325'. The prompt returns to '[bash-3.2\$]' with a cursor.

```
[bash-3.2$ javac TestCompteur.java ]
[bash-3.2$ java TestCompteur ]
Tout est fini
Le compteur vaut 16325
bash-3.2$ █
```

16325 n'est pas égal à 20000 !!!!!

D'où vient le problème

- Le problème est la non atomicité de l'opération `c.setValeur(c.getValeur()+1);`
 - C'est à dire que plusieurs threads (qui partagent le même compteur) peuvent faire cette opération en même temps
- Scénario possible
 - Thread 1 prend la valeur du compteur (par exemple 0)
 - Thread 2 prend la valeur du compteur (toujours 0)
 - Thread 1 met la valeur du compteur à jour (à 1)
 - Thread 2 met la valeur du compteur à jour (à 1)
 - À ce point les deux threads ont incrémenté le compteur mais ils ne se sont pas mis d'accord pour le faire
- On remarque aussi qu'on a pas le même résultat à chaque exécution

Comment y remédier

- Principe en programmation concurrente
 - On ne peut faire aucune supposition sur l'ordonnancement des exécutions
 - Tout ordre des instructions des processeurs est possible
- Il faut donc prendre des précautions
 - Par exemple s'assurer que lorsque l'on exécute une instruction, il n'y a pas d'autres threads qui exécute la même instruction
 - On rend la suite d'instructions **atomique**
 - On parle alors de partie du code en **section critique**
 - Dans cette section il n'y a à tout moment qu'au plus un thread
 - Si un thread est présent les autres qui veulent accéder à cette partie du code doivent attendre leur tour

Liens avec les flux et le réseau



Mais comment fait-on
les sections critiques ?

- Pour faire assurer qu'un seul processus accède à une partie du code (exclusion mutuelle), on utilise un système de verrou
- Le processus qui rentre dans le code ferme le verrou et il le rouvre quand il sort du code pour qu'un autre puisse le fermer de nouveau
- En java, on a le mot clef **synchronized**

Fonctionnement de synchronized

- Le mot clef **synchronized** est utilisé dans le code pour garantir qu'à certains endroits et à un moment donné au plus un processus exécute la portion du code
- Deux utilisations
 - Soit on déclare une méthode **synchronized**
 - **public synchronized int f(int a){...}**
 - À ce moment la méthode **synchronized** d'un même objet ne peut pas être exécuté par deux threads en même temps
 - Soit on verrouille un bloc de code en utilisant
 - **synchronized(objet) {.../*code à verrouiller*/...}**
 - ici objet est donné car on utilise le verrou associé à cet objet
 - tout objet possède un verrou

Retour sur notre exemple

```
import java.net.*;
import java.io.*;
import java.lang.*;

public class CodeCompteur implements Runnable{

    private Compteur c;

    public CodeCompteur(Compteur _c){
        this.c=_c;
    }

    public void run(){
        for(int i=0; i<10000; i++){
            synchronized(c){
                c.setValeur(c.getValeur()+1);
            }
        }
    }
}
```

- **Attention** : Ne pas synchroniser n'importe quoi
- Synchronizer des parties de codes qui terminent (sinon le verrou reste fermé !!!)
- Trouver où mettre les verrous est difficile !!!
- Attention aux deadlocks !

ATTENTION

- Ne pas synchroniser n'importe comment
- Rappeler vous que les verrous sont associés à des objets
 - Deux codes synchronisés sur le même objet ne pourront pas être exécutés en même temps
- Synchronizer des parties de code qui terminent sinon le verrou reste bloqué pour toujours
- Attention aux deadlocks !!!!
 - Deux thread attendent que le verrou pris par un autre thread se libère
- **Remarque :**
 - `synchronized int f(...)` { ... } est pareil que
 - `int f(....) { synchronized(this){...}}`
 - Le verrou des méthodes et le verrou de l'objet associé

Un autre problème de la concurrence

- On peut avoir des dépendances entre les sections critiques
- Par exemple, dans le problème des producteurs/consommateurs
 - Les producteurs écrivent une valeur dans une variable
 - Les consommateurs lisent les valeurs écrites dans la variable
 - On ne veut pas qu'un producteur écrase une valeur qui n'a pas été lue
 - On ne veut pas qu'une valeur soit lue deux fois
- Si les consommateurs sont plus rapides que les producteurs, alors les valeurs risquent d'être lues deux fois
- Si les producteurs sont trop rapides, les valeurs d'être perdues
- On ne veut pas que les producteurs et les consommateurs lisent et écrivent en même temps
- Comment faire ?

Première solution (1)

- On crée un objet **VariablePartagee** qui contient une valeur entière **val** et un booléen **pretaecrire**
- Si le booléen est à vrai, on peut écrire une fois la variable et on met le booléen à false
- Si le booléen est à faux, on peut lire une fois la variable et on met le booléen à vrai
- On garantit ainsi que chaque valeur ait écrite une fois et lue une fois

Première solution (2)

```
public class VariablePartagee {  
    public int val;  
    public boolean pretaecrire;  
  
    public VariablePartagee() {  
        val=0;  
        pretaecrire=true;  
    }  
  
    public int lire() {  
        while(pretaecrire==true) {}  
        pretaecrire=true;  
        return val;  
    }  
  
    public void ecrire(int v) {  
        while(pretaecrire==false) {}  
        pretaecrire=false;  
        val=v;  
    }  
}
```

Code Producteur

```
public class CodeProducteur implements Runnable{

    private VariablePartagee var;

    public CodeProducteur(VariablePartagee _var){
        this.var=_var;
    }

    public void run(){
        for(int i=0; i<100; i++){
            var.ecrire(i);
        }
    }
}
```

Code Consommateur

```
public class CodeConsommateur implements Runnable{

    private VariablePartagee var;

    public CodeConsommateur(VariablePartagee _var){
        this.var=_var;
    }

    public void run(){
        for(int i=0; i<100; i++){
            System.out.println(var.lire());
        }
    }
}
```


Code Principal

```
public class TestProdCons{
    public static void main(String[] args){
        try{
            VariablePartagee var=new VariablePartagee();
            CodeProducteur prod=new CodeProducteur(var);
            CodeConsommateur cons=new CodeConsommateur(var);
            Thread []t=new Thread[20];
            for(int i=0; i<10; i++){
                t[i]=new Thread(prod);
            }
            for(int i=10; i<20; i++){
                t[i]=new Thread(cons);
            }
            for(int i=0; i<20; i++){
                t[i].start();
            }
            for(int i=0; i<20; i++){
                t[i].join();
            }
        }
        catch(Exception e){
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Problème de cette méthode

- Les méthodes lire et ecrire de la variable partagée doivent être déclarées **synchronized** car elles manipulent le booléen **pretaecrire** et l'entier **val** de façon concurrente
- Cela ne suffit pas, pourquoi :
 - Un consommateur arrive
 - Il fait lire, il prend le verrou et il reste bloqué dans la boucle while en gardant le verrou
 - Si un producteur veut produire il doit prendre le verrou mais il ne peut pas car c'est le consommateur qu'il l'a
 - On doit éliminer cette attente active

Idée de solution

- Mettre un booléen pour savoir si la valeur doit être lue et écrite
- Faire synchronized pour assurer que plusieurs threads ne modifient pas ce booléen en même temps
- Utiliser les méthodes **wait()**, **notify()** et **notifyAll()**
 - **wait()** permet de relacher un verrou que l'on possède et attendre une notification
 - **notify()/notifyAll()** permet d'autoriser un/ tous les threads en attente de tenter de reprendre le verrou
 - ATTENTION : il est mieux de posséder le verrou pour faire ces opérations

Les nouvelles méthodes lire/écrire

```
public synchronized int lire() {
    try{
        while(pretaecrire==true) {
            wait();
        }
        pretaecrire=true;
        notifyAll();
    }
    catch(Exception e) {
        System.out.println(e);
        e.printStackTrace();
    }
    return val;
}

public synchronized void écrire(int v){
    try{
        while(pretaecrire==false) {
            wait();
        }
        pretaecrire=false;
        notifyAll();
    }
    catch(Exception e) {
        System.out.println(e);
        e.printStackTrace();
    }
    val=v;
}
```