Outils de développement,

programmation événementielle et IHM

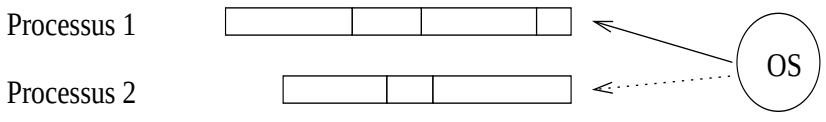
Chapitre 3 compléments Java, les threads

Cyrille Bertelle - UFRST Le Havre

0-0

10.1 Introduction

- Un programme est caractérisé par
  - son code
  - son espace mémoire (données)
- Sur un OS évolué, exécution concurrente possible des processus avec “temps partagé” géré par un scheduler



Si processus indépendants :

- contexte différent à chaque changement de processus ;
- possibilité de partager une mémoire (lourd à gérer) ;
- communications parfois nécessaires.

Une solution plus simple : *les threads* ou processus légers qui s'exécutent en parallèle mais partagent les données.

Exemple : concurrence de traitement lors de chargements d'images avec les navigateurs Web ou environnement graphique Java.

10.2 Création de threads avec Java

Threads gérés dans différents langages. Gestion lourde avec C et simplifiée avec Java qui propose deux solutions :

- Objet héritant de la classe `java.lang.Thread`. Deux objets de cette classe peuvent s'exécuter directement en concurrence.
- Objet implémentant l'interface `java.lang.Runnable`
  - On doit définir une méthode `run()` ;
  - On range ces objets dans des objets Threads (enveloppes) qui s'exécutent concurremment.

Exemples: `TestThread1.java` et `TestThread2.java`

TestThread1.java

```
class TPrint extends Thread {

    String txt;
    int attente;

    public TPrint(String t, int p) {
        txt = t;
        attente = p;
    }

    public void run() {
        for (int i=0; i<8; i++) {
            System.out.print(txt+i+" ");
```



```
            try {
                sleep(attente);
            }
            catch(InterruptedException e) {};
        }
    }
}

public class TestThread1 {
    static public void main(String args[]) {
        TPrint a = new TPrint("A", 100);
        TPrint b = new TPrint("B", 200);
        a.start();
        b.start();
```

```
    }
}

// résultat de l'exécution :
// A0 B0 A1 B1 A2 A3 B2 A4 A5 B3 A6 A7 B4 B5 B6 B7
```


TestThread2.java

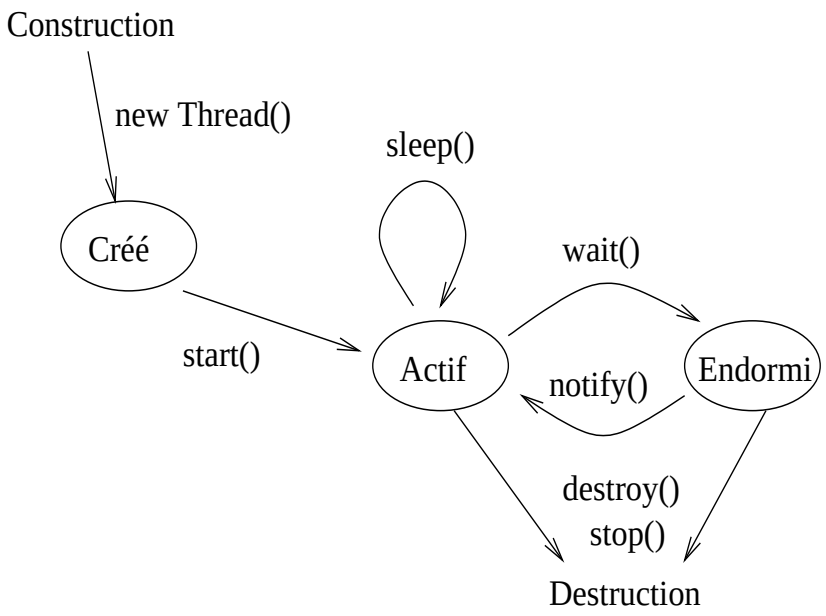
```
class TPrint implements Runnable {

    String txt;
    int attente;

    public TPrint(String t, int p) {
        txt = t;
        attente = p;
    }

    public void run() {
        for (int i=0; i<8; i++) {
            System.out.print(txt+i+" ");
```

```
            try {
                Thread.currentThread().sleep(attente);
            }
            catch(InterruptedException e) {};
        }
    }
}

public class TestThread2 {
    static public void main(String args[]) {
        TPrint a = new TPrint("A", 100);
        TPrint b = new TPrint("B", 200);
        new Thread(a).start();
        new Thread(b).start();
```



```
    }
}

// résultat de l'exécution :
// A0 B0 A1 B1 A2 A3 B2 A4 A5 B3 A6 A7 B4 B5 B6 B7
```

10.3 Les méthodes de gestion des threads



10.4 Synchronisation et Exclusion

Lorsque plusieurs threads travaillent sur des mêmes ressources, il est parfois utile ou nécessaire de limiter le parallélisme en assurant qu'un certain objet ne subisse pas en même temps plusieurs séquences d'actions concurrentes.

Exemple : Dans `TestThread3.java`, chaque thread affiche des mots caractère par caractère. Les mots sont mélangés. On souhaite conserver le parallélisme du traitement en garantissant que chaque mot entier ne soit pas coupé.

TestThread3.java

```
class TPrint extends Thread {

    String txt;

    public TPrint(String t) {
        txt = t;
    }

    public void run() {
        for (int j=0; j<3; j++) {
            for (int i=0; i<txt.length(); i++) {
                System.out.print(txt.charAt(i));
                try { sleep(100); }
```



```
                catch (InterruptedException e) {};
            }
        }
    }
}

public class TestThread3 {
    static public void main(String args[]) {
        TPrint a = new TPrint("bonjour ");
        TPrint b = new TPrint("au revoir ");
        a.start();
        b.start();
    }
}
```

```
// resultat de l'exécution :
// baoun jroeuvro ibro najuo urre vbooinrj oauur  revoir
```


Construction d'une classe moniteur

On définit une classe "moniteur" contenant une méthode "synchronized" qui va permettre l'écriture d'un mot dans son entier : lorsque cette méthode est appelée dans un thread, elle bloque le déroulement des autres threads jusqu'à ce qu'elle soit arrivée à la fin de son traitement.

Attention : pour que ce mécanisme fonctionne, il faut que les threads partagent le même moniteur. Ici, cela se traduit par la définition, dans la classe des threads, d'un champ "static" pour contenir le moniteur sur lequel on invoquera les méthodes synchronisées.
(cf. `TestThread4.java`)

TestThread4.java

```
class MoniteurImpression {
    synchronized public void imprime(String t) {
        for (int i=0; i<t.length(); i++) {
            System.out.print(t.charAt(i));
            try { Thread.currentThread().sleep(100); }
            catch (InterruptedException e) {};
        }
    }
}

class TPrint extends Thread {

    String txt;
```



```
    static MoniteurImpression mImp =
             new MoniteurImpression();

    public TPrint(String t) {
        txt = t;
    }

    public void run() {
        for (int j=0; j<3; j++)
            mImp.imprime(txt);
    }
}

public class TestThread4 {
```

```
    static public void main(String args[]) {
        TPrint a = new TPrint("bonjour ");
        TPrint b = new TPrint("au revoir ");
        a.start();
        b.start();
    }
}

// résultat de l'exécution :
// bonjour au revoir bonjour au revoir bonjour au revoir
```


10.5 Attente explicite : wait() - notify()

Les appels aux méthodes synchronisées sont appelés *sections critiques*. Il faut les utiliser avec prudence :

- elles suppriment la caractère concurrent du traitement d'où une dégradation des performances ;
- elles peuvent conduire à des situations d'interblocage.

Scénario Producteur/Consommateur

- Une mémoire “tampon” est gérée par un producteur d’objets et un consommateur. Cette mémoire ne peut contenir qu’un objet.
- On synchronise l’accès à cette mémoire (production et consommation non simultanées).
- Si le producteur doit déposé un objet alors qu’il en existe déjà un dans la mémoire tampon, il attends ... blocage !
- Même type de blocage pour un consommateur qui attends le dépot d’un objet.

Relâchement d’exclusion

La solution du problème précédent passe par la mise en œuvre d’une procédure de relâche d’exclusion :

- L’attente du Producteur ou du Consommation se fait au moyen de la méthode `wait()` (susceptible de déclencher une exception `InterruptedException`) qui relâche l’exclusion sur l’objet.
- Un processus sort de l’attente lorsqu’un autre processus exécute la méthode `notify()` (qui relance *un* processus en attente) ou la méthode `notifyAll()` (qui relance *tous* les processus en attente)

(cf. `ProdConsTest.java`)

ProdConsTest.java

```
class MoniteurProdCons {
    String tampon;
    boolean estVide = true;

    synchronized void prod(String m) {
        if (!estVide) {
            System.out.println("Producteur attend");
            try { wait(); }
            catch (InterruptedException e) {}
        }
        System.out.println("Produit : "+ m);
        tampon = m; estVide=false; notify();
    }
```



```
    synchronized void cons() {
        if (estVide) {
            System.out.println("Consommateur attend");
            try { wait(); }
            catch (InterruptedException e) {}
        }
        System.out.println("Consomme : "+ tampon);
        estVide=true; notify();
    }
}

class Producteur extends Thread {
    MoniteurProdCons tampon;
```

```
    public Producteur (MoniteurProdCons t) {
        tampon = t;
    }

    public void run() {
        tampon.prod("message1");
        tampon.prod("message2");
        try { sleep(100); }
        catch(InterruptedException e) {}
        tampon.prod("message3");
    }
}

class Consommateur extends Thread {
```



```
    MoniteurProdCons tampon;

    public Consommateur(MoniteurProdCons t) {
        tampon = t;
    }

    public void run() {
        tampon.cons(); tampon.cons(); tampon.cons();
    }
}

public class ProdConsTest {
    public static void main(String argv[]) {
        MoniteurProdCons tampon =
```

```
                    new MoniteurProdCons();
        new Producteur(tampon).start();
        new Consommateur(tampon).start();
    }
}
```