Chapitre 7

Le parallélisme en Java

L'espoir du **parallélisme** est de pouvoir faire exécuter plusieurs tâches à un ordinateur (avec plusieurs processeurs). Ceci trouve un début d'application sur certains gros ordinateurs (en particulier les *Cray* et la *Connection machine*). Sur un ordinateur possédant un, deux ou quelques microprocesseurs, on effectue du **pseudo-parallélisme** : on attribue une tranche de temps à chaque tâche lancée de façon à ce que les utilisateurs aient l'impression que ces tâches s'effectuent en même temps. Ce pseudo-parallélisme est depuis longtemps implémenté dans les systèmes d'exploitation. Le premier langage de programmation à tenir compte de celui-ci est le langage ADA du début des années 1980, repris dans le langage Java alors qu'il n'apparaissait ni en C ni en C++. Il s'agit des **processus légers** (*threads* en anglais).

7.1 Introduction à la programmation parallèle

7.1.1 Le parallélisme

Le besoin.- Effectuer des actions en parallèle est absolument indispensable de façon naturelle : nous respirons en même temps que nous marchons, par exemple. Nous avons besoin de ce parallélisme dans la vie courante : nous conduisons en écoutant de la musique ; je ne parle pas des étudiants qui ont envie de raconter leur week-end en écoutant le cours de programmation. Nous avons également besoin de ce parallélisme pour l'utilisation des ordinateurs : nous voulons pouvoir imprimer pendant que nous écrivons la page suivante d'un document et qu'une page Web se charge sur notre navigateur.

Les solutions.- Une première solution concernant l'informatique est de disposer de plusieurs ordinateurs. Cette solution est onéreuse et demande de la place. De plus il faut transférer le fichier d'un ordinateur à l'autre pour qu'on puisse imprimer sur l'un tant que l'on travaille sur la suite du document sur l'autre. C'est d'ailleurs l'un des premiers problèmes du parallélisme, celui du *partage des données*.

Une autre solution consiste à ne disposer que d'un seul ordinateur avec de nombreux processeurs. On parle de **parallélisme** proprement dit dans ce cas. Des essais de réalisation existent, non pas pour les problèmes évoqués ci-dessus mais pour des problèmes de calculs scientifiques tels que la prévision météorologique. La mise en place n'est pas simple. Nous renvoyons à un cours de parallélisme pour les problèmes, problèmes que nous allons de toute façon rencontrer également avec les processus légers.

Une troisième solution, qui date des années 1950, consiste à diviser le temps d'utilisation du processeur en **tranches de temps** (*time slice* en anglais) et d'attribuer des tranches de temps à chacune des tâches de façon telle qu'on ait l'impression que les tâches se déroulent en parallèle. On parle quelquefois de **pseudo-parallélisme** pour cette méthode. Cette méthode a d'abord été implémentée pour les systèmes d'exploitation sur lesquelles plusieurs utilisateurs pouvaient lancer une tâche à la fois, puis plusieurs tâches à la fois. C'est le cas de tous les systèmes d'exploitation modernes mais rappelons qu'avec MS-DOS, dont on peut encore utiliser une émulation *via* une fenêtre de commande sous Windows, on ne peut lancer qu'une application à la fois.

7.1.2 La programmation parallèle

Les solutions.- La programmation parallèle pour du vrai parallélisme est délicate et pose encore de nombreux problèmes ; nous renvoyons à un cours de programmtion parallèle pour une initiation.

Comme nous l'avons dit ci-dessus, le pseudo-parallélisme est pendant longtemps resté l'apanage du système d'exploitation. Les concepteurs du langage ADA, au début des années 1980, ont pris ce phénomène en compte. Le langage Java s'en est inspiré pour ses **processus légers** (*threads* en anglais ; Sun parle d'ailleurs de *multithreading* au lieu de parallélisme ou de pseudo-parallélisme).

Implémentation.- On peut bien sûr implémenter nous-même ce pseudo-parallélisme grâce à une boucle qui effectue un petit peu de la première tâche, puis un petit peu de la seconde tâche et ainsi de suite. C'est d'ailleurs ce que l'on doit faire en langage C ou en C++. Un tel programme devient très vite complexe. On a donc intérêt à diviser le travail en deux parties : la conception de chacune des tâches, d'une part, et le pseudo-parallélisme d'autre part. Java permet ceci en nous donnons des outils tout faits pour le pseudo-parallélisme.

7.1.3 Notion de processus léger

Définition.- Un **processus léger** (*thread* en anglais, bien que la traduction littérale soit 'fil') est une unité de programme exécutée indépendamment des autres unités de programme.

Remarque.- On peut se demander pourquoi on a besoin du qualificatif 'léger'. Il s'agit en fait d'une question d'implémentation. Les processus sont connus depuis longtemps dans le vocabulaire des systèmes d'exploitation. Un processus possède, en particulier, son espace mémoire propre. Des processus légers, par contre, partagent le même espace mémoire.

Exécution des processus légers.- Les processus légers peuvent être exécutés séquentiellement et il n'y a pas alors besoin des outils du parallélisme. Mais, bien entendu, l'idée est de concevoir des processus légers pour les exécuter en parallèle.

7.2 Mise en place des processus légers

7.2.1 Mise en place d'un processus unique

La mise en place d'un processus unique n'a pas tellement d'intérêt (autant concevoir un programme habituel) mais il s'agit d'une étape vers la mise en place de plusieurs processus.

7.2.1.1 Syntaxe

Définition d'un processus léger.- En Java un processus léger est un objet d'une classe dérivée de la classe :

```
Thread
```

du paquetage `java.lang`.
La méthode :

```
public void run()
```

doit nécessairement être surchargée. Il s'agit de la méthode qui est lancée automatiquement lors de l'accès à un processus.

Appel d'un processus.- La gestion d'appel des processus est laissée à l'initiative du programmeur. Dans son programme principal, le programmeur doit déclarer un objet processus, l'instantier et l'appeler grâce à la méthode :

```
void start()
```

de la classe `Thread` (et donc de ses classes dérivées).

7.2.1.2 Exemple

Exemple.- Écrivons une application java mettant en place un processus qui affiche les entiers de 1 à 9.

Le programme.- Ceci nous conduit au programme suivant :

```
class AfficheThread extends Thread
   {
   public void run()
      {
```

```
      for (int i = 1; i < 10; i++)
         System.out.println(i);
      }
   }

public class Exemple
   {
   public static void main(String args[])
      {
      AfficheThread t;
      t = new AfficheThread();
      t.start();
      }
   }
```

L'exécution nous donne bien l'affichage des neuf premiers entiers mais on ne voit pas l'intérêt d'utiliser un processus léger pour ce problème.

Remarque.- Une classe processus peut être exécutable :

```
public class AfficheThread extends Thread
   {
   public void run()
      {
      for (int i = 1; i < 10; i++)
         System.out.println(i);
      }

   public static void main(String args[])
      {
      AfficheThread t;
      t = new AfficheThread();
      t.start();
      }
   }
```

mais ceci est à éviter.

7.2.2 Mise en place de plusieurs processus

7.2.2.1 Appel sans parallélisme

Introduction.- Si on fait appel à plusieurs processus sans rien ajouter, ils seront effectués séquentiellement.

Exemple.- Appelons deux processus :

```
class AfficheThread extends Thread
   {
   public void run()
      {
```

```
      for (int i = 1; i < 10; i++)
         System.out.println(i);
      }
   }

public class DeuxThread
   {
   public static void main(String args[])
      {
      AfficheThread t1, t2;
      t1 = new AfficheThread();
      t2 = new AfficheThread();
      t1.start();
      t2.start();
      }
   }
```

Lors de l'exécution, les chiffres de 1 à 9 sont affichés une première fois puis une seconde fois. Ceci nous montre que le parallélisme n'est pas géré si on ne l'indique pas explicitement.

7.2.2.2 Gestion du parallélisme en cédant la place

Introduction.- Une première façon de gérer le parallélisme consiste à parsemer dans le code des processus des appels à la fonction :

```
void yield()
```

(céder sa place, abandonner en anglais). Ceci interrompt l'exécution du processus durant un très court instant, instant dont peuvent profiter les autres processus pour prendre place.

Exemple.- Réécrivons notre application précédente en introduisant cette fonction :

```
class AfficheThread extends Thread
   {
   public void run()
      {
      for (int i = 1; i < 10; i++)
         {
         System.out.println(i);
         yield();
         }
      }
   }

public class Parallele
   {
   public static void main(String args[])
      {
      AfficheThread t1, t2;
      t1 = new AfficheThread();
      t2 = new AfficheThread();
      t1.start();
```

```
      t2.start();
      }
   }
```

Exécution.- Lors de l'exécution sous Linux, on a une stricte alternance des processus : 1, 1, 2, 2, 3, 3... Sous Windows, cela dépend de l'exécution. On a, par exemple :

```
C:>java Parallele
1
1
2
3
2
4
3
5
6
4
7
5
6
8
7
9
8
9
```

Remarque.- Comme nous venons de le voir sur cet exemple, le langage Java est portable d'un système informatique à l'autre d'un point de vue syntaxique mais non du point de vue sémantique. En effet, comme pour le paquetage `awt`, Java ne gère pas le fonctionnement concurrent des processus légers lui-même : il le confie au système d'exploitation.

7.2.2.3 Gestion du parallélisme par assoupissement

Assoupissement.- On ne contrôle pas le temps laissé aux autres processus pour pouvoir prendre la main avec le procédé précédente. Un autre procédé consiste à endormir le processus en cours durant un temps déterminé. Ceci s'effectue grâce à la méthode :

```
void sleep(int)
```

de la classe `Thread`.

Exception.- Lorsqu'un processus endormi est réveillé (*interrompu* dans le vocabulaire Java), une exception du type `InterruptedException` est levée. On doit donc nécessairement capturer cette exception, même si elle ne donne lieu à aucun traitement.

Exemple.- Réécrivons notre application en utilisant l'assoupissement au lieu de l'abandon :

```
class AfficheThread extends Thread
   {
   public void run()
      {
```

```
      for (int i = 1; i < 10; i++)
         {
         try
            {
            System.out.println(i);
            sleep(500);
            }
         catch(InterruptedException e)
            {
            }
         }
      }
   }

public class Endormi
   {
   public static void main(String args[])
      {
      AfficheThread t1, t2;
      t1 = new AfficheThread();
      t2 = new AfficheThread();
      t1.start();
      t2.start();
      }
   }
```

On a alors l'affichage avec une alternance stricte puisque l'autre processus a suffisamment de temps pour prendre la main lorsque l'autre est endormi.

7.3 La synchronisation

7.3.1 Notion

Le problème.- Des processus légers s'exécutant en parallèle peuvent avoir à accéder à la même *ressource* au cours de leur exécution. Le moment où ils accèdent à cette ressource peut ne pas avoir d'importance ou en avoir.

Considérons, par exemple, deux tâches qui auront à imprimer un document à un certain moment. Si l'imprimante est entièrement partagée et accepte comme entrée des caractères (c'était le cas des anciennes imprimantes), on risque d'avoir l'impression du début du premier document, puis du second document et ainsi de suite. Ce n'est évidemment pas ce qui est voulu. Si l'imprimante imprime page par page, on aura à trier le paquet de pages imprimées, ce qui est également gênant.

Il faut donc **synchroniser** les processus de façon à ce que ce phénomène ne se passe pas.

La solution de Java.- Il existe de nombreuses façons de résoudre ce problème de synchronisation. Les concepteurs de Java ont choisi la méthode des **moniteurs** de HOARE (1974), assez fruste mais fonctionnant : lors de la définition des processus, le programmeur décide que certaines ressources, qui sont nécessairement des objets en Java, sont des moniteurs ; lorsque qu'un processus de ce type a accès à un objet marqué comme moniteur, cet objet est **verrouillé**, c'est-à-dire qu'aucun autre processus, de ce type ou d'un autre type, ne peut accéder à cet objet tant que l'action du

processus sur cet objet n'est pas terminée.

7.3.2 Syntaxe

Marquage d'un objet en tant que moniteur.- Pour marquer que l'objet *objet* est un moniteur dans une partie de code, on crée un bloc précédé du préfixe :

`synchronized(`*objet*`)`

Cas d'une méthode.- La philosophie de la programmation orientée objet est qu'une action est représentée par une méthode. Il n'y a donc aucune raison de verrouiller une partie de code mais plutôt tous les objets partagés d'une méthode. Pour cela, lors de la définition de la méthode on utilise le modificateur :

`synchronized`.

7.3.3 Un exemple

Le problème.- Émulons le problème de l'impression. Concevons deux processus, l'un affichant (à l'écran) les nombres de 1 à 10 et l'autre les lettres de 'a' à 'f' et faisons-les partir presqu'en même temps.

Ces impressions sont suffisamment rapides pour qu'on n'ait pas à les faire exécuter en parallèle. Imaginons cependant qu'une action est effectuée avant l'impression et que nous ne sachions pas quelle doit être la première série d'impression.

Si on ne synchronise pas, on a toutes les chances de voir afficher un chiffre, une lettre, un chiffre, une lettre et ainsi de suite.

Exercice.- *Écrire un programme Java mettant en place ces deux processus sans synchronisation et observer le résultat de l'exécution.*

Un programme.- Écrivons un programme Java mettant en place ces deux processus, dont l'un verrouille la sortie sur écran (représenté en Java par `System.out`, rappelons-le) :

```
class Compter extends Thread
   {
   private static final int DELAI = 500;

   public void run()
      {
      try
         {
         synchronized(System.out)
            {for (int i = 0; i < 10; i++)
                {
                System.out.println(i);
                sleep(DELAI);
                }
            }
         }
      catch(InterruptedException e)
         {
         }
```

```
      }
   }

class Alpha extends Thread
   {
   private static final int DELAI = 500;

   public void run()
      {
      try
         {
         for (char c = 'a'; c <= 'f'; c++)
            {
            System.out.println(c);
            sleep(DELAI);
            }
         }
      catch(InterruptedException e)
         {
         }
      }
   }

public class Synchro
   {
   public static void main(String [] args)
      {
      Compter C;
      Alpha A;
      C = new Compter();
      A = new Alpha();

      A.start();
      C.start();
      }
   }
```

L'exécution donne :

```
linux:# java Synchro
a
0
1
2
3
4
5
6
7
8
```

```
9
b
c
d
e
f
```

c'est-à-dire que l'on commence à afficher les lettres (l'écran n'étant pas verrouillé), puis tous les chiffres (car l'écran est alors verrouillé) puis la suite des lettres.

Exercices

Exercice 1.- *Concevoir une classe de processus affichant dix fois l'heure et un message, une seconde s'écoulant entre chaque affichage. Écrire une application Java lançant deux tels processus.*

Remarque.- L'instantiation d'un objet de la classe :

`Date`

du paquetage `java.util` permet de récupérer la date et l'heure. On peut afficher un tel objet.

Le programme.- Ceci nous conduit au programme suivant :

```
import java.util.Date;

class SalutThread extends Thread
   {
   private String salut;
   private static final int REPETITIONS = 10;
   private static final int DELAI = 1000;

   public SalutThread(String message)
      {
      salut = message;
      }

   public void run()
      {
      try
         {
         for (int i = 0; i < REPETITIONS; i++)
            {
            Date maintenant = new Date();
            System.out.println(maintenant + " " + salut);
            sleep(DELAI);
            }
         }
      catch(InterruptedException e)
         {
         }
      }
   }

public class SalutThreadTest
   {
   public static void main(String [] args)
      {
      SalutThread t1, t2;
      t1 = new SalutThread("Bonjour");
      t2 = new SalutThread("Au revoir");

      t1.start();
```

```
      t2.start();
      }
   }
```

Exécution.- Une exécution nous donne :

```
linux:# java SalutThreadTest
Sat Mar 09 15:13:50 GMT 2002 Bonjour
Sat Mar 09 15:13:50 GMT 2002 Au revoir
Sat Mar 09 15:13:51 GMT 2002 Bonjour
Sat Mar 09 15:13:51 GMT 2002 Au revoir
Sat Mar 09 15:13:52 GMT 2002 Bonjour
----------
```

ce qui montre bien que les deux processus s'exécutent en parallèle.