Chapitre 9

Les threads

9.1 *Multitasking* et threads

La notion de *thread*[1] est fondamentale en programmation moderne. Un PC sous DOS n'exécutait qu'un seul programme à la fois, même si certains processus fonctionnaient par le moyen d'interruptions. L'interruption ne permettait aucun parallélisme. Par exemple, lors de la frappe d'une touche clavier, le processeur recevait une interruption matérielle, sauvegardait son contexte, allait décoder la touche frappée et la mémorisait avant de reprendre le travail interrompu.

Les ordinateurs modernes ne se contentent plus d'effectuer un programme à la fois. Un mécanisme complexe au sein du système d'exploitation permet à plusieurs programmes de s'exécuter en même temps. On sait, bien sûr, qu'en réalité, le processeur n'exécute qu'un seul programme à la fois[2], mais un subil jeu de partage de ressources et une régulation interne permettent à plusieurs programmes de s'exécuter presque simultanément, chacun recevant une petite tranche de temps du processeur pour exécuter quelques centaines de milliers d'instructions. Les systèmes d'exploitation modernes réalisent cela avec brio. Il est loin le temps où Windows plantait parce qu'un programme accaparait toutes les ressources de la machine. A titre d'exemple, la machine sur laquelle je suis en train de composer ces notes fait tourner concurrement 145 processus (avec environ 6000 fichiers ouverts, en ce compris les nombreux sockets nécessaire au fonctionnement du réseau)[3].

9.1.1 Threads

La notion de *thread* (fil d'exécution) suppose un système d'exploitation multitâche. Mais il ne faut pas confondre thread et processus. Dans les deux cas, il s'agit de portions de code exécuté séparément. Lorsqu'on parle de parallélisme, nous savons qu'il s'agit d'une illusion, les différents processus et threads se partageant le processeur, sur base volontaire ou non. Un *processus* comporte au moins un thread. Un processus possède une pile (gestion des appels de procédures et variables locales), un segment de code et un segment de données (variables statiques et globales). Il est possible de réaliser une copie intégrale d'un processus (par l'entremise de l'instruction *fork*). Dans ce cas, les deux processus possèdent chacun leurs segments. Dans le cas de threads, le partage se maintient au niveau du code et des données, mais chaque

[1] En français, on dit *processus léger*, ce qui est un peu long...

[2] Cela se complique un peu quand il y a plusieurs processeurs...

[3] La commande `ps aux|wc -l` permet d'afficher toutes les tâches en cours et de les compter. On peut également trouver 145 répertoires portant de noms numériques dans le répertoire `/proc` qui correspondent à ces différents processus.

thread possède sa pile propre. Une conséquence importante est que deux threads issus d'un même processus peuvent accéder aux mêmes données. La communication entre processus est beaucoup plus complexe à gérer et ne correspond jamais à des manipulations de variables : lorsque plusieurs processus indépendant tournent sur une machine, ils ne peuvent pas interagir entre eux directement puisqu'ils disposent chacun de leur mémoire propre. S'ils doivent communiquer, ils peuvent le faire par le système des fichiers (un cas particulier étant constitué par les tubes ou *pipes*, des fichiers non mémorisés sur disque). Dans le cas de threads issus d'un même processus, c'est beaucoup plus complexe, puisqu'ils peuvent utiliser les mêmes données. Il est de la responsabilité du programmeur de veiller à ce que ces interactions ne déclenchent pas de catastrophes.

9.1.2 Exemples

Si le *multitasking* semble assez évident à justifier (par exemple écouter un morceau de musique mp3 pendant qu'on écrit une lettre ennuyeuse ou regrader une vidéo pendant l'impression d'un document de 30 pages), les threads peuvent sembler moins nécessaires à l'utilisateur. On peut pourtant trouver bien des exemples où un même programme réalise plusieurs tâches simultanées :

- une interface graphique continue à se mettre à jour et à accepter des actions de l'utilisateur pendant qu'une autre tâche se déroule
- le navigateur interne de KDE sous Linux offre une représentation spatiale de la taille de dossiers qui ne peut pas évaluer instantanément l'espace occupé par tous les fichiers d'une arborescence. Il se met à jour au fur et à mesure de l'exploration du disque qui s'effectue dans un thread séparé. Pendant ce temps, l'utilisateur peut continuer à manipuler le programme et lancer d'autres actions
- un gestionnaire de courrier est capable de consulter le serveur pour annoncer la présence de nouveaux messages pendant que l'utilisateur est en train d'en rédiger un autre
- un lecteur de fichiers multimédia joue la liste des morceaux choisis par l'utilisateur tout en parcourant le disque dur à la recherche d'ajouts ou de suppressions de morceaux

9.2 La classe `Thread`

La plupart des langages se révèlent incapables, sans extension, de gérer les threads. Une notable exception sont les langages Java et C#. Dans le cas de Java, la manière dont les différents threads se partagent le processeur dépend du système d'exploitation. Nous n'évoquerons pas cette question, qui est centrale pour les systèmes en temps réel.

Le mécanisme des threads se concentre dans la méthode `start()` de la classe `Thread`. Cette méthode incorpore un mécanisme complexe qui va créer un nouveau thread à l'intérieur de la machine Java. Dès que le processus est lancé, la méthode `start()` se termine et on passe à l'instruction suivante. Le processus créé se déroule en parallèle et appelle la méthode `run()` de l'instance qui a exécuté `start()`. Il appartient au programmeur de maintenir l'exécution par deux techniques : en appelant des méthodes d'autres objets ou en faisant durer artificiellement le thread à l'aide d'une boucle longue, voire infinie. Si on n'a rien de particulier à faire que de maintenir le thread en vie, on placera un appel de la méthode `sleep()`[4] de manière à ne pas monopoliser inutilement le processeur dans une boucle vide. Un thread endormi n'utilise quasiment aucune ressource.

[4]Cette méthode ne supportant pas d'être interrompue, il convient de prévoir l'exception qui risque de se déclencher si quelqu'un interrompait le thread. Nous verrons cela plus loin.

Voici un exemple de thread qui va afficher un message sur la console de sortie toutes les secondes.

```
public class MonThread extends Thread{

    /** Crée une nouvelle instance de MonThread */
    public MonThread() {
    }
    public void run() {
        while(true){
            System.out.println("encore");
            try {
                sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }
}
```

Pour lancer ce thread, qui doit en principe durer l'éternité, on peut imaginer ce code :

```
MonThread thread1 = new MonThread();
thread1.start();
```

Il faut noter que la méthode `start()` ne peut s'utiliser qu'une seule fois. Une fois interrompu, le thread contenu dans l'instance ne peut pas être réinitialisé. Il faut utiliser une nouvelle instance.

En pratique un thread ne s'interrompt que s'il le décide : si la méthode run se termine naturellement ou si on interrompt la boucle. Ainsi, on pourrait remplacer la boucle `while(true)` par une boucle avec compteur et une valeur maximale. On peut aussi subordonner l'arrêt de la boucle à une variable qui sera manipulée par un moyen quelconque.

Les premières versions de Java prévoyaient trois méthodes pour gérer le déroulement d'un thread : `stop()`, `suspend()` et `resume()`. Leur usage est fortement découragé : en effet, l'interruption d'un thread, que ce soit pour l'arrêter définitivement ou temporairement, étant déclenchée à l'extérieur de lui, il n'est pas possible de prévoir si l'action des routines du thread sur le contexte extérieur est dans un état stable (c'est le cas des verrous qu'il peut placer, par exemple). On considère comme beaucoup plus sûr de laisser le thread lui-même décider du moment où il va s'interrompre. Dans notre exemple, il suffirait de prévoir un variable logique `continuer` ayant la valeur `true` au départ qu'on pourra faire passer à `false` à tout moment. La boucle se réécrira à l'aide de `while(continuer)`. La modification de la variable de contrôle peut se faire à l'aide d'un accesseur public qui sera appelé depuis le thread ou depuis un autre thread.

9.3 L' interface `Runnable`

L'utilisation d'une sous-classe de Thread présente un inconvénient majeur : il devient impossible de prévoir une classe dérivée sous forme de thread. Heureusement, il existe l'interface `Runnable`, qui a pour seul but de forcer la création d'une méthode `run()` dans une classe. Il suffit de passer une instance de cette classe en paramètre du constructeur d'une instance de

Thread dont on peut ensuite appeler la méthode `start()`. Le code de la version « runnable » de notre classe sera presque identique à ce qui a été vu plus haut. C'est l'instanciation qui est différente :

```
MonThread2 thread2 = new MonThread2();
new Thread(thread2).start();
```

On peut enfin utiliser une petite astuce : doter la nouvelle classe d'une méthode `start()` qui réalise automatiquement la création du thread et son démarrage.

```
public class MonThread3 implements Runnable{
    private boolean continuer = true;

    /** Crée une nouvelle instance de MonThread3 */
    public MonThread3() {
    }
    public void run() {
        while(continuer){
            System.out.println("again/encore");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }

    public void start(){
        new Thread(this).start();
    }
    public void setContinuer(boolean continuer) {
        this.continuer = continuer;
    }
}
```

Cette fois, la création et l'activation du thread se fera comme si on avait affaire à un descendant de la classe Thread.

```
MonThread3 thread3 = new MonThread3();
thread3.start();
```

La différence importante entre cette version et la première est l'appel de la méthode `sleep()`. La classe que nous définissions n'en possède pas et il n'est donc pas possible d'appeler simplement `sleep()`. Heureusement, la classe Thread possède une méthode statique qui peut jouer le même rôle.

9.4 Interrompre proprement un thread

Nous avons vu plus haut que la méthode `suspend()` présentait l'inconvénient d'interrompre le thread sans tenir compte de l'état du contexte. Il est heureusement possible de demander à un thread d'attendre qu'on le libère. La différence réside dans le fait que le thread

décidera lui-même du moment où il se mettra à attendre. La classe `Object` implément trois méthodes pour gérer ce mécanisme : `wait()`, `notify()` et `notifyall()`. Malgré cela, l'utilisation de ce mécanisme doit faire l'objet de la plus grande attention : en effet, il faut penser que dans un système par essence multitâche, une interruption peut survenir au pire moment. L'exemple complet qui suit montre un compteur susceptible d'être démarré, momentanément interrompu, relancé et définitivement arrêté. Afin de simplifier l'écriture, j'ai écrit une classe dérivé directement de `Thread` et non une implémentation de `Runnable`. La classe utilise l'énumération `Etat` qui regroupe les états possibles du thread : né, actif, suspendu et mort. J'ai laissé tomber les directives `import` et `package`.

```
// fichier Etat.java
public enum Etat {NE,ACTIF,ENDORMI,SUSPENDU,MORT};

// fichier Compteur.java
public class Compteur extends Thread{

    Etat etat;
    Afficheur afficheur;

    public Compteur(Afficheur afficheur) {
        this.afficheur=afficheur;
        etat=Etat.NE;
    }

    public Etat getEtat(){
        return etat;
    }
    public void tuer(){
        etat=Etat.MORT;
    }

    @Override
    public void start(){
        etat=Etat.ACTIF;
        super.start();
    }

    public void suspendre(){
        etat=Etat.SUSPENDU;
    }

    // IMPORTANT synchronized
    public synchronized void ranimer(){
        etat=Etat.ACTIF;
        notify();
     }

    @Override
    public void run(){
        System.err.println("démarrage");

        // On gère la mort du thread manuellement pour éviter tout problème
        while(etat!=Etat.MORT){
```

```
            try{
               afficheur.incrementer();
               // endormissement d'une seconde
               Thread.sleep(1000);
               // suspension gérée manuellement
               synchronized(this){
                    while(etat==Etat.SUSPENDU) wait();
               }
            }
            catch(Exception e){
                System.exit(1);
            }
        }
    }
}
```

Quelques commentaires :

1. afin de pouvoir synchroniser le thread avec sa variable `etat`, j'ai surdéfini la méthode `start()` pour faire changer la variable d'état avant d'appeler la méthode normale.
2. la terminaison est contrôlée, comme dans les exemples précédents, par une boucle *while*. La variable `etat` peut prendre la valeur finale grâce à la méthode `tuer()`. On voit donc que la mort effective se produira toujours en début de boucle. On supposera que tous les verrous ou blocages seront désactivés avant la fin de la boucle. On peut évidemment compliquer le code pour empêcher ou retarder la sortie de boucle si des blocages subsistent.
3. le mécanisme de suspension temporaire est localisé dans la boucle `while` interne. On pourrait croire à tort qu'un simple `if` suffirait. La documentation insiste sur la nécessité d'utiliser une boucle et de l'inscrire dans un contexte synchronisé, qui ne peut en aucun cas être interrompu.
4. la méthode ranimer, elle aussi comportant la caractéristique `synchronized` consiste en un appel de la méthode `notify()`. Il n'est pas nécessaire que la méthode soit appelée dans l'objet. On pourrait avoir un appel `instance.notify()` ailleurs, à condition qu'on synchronise sur l'objet.
5. l'absence des `synchronized` provoque respectivement une exception pour ce qui concerne la méthode `ranimer()` et l'arrêt du thread pour l'appel de `wait()`.

Je propose sur mon site une classe `PanneauControle` qui permet de jouer avec le thread. Cette clase implémente l'interface `Afficheur` dont la seule méthode est `incrementer()`. On voit que le compteur appelle cette méthode dans la boucle, qui dure environ une seconde. Cela permet au compteur de se manifester quand il s'exécute. Une deuxième version utilise deux compteurs simultanés pour montrer leur indépendance. J'ai dû modifier un peu la classe `Compteur` pour qu'elle sache dans quelle zone de texte elle doit afficher les nombres croissants.

9.5 Attendre qu'un thread se termine

Lorsqu'un thread a pour simple mission de réaliser une tâche sans pour autant bloquer le programme principal, il arrive parfois que ce dernier ait besoin à un moment du résultat

calculé par un thread pour pouvoir continuer. On dispose de la méthode `join()`. Elle ne peut s'employer qu'avec les threads dérivés de `Thread`, mais on peut facilement accéder à la méthode join du thread interne si la classe implémente `Runnable`. L'exemple suivant l'illustre.

```
public class LongueTache implements Runnable {

    private Thread thread;
    private int iterations;

    public LongueTache(int iter){
        thread = new Thread(this);
        iterations=iter;
    }

    public void start(){
        thread.start();
    }

    public void sleep(long millis)throws InterruptedException {
        Thread.sleep(millis);
    }

    /**
     * Reprise de la méthode join du thread interne
     * @throws java.lang.InterruptedException
     */
    public void join()throws InterruptedException{
        thread.join();
    }

    @Override
    public void run() {
        int n=0;
        while(n<iterations){
            n++;
            try {
                sleep(1000);
            } catch (InterruptedException ex) {
                                System.err.println("Problème!!!");
                            }
            System.err.println("Message du thread numéro "+n);
        }
        System.err.println("Le thread est terminé");
    }

    public static void main(String[] arg){
        LongueTache lt = new LongueTache(20);
        lt.start();
        System.out.println("J'ai lancé le thread");
        for(int i=1;i<=20;i++){
            try {
                Thread.sleep(500);
```

```
            } catch (InterruptedException ex) {
                                System.err.println("Problème!!!");
                            }
            System.out.println("Message␣du␣programme␣principal␣numéro␣"+i);
        }
        System.out.println("Travail␣terminé.");
        System.out.println("J'attends␣la␣fin␣du␣thread");
        try {
            lt.join();
        } catch (InterruptedException ex) {
                                System.err.println("Problème!!!");
        }
        System.out.println("Je␣peux␣terminer␣mon␣boulot␣maintenant");
        System.out.println("Voilà!");
    }
}
```

La méthode `join()` est surchargée par une autre méthode qui accepte un paramètre semblable à celui de `sleep()`d afin de prévoir un temps limite au-delà duquelle, on se lasse d'attendre.

9.6 Threads et synchronisation

Nous avons vu une première approche des problèmes qui surviennent lors de la « synchronisation ». L'exemple suivant va proposer des instances concurrentes de threads, qui manipulent ensemble les mêmes données. Je donne une première version de la classe, sous une forme abstraite (elle ne possède pas de méthode `run()`).

```
public abstract class ListeurAbstrait implements Runnable {

    public static ArrayList<Integer> listeCommune;
    protected int iIntervalle;
    protected Thread thread;
    protected String name;
    PrintStream canalSortie;

    {
        listeCommune = new ArrayList<Integer>();
    }

    public ListeurAbstrait(String strName, int iIntervalle,
        PrintStream canal) {
        thread = new Thread(this);
        this.iIntervalle = iIntervalle;
        this.name = strName;
        this.canalSortie=canal;
        //TODO Collections.synchronizedList()
    }

    protected void modifierListe() {
        int i = (int) (Math.random() * 2);
        if (i == 1) {
```

```
            canalSortie.println("Element␣ajouté␣par␣" + name);
            Integer intN = (int) (Math.random() * 20);
            listeCommune.add(intN);
        } // supprimer un élément si possible
        else {
            canalSortie.println("Elément␣supprimé␣par␣" + name);
            if (listeCommune.size() > 0) {
                listeCommune.remove(0);
            }
        }
    }

    protected void afficherListe() {
        canalSortie.println("Affichage␣par␣"+name);
        for (int ii = 0; ii < listeCommune.size(); ii++) {
            try {
                Thread.sleep(iIntervalle);
            } catch (Exception ex) {
                System.err.println("Problème␣pendant␣le␣sommeil");
            }
            canalSortie.print(listeCommune.get(ii) + "-");
        }

    }

    /* La méthode run sera définie dans les classes filles
    @Override
    public void run(){
    }
    */


    /**
     * Démarrage du thread
     */
    public void start() {
        thread.start();
    }
}
```

À nouveau quelques remarques :

1. la classe possède une variable statique `listeCommune` qui sera partagée par toutes les instances.
2. la méthode `modifierListe()` ajoute ou retranche un élément à la liste, de manière aléatoire.
3. la méthode `afficherListe()` fait ce que son nom indique. Le gros danger est évidemment que la liste soit modifiée au cours de son affichage (par exemple, par disparition d'un élément). Pour augmenter la probabilité d'un tel incident, j'ai placé un endormissement au milieu de celle-ci.
4. le dernier paramètre du constructeur me permet de fournir soit `System.out` soit `System.err` lors de l'instanciation, de manière à distinguer par la couleur les messages envoyé par deux instances différentes.

Voici une première ébauche de la classe réelle, qui va essentiellement définir la boucle `run()`.

```
public class ListeurNormal extends ListeurAbstrait implements Runnable {

    public ListeurNormal(String strName, int iIntervalle,
                PrintStream canal) {
        super(strName, iIntervalle,canal);
    }

    /**
     * Boucle de survie du thread
     */
    @Override
    public void run() {

        while (true) {
            modifierListe();
            afficherListe();
        }

    }

    public static void main(String[] arg) {
        ListeurNormal l1 = new ListeurNormal("Listeur 1", 500,System.out);
        ListeurNormal l2 = new ListeurNormal("Listeur 2", 600,System.err);
        l1.start();
        l2.start();
    }
}
```

En instanciant deux listeurs, je vois une série de messages s'afficher, provenant de chacun d'eux, jusqu'au moment où l'un des deux rencontre une exception, par suite de la disparition d'un élément de la liste qu'il est sur le point d'afficher. Le thread de ce listeur meurt, mais l'autre continue imperturbablement.

Pour éviter ce problème, il suffit de rendre les contextes critiques protégés.

```
public class ListeurSync extends ListeurAbstrait implements Runnable {

    public ListeurSync(String strName, int iIntervalle,PrintStream canal) {
        super(strName, iIntervalle,canal);
    }

    @Override
    public void run() {
        while (true) {

            synchronized (listeCommune) {
                modifierListe();
            }
            synchronized (listeCommune) {
                afficherListe();
            }
            System.err.println("/");
```

```
        }

    }

    public static void main(String[] arg) {
        ListeurSync listeur1 = new ListeurSync("Listeur␣1", 500,System.out);
        ListeurSync listeur2 = new ListeurSync("Listeur␣2", 600,System.err);
        listeur1.start();
        listeur2.start();
    }
}
```

Cette fois, l'exécution se déroule sans problème.

9.7 Et plus encore...

Le chapitre sur les threads est à l'image du cours. Il reste tant de choses à explorer dans Java...

Pour ce qui concerne les threads, il faudrait encore parler des apports de Java 5. Notamment, une nouvelle manière de considérer la synchronisation avec la disposition de verrous (`Lock`) et la possibilité de poser des verrous sélectifs qui n'interdisent le partage que lors d'une écriture. La présence des interfaces `Queue` et `BlockingQueue` permet également de gérer plus efficacement le mécanisme du producteur/consommateur. Le problème est simple à exposer. Un consommateur doit attendre qu'un producteur ait fini de produire la ressource dont il a besoin pour réaliser une tâche (par exemple, examiner un message acheminé par un autre thread). La situation se complique lorsqu'il y a plusieurs producteurs et plusieurs consommateurs. Les producteurs déposent leurs ressources à la fin de la queue et les producteurs se servent au début. Lorsqu'aucune ressource n'est disponible, il faut attendre. Toute cette mécanique est facilitée dans Java 5.