Introduction

Il est assez fréquent de faire plusieurs choses à la fois. Par exemple en même temps que vous apprenez un cours, vous pouvez penser ce que vous allez manger pour le diner. Notre cerveau, de façon naturelle, a cette aptitude à traiter plusieurs flux de pensées à la fois. On parle de multitâche.

Dans le monde informatique, cette notion existe aussi et elle est assez fréquemment utilisée de nos jours. En même temps que nous ouvrons une page web d'actualités, nous pouvons télécharger des données, écouter de la musique, etc. On pourra toujours le faire avec une seule tâche mais c'est plus flexible et performant de le réaliser avec des programmes multitâches.

Tous les processeurs des ordinateurs modernes sont multitâches et il en est de même de la majorité des applications que nous utilisons.

Java implémente cette notion de multitâche (Thread). L'objectif de ce chapitre est d'étudier les bases du multitâche en Java, les librairies de l'API qui le décrivent et finalement par des exemples, montrer comment cela fonctionne.

La classe « ***java.lang.Thread*** » est la classe centrale de réalisation du multitâche en Java (Confer Java Doc pour l'explication des constructeurs et méthodes de cette classe).

I- Définitions et généralité sur les Thread

Un thread est une tâche s'exécutant dans un programme. La machine virtuelle Java permet à une application d'exécuter simultanément plusieurs tâches.

Chaque tâche a une priorité. Les threads avec une priorité supérieure sont exécutés de préférence aux threads ayant une priorité inférieure. Chaque thread peut ou non être marqué comme un démon (tâche s'exécution en background). Lorsque le code exécuté dans un thread crée un nouvel objet Thread, le nouveau thread a sa priorité initialement définie égale à la priorité du thread de création, et est un thread de démon si et seulement si le thread de création est un démon.

Lorsqu'une machine virtuelle Java démarre, il existe généralement un seul thread non-daemon (qui appelle généralement la méthode nommée principale d'une classe désignée). La machine virtuelle Java continue d'exécuter des threads jusqu'à ce que l'une des situations suivantes se produise:

- La méthode de sortie de la classe *Runtime* a été appelée et le gestionnaire de sécurité a autorisé l'opération de sortie à se dérouler.
- Tous les threads qui ne sont pas des threads Daemon sont morts, soit en revenant de l'appel à la méthode d'exécution, soit en lançant une exception qui se propage au-delà de la méthode d'exécution.

Il existe deux façons de créer un nouveau thread d'exécution. L'un est de déclarer une classe comme une sous-classe de Thread. Cette sous-classe doit remplacer la méthode d'exécution de la classe Thread. Une instance de la sous-classe peut ensuite être attribuée et démarrée. L'autre façon de créer un thread est de déclarer une classe qui implémente l'interface *Runnable*. Cette classe implémente ensuite la méthode *run()*. Une instance de la classe peut ensuite être attribuée, passée comme argument lors de la création de Thread et démarrée. Ces deux méthodes seront détaillées plus loin dans le cours.

Tous les threads ont un nom pour l'identification. Plusieurs thread peuvent avoir le même nom. Si un nom n'est pas spécifié lorsqu'un thread est créé, un nouveau nom est auto-généré.

Sauf indication contraire, passer un argument de type *null* sur un constructeur ou une méthode dans cette classe entraînera une exception de type *NullPointerException*.

II- Exécution d'un Thread

Le « thread » avec petit « t » est un processus s'exécutant dans un espace bien réservé dans la machine virtuelle Java tandis qu'un « Thread » avec un grand « T » est une instance de la classe « ***java.lang.Thread*** ». On peut confondre les deux termes suivant leur contexte d'utilisation. Pour démarrer un thread dans un JRE, on construit une instance de la classe Thread et on invoque sa méthode « ***start*** ».

```
Thread t = new Thread();
t.start();
```

Pour donner à un Thread une tâche à effectuer, il faut réécrire la méthode ***public void run()*** ou implémenter l'interface ***Runnable*** passer l'objet ***Runnable*** au constructeur de classe ***Thread***. Dans la plupart des cas c'est la seconde méthode qui est préférée car cela permet de séparer la tâche qu'effectue le thread de sa définition.

Pour un thread, la méthode ***run()*** est comparable à ce que la ***main()*** est pour une classe Java standard.

1- Hériter la classe Thread

En imaginant que l'on veuille calculer la valeur SHA (Secure Hash Algorithm) d'une liste de fichiers fournie en entrée, suivant la taille du fichier, le calcul du code SHA aura une vitesse proportionnelle.

Exemple 1

```
public class DigestThread extends Thread {

  private File input;

  public DigestThread(File input) {
   this.input = input;
  }

  public void run() {
    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read()) != -1) ;
      din.close();
      byte[] digest = sha.digest();
      StringBuffer result = new StringBuffer(input.toString());
      result.append(": ");
      for (int i = 0; i < digest.length; i++) {
        result.append(digest[i] + " ");
      }
      System.out.println(result);
    }
    catch (IOException e) {
      System.err.println(e);
    }
    catch (NoSuchAlgorithmException e) {
      System.err.println(e);
    }
  }
  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      File f = new File(args[i]);
      Thread t = new DigestThread(f);
      t.start();
    }

  }

}
```

Puisque la signature de la méthode ***run()*** est fixée, il sera impossible de la passer un argument ou qu'elle retourne une valeur. Par conséquent, nous allons utiliser différentes astuces pour passer un argument ou pour retourner une valeur à partir d'un thread. Par exemple, pour passer une valeur au thread, on peut le faire à partir du constructeur de la classe Thread.

Par contre, retourner des valeurs à partir des threads est un peu plus complexe à cause du mode fonctionnement asynchrone des threads. Nous le verrons plus loin dans ce chapitre.

2- Implémenter l'interface *Runnable*

L'autre méthode pour ne pas réécrire les méthodes de la classe Thread est de ne peux la faire hériter. On peut définir la tâche que l'on veut effectuer dans le thread avec la méthode ***run()*** de l'interface ***Runnable***.

En dehors des méthodes de cette interface, on peut définir d'autres méthodes avec les noms de notre choix à la seule condition qu'elles n'interfèrent pas dans le fonctionnement du thread. L'utilisation de ***Runnable*** permet aussi d'hériter d'une autre classe car Java interdit l'héritage multiple. Pour démarrer un thread qui exécute la tâche de l'interface ***Runnable***, on fait :

```
Thread t = new Thread(myRunnableObject);
t.start(  );
```

Exemple 2 :

```
public class DigestRunnable implements Runnable {

  private File input;

  public DigestRunnable(File input) {
   this.input = input;
  }

  public void run() {
    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read()) != -1) ;
      din.close();
      byte[] digest = sha.digest();
      StringBuffer result = new StringBuffer(input.toString());
      result.append(": ");
      for (int i = 0; i < digest.length; i++) {
        result.append(digest[i] + " ");
      }
      System.out.println(result);
    }
    catch (IOException e) {
      System.err.println(e);
    }
    catch (NoSuchAlgorithmException e) {
      System.err.println(e);
    }
```

```
  }

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      File f = new File(args[i]);
      DigestRunnable dr = new DigestRunnable(f);
      Thread t = new Thread(dr);
      t.start();
    }

  }

}
```

III- Retourner une information depuis un Thread

L'une des choses les plus difficiles pour un programmeur habitué à la programmation mono-thread est l'envoi de valeurs en retour lorsqu'ils travaillent sur des programmes multithreads. La récupération de valeur à la fin de l'exécution du thread est parmi les aspects les moins compris de la programmation multithread. Par exemple dans les deux derniers exemples, instinctivement on serait tenté d'utiliser une variable pour stocker le résultat du HASH et utilisé un accesseur pour le récupérer comme le montre les exemples 3 &4. L'exemple 3 calcule le HASH et l'exemple 4 l'affiche.

Exemple 3

```
public class ReturnDigest extends Thread {

  private File input;
  private byte[] digest;

  public ReturnDigest(File input) {
   this.input = input;
  }

  public void run() {
    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read()) != -1) ;
      din.close();
      digest = sha.digest();
    }
    catch (IOException e) {
      System.err.println(e);
    }
    catch (NoSuchAlgorithmException e) {
      System.err.println(e);
    }

  }

  public byte[] getDigest() {
    return digest;
  }

}
```

Exemple 4

```
public class ReturnDigestUserInterface {

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {

      // Calculate the digest
      File f = new File(args[i]);
      ReturnDigest dr = new ReturnDigest(f);
      dr.start();

      // Now print the result
      StringBuffer result = new StringBuffer(f.toString());
      result.append(": ");
      byte[] digest = dr.getDigest();
      for (int j = 0; j < digest.length; j++) {
        result.append(digest[j] + " ");
      }
      System.out.println(result);

    }

  }

}
```

L'exemple 3 associé à l'exemple 4 ne va pas marcher car la méthode va essayer de lire le contenu de la variable avant que le thread n'ait la chance d'être initialisé. Si c'était un programme mono thread, cela allait marcher.

1- Méthode par course de vitesse (Race condition)

Il consiste aussi à stocker le résultat dans une variable avec un accesseur ***getXXX***. Après exécution des threads, on récupère les résultats. Il faut avoir la chance que la seconde boucle finisse son exécution après que la première soit terminée, sinon on aura toujours une erreur.

Cela dépend du nombre de Thread, de la vitesse du CPU, du disque dur et de l'algorithme utilisé par la JVM.

Exemple 5

```
public static void main(String[] args) {

    ReturnDigest[] digests = new ReturnDigest[args.length];
    for (int i = 0; i < args.length; i++) {
    // Calculate the digest
    File f = new File(args[i]);
    digests[i] = new ReturnDigest(f);
    digests[i].start( );
    }
    for (int i = 0; i < args.length; i++) {
        // Now print the result
        StringBuffer result = new StringBuffer(args[i]);
        result.append(": ");
        byte[] digest = digests[i].getDigest( );
        for (int j = 0; j < digest.length; j++) {
        result.append(digest[j] + " ");
        }
        System.out.println(result);
    }

}
```

2- L'élection (Polling)

On procède toujours par deux boucles, mais la seconde vérifie que l'accesseur n'envoie plus une valeur témoin appelée « flag ».

Quand la valeur retournée par le thread est différent de « flag », cela signifie que le thread concerné a fini son exécution. On peut prendre pour flag la valeur nulle par exemple.

Exemple 6

```
public static void main(String[] args) {

    ReturnDigest[] digests = new ReturnDigest[args.length];
    for (int i = 0; i < args.length; i++) {
        // Calculate the digest
        File f = new File(args[i]);
        digests[i] = new ReturnDigest(f);
        digests[i].start();
    }
    for (int i = 0; i < args.length; i++) {
        while (true) {
            // Now print the result
            byte[] digest = digests[i].getDigest();
            if (digest != null) {
                StringBuffer result = new StringBuffer(args[i]);
                result.append(": ");
                for (int j = 0; j < digest.length; j++) {
                    result.append(digest[j] + " ");
                }
                System.out.println(result);
                break;
            }
        }
    }

}
```

3- Fonction de rappel (Callback)

Il existe une solution plus simple que faire une boucle infinie qui vérifie l'état de chaque thread. Il serait préférable que le thread informe la fonction main lorsqu'il aura fini. La fonction principale main peut aller en mode de veille (***sleep***) en attendant.

La fonction qui se charge d'informer la fonction principale est appelée fonction de rappel (***callback***).

Exemple 7

```
public class CallbackDigest implements Runnable {

  private File input;

  public CallbackDigest(File input) {
   this.input = input;
  }

  public void run() {
    try {
      FileInputStream in = new FileInputStream(input);
      MessageDigest sha = MessageDigest.getInstance("SHA");
      DigestInputStream din = new DigestInputStream(in, sha);
      int b;
      while ((b = din.read()) != -1) ;
      din.close();
      byte[] digest = sha.digest();
      CallbackDigestUserInterface.receiveDigest(digest, input.getName());
    }
    catch (IOException e) {
      System.err.println(e);
    }
    catch (NoSuchAlgorithmException e) {
      System.err.println(e);
    }

  }

}
```

Exemple 8

```
public class CallbackDigestUserInterface {

  public static void receiveDigest(byte[] digest, String name) {

    StringBuffer result = new StringBuffer(name);
    result.append(": ");
    for (int j = 0; j < digest.length; j++) {
      result.append(digest[j] + " ");
    }
    System.out.println(result);

  }

  public static void main(String[] args) {

    for (int i = 0; i < args.length; i++) {
      // Calculate the digest
      File f = new File(args[i]);
      CallbackDigest cb = new CallbackDigest(f);
      Thread t = new Thread(cb);
      t.start();
    }

  }

}
```

IV- Synchronisation

L'avantage des ressources partagées est que contrairement aux ressources personnelles, elles servent à beaucoup. On peut prendre l'exemple d'une bibliothèque personnelle et d'une autre publique qui prête le livre aux adhérents.

Aussi les ressources partagées ont beaucoup de désavantages. Dans une bibliothèque publique, il faut faire le rang pour sortir un livre, faire des réservations de livre, il est interdit d'écrire dans le livre.

Un thread est comme un adhérent d'une bibliothèque publique. Il passe des ressources partagées contrairement aux applications multiprocessus qui font une copie de chaque processus dont elles ont besoin.

Le problème avec les threads est que lorsque deux threads ont besoin de la même ressource au même moment, l'un doit attendre l'autre pour finir.

Pour assigner une ressource à un Thread jusqu'à ce qu'il finisse le traitement en cours, on utilise les blocs de synchronisation.

1- Synchroniser un bloc de code

Pour indiquer qu'un bloc de code d'instructions doit être exécuté ensemble, on utilise le mot clef « ***synchronized*** »

Exemple 9

```
String input = null;
synchronized (System.out) {
    System.out.print(input + ": ");
    for (int i = 0; i < digests.length; i++) {
        System.out.print(digests[i] + " ");
    }
    System.out.println();
}
```

Java ne donne pas une méthode pour arrêter un Thread qui utilise déjà une ressource mais peut protéger la ressource avec « ***synchronized*** ». La synchronisation peut être utilisée ailleurs que dans les threads.

2- Synchroniser une méthode

Au lieu de synchroniser un bloc d'une méthode, on peut décider de synchroniser toute la méthode. La signature de la méthode devient alors.

Exemple 10

```
public synchronized void myMethod(){
    //Some instructions
}
```

3- Alternatif à la synchronisation

La synchronisation n'est pas la seule méthode pour planifier les threads. Par exemple on peut utiliser des variables locales à un thread. Les variables locales n'ont pas de problème de synchronisation.

Les fonctions qui prennent des arguments de types primitifs n'ont pas aussi besoin d'être synchronisées car les types primitifs sont passés par valeur.

Les String sont « ***thread safe »*** car ils sont immuables, pour rendre un objet immuable, il faut le déclarer « ***private »***.

V- Les impasses (Deadlock)

On assiste à une situation d'impasse lorsqu'au moins deux threads ont besoin chacun de la ressource que possède l'autre. Les impasses peuvent être sporadiques et difficilement détectables

La façon la plus efficace pour empêcher les impasses est d'éviter les synchronisations inutiles. Il est mieux d'utiliser des objets immuables et des types primitifs. Il faut considérer les impasses comme un problème et architecturer son code autour de ça. Aussi quand vous remarquez qu'il y a des ressources partagées dans votre code, il faut vous assurer que les objets (classes) y ont accès dans le même ordre : Objet X avant Objet Y.

VI- Planification des threads

1- Priorité

La priorité est un nombre entre 1 et 10 (10 est la priorité la plus forte). En Java, quand plusieurs threads sont à l'état prêt, celui qui a la plus grande priorité est exécuté. Pour changer la priorité d'un thread, on utilise la méthode « ***setPriority()*** ».

En général, il faut éviter de donner à un thread une grande priorité car il pourrait empêcher les autres d'avoir un temps processeur : on dit qu'il les affame (Starvation).

2- Préemption

Chaque machine virtuelle dispose d'un ordonnanceur de threads qui détermine le thread à exécuter à un moment donné. Il existe deux types de planificateur de threads : préemptifs et coopératifs.
Un ordonnanceur de thread préemptif détermine quand un thread a épuisé sa part de Temps CPU, pause ce thread, puis passe le Temps CPU à un autre thread. L'ordonnanceur de threads coopératif attend que le thread en cours se mette en pause avant de passer le temps CPU à un autre thread. Une machine virtuelle qui utilise l'ordonnancement de thread coopératif est beaucoup plus susceptible d'affamer les threads qu'une machine virtuelle qui utilise l'ordonnancement de thread préemptif, car avec une priorité élevée, un thread non coopératif peut utiliser à lui tout seul tout le temps CPU

Toutes les machines virtuelles Java utilisent un ordonnanceur préemptif de thread entre priorité. C'est-à-dire, si un thread de priorité inférieure est en cours d'exécution lorsqu'un thread de priorité plus élevée est capable de s'exécuter, la machine virtuelle sera tôt ou tard (et probablement plus tôt) obligée de mettre en pause le thread de priorité inférieure pour permettre l'exécution du thread de priorité plus élevée. Le thread de priorité plus élevée préempte le thread de priorité inférieure.

La situation dans laquelle plusieurs threads de la même priorité peuvent être exécutés est plus délicate. Un planificateur de thread préemptif mettra parfois en pause l'un des threads pour permettre au suivant d'obtenir un certain temps CPU. Cependant, un planificateur de thread coopératif ne le fera pas. Il attendra que le thread en cours d'exécution renonce explicitement au temps CPU ou arrive à un point d'arrêt. Si le thread en cours d'exécution n'abandonne jamais le temps CPU et ne vient jamais à un point d'arrêt et si aucun thread de priorité plus élevée ne le préempte, tous les autres threads seront affamés. Ceci est une mauvaise chose. Par conséquent, il est important de s'assurer que tous vos threads se mettent périodiquement en pause afin que les autres threads aient l'opportunité de s'exécuter.

Il existe 10 façons pour un thread de s'arrêter en faveur d'autres threads ou indiquer qu'il est prêt à faire une pause.

- Il peut bloquer sur les E / S.

- Il peut bloquer sur un objet synchronisé.
- Il peut rendre la ressource CPU.
- Il peut être mis en veille.
- Il peut rejoindre un autre thread.
- Il peut attendre un objet.
- Il peut finir.
- Il peut être préempté par un thread prioritaire.
- Il peut être suspendu.
- Il peut s'arrêter.

Vous devriez vérifier que chaque méthode *run ()* que vous écrirez possède l'une de ces conditions et qu'elle se produira avec une fréquence raisonnable. Les deux dernières possibilités sont obsolètes car elles ont le potentiel de laisser des objets dans des états incohérents, alors focalisons nous sur les huit autres façons de rendre un thread coopératif de la machine virtuelle.

a- Le blocage

Le blocage se produit à chaque fois qu'un thread doit s'arrêter et attendre une ressource qu'il n'a pas. Le plus fréquent dans un programme réseau, un thread va volontairement abandonner le contrôle du CPU lorsqu'il est bloqué sur un E / S. Étant donné que les CPU sont beaucoup plus rapides que les réseaux et les disques, un programme réseau bloque souvent tout en attendant que les données arrivent du réseau ou soient envoyées au réseau. Même si cela ne dure que quelques millisecondes, c'est largement suffisant pour d'autres threads de s'exécuter.

Les threads peuvent également bloquer lorsqu'ils ont besoin d'une méthode ou d'un bloc synchronisé. Si le thread ne possède pas déjà le verrou pour l'objet en cours de synchronisation et qu'un autre thread possède ce verrouillage, le thread se mettra en pause jusqu'à ce que le verrou soit relâché. Si le verrou n'est jamais relâché, le thread est indéfiniment arrêté.

Ni le blocage dû aux E/S, ni le blocage sur un verrou ne libéreront les verrous que le thread possède déjà. Pour les blocages dû aux d'E/S, ce n'est pas grave car, finalement, les E /S seront débloqués et le thread continuera ou une exception de type *IOException* sera lancée et le thread sortira ainsi du bloc ou de la méthode synchronisée et libérera à son tour ses verrous.

Cependant, un thread bloquant sur un verrou qu'il ne possède pas n'abandonnera jamais ses propres verrous. Si un thread attend un verrou qu'un second thread possède et que le second thread attend un verrou dont le premier thread possède, on obtient une impasse.

b- Rendre la ressource (yielding)

La deuxième façon pour un thread d'abandonner le temps CPU est de le céder explicitement. Un thread le fait en invoquant la méthode statique *Thread.yield ( ):*

public static void yield ( )

Ceci indique à la machine virtuelle qu'il peut exécuter un autre thread prêt à être exécuter. Certaines machines virtuelles, en particulier sur les systèmes d'exploitation en temps réel, peuvent ignorer cet indice.

Avant de céder, un thread devrait s'assurer que celui-ci ou son objet *Runnable* associé est dans un état cohérent qui peut être utilisé par d'autres objets. Le yielding ne libère aucun verrouille du thread. Par conséquent, idéalement, un thread ne devrait pas être synchronisé sur quelque chose lors de son yielding. Si les seuls autres threads en attente d'exécution lorsqu'un vous avez fait le yielding sont bloqués car ils ont besoin des ressources synchronisées que le thread en état de yielding possède, alors les autres threads ne pourront pas fonctionner. A la place, le temps CPU revient au seul thread qui peut continuer qui peut s'exécuter : celui qui vient de procéder au yielding, donc l'objectif du yielding n'est pas atteint.

On peut facilement procéder au yielding. Par exemple si la méthode *run ( )* du thread se est composé d'une boucle infinie, il suffit d'appeler *Thread.yield ( )* à la fin de la boucle.

```
1  public void run () {
2      While (true)
3      {
4          // Tâche à exécuter par le thread
5          Thread.yield ();
6      }
7  }
```

Tant que la méthode *run ()* n'est pas synchronisée (normalement, une très mauvaise idée de toute façon), cela devrait donner à d'autres thread de la même priorité l'opportunité de

s'exécuter. Si chaque itération de la boucle prend beaucoup de temps, vous voudrez peut-être faire plus d'appels *Thread.yield ()* dans le reste du code. Cela devrait avoir un effet minimal dans le cas où le yielding n'est pas nécessaire.

c- La mise en veille

Le sommeil est une forme de yielding plus puissante. Alors que le yielding indique seulement qu'un thread est prêt à faire une pause et que d'autres threads à priorité égale peuvent être exécutés, un thread qui se met en veille se mettra en pause si un autre thread est prêt à fonctionner ou non. Cela peut donner non seulement à d'autres threads de la même priorité, mais aussi des threads de priorités inférieures, une opportunité de s'exécuter. Cependant, un thread en veille ne libère les verrous qu'il possède. Par conséquent, d'autres threads nécessitant les mêmes verrous seront bloqués même si le CPU est disponible. Par conséquent, vous devriez essayer d'éviter de mettre en veille les threads dans une méthode ou un bloc synchronisé.

Parfois, il est utile de mettre en veille un thread non pas seulement pour céder du temps processeur aux autres threads. Mettre un thread en veille pendant une période de temps spécifiée vous permet d'écrire un code qui s'exécute à intervalle de temps régulier : une fois par seconde, chaque minute, toutes les dix minutes, etc. Par exemple, si vous étiez en train d'écrire un programme de surveillance réseau qui récupère une page à partir d'un serveur Web toutes les cinq minutes et envoie un courrier électronique au webmaster si le serveur a crashé, vous pouvez l'implémenter comme un thread qui va en veille pendant cinq minutes entre l'exécution de cette tâche.

Un thread va en veille en invoquant l'une des deux méthodes surdéfines *static Thread.sleep ( )*. Le premier prend le nombre de millisecondes de veille comme argument. La seconde prend le nombre de millisecondes et le nombre de nanosecondes:

```
1   public static void sleep(long milliseconds)
2   throws InterruptedException
3   public static void sleep(long milliseconds, int nanoseconds)
4   throws InterruptedException
```

Alors que la plupart des horloges modernes de l'ordinateur ont au moins une précision de près de millisecondes, la précision de la nanoseconde est plus rare. Il n'y a aucune garantie sur

n'importe quelle machine virtuelle particulière que vous pouvez effectivement mettre en veille en nanoseconde ou même en milliseconde. Si le matériel local ne peut pas supporter ce niveau de précision, le temps de veille est simplement arrondi à la valeur la plus proche qui peut être mesurée. Par exemple:

Exemple 10

```
1  public void run( ) {
2      while (true) {
3          if (!getPage("http://itupendo.net/javafaq/")) {
4              mailError("dcarrena@itupendo.net");
5          }
6          try {
7              Thread.sleep(300000); // 300,000 milliseconds == 5 minutes
8          }
9          catch (InterruptedException e) {
10             break;
11         }
12     }
13 }
```

Il n'est pas absolument garanti qu'un thread passera aussi longtemps qu'il le souhaite en veille. À l'occasion, le thread peut ne passer plus de temps que prévu, simplement parce que la VM est occupée à faire d'autres choses. Il est également possible qu'un autre thread fasse quelque chose pour réveiller le thread avant la fin de la durée demandée. Généralement, cela se fait en invoquant la méthode d'interruption de veille du thread :

```
1  public void interrupt( );
```

C'est l'un de ces cas où la distinction entre le thread et l'objet Thread est importante. Juste parce que le thread est en veille, cela ne signifie pas que d'autres threads en cours d'exécution ne peuvent pas fonctionner avec l'objet Thread correspondant à travers ses méthodes et ses champs. En particulier, un autre thread peut invoquer la méthode *interupt ( )* de l'objet Thread en veille, que le thread en veille connaît comme étant une exception *InterruptedExeception*. À partir de ce moment, il est revient de la veille et s'exécute comme d'habitude, au moins jusqu'à ce qu'il parte en veille de nouveau. Dans l'exemple précédent, une exception de type *InterruptedException* est utilisée pour mettre fin authread qui autrement serait exécuté pour toujours. Lorsque l'exception *InterruptedException* est lancée, la boucle infinie est cassée, la méthode *run ()* se termine et le thread disparaît. Le thread de l'interface utilisateur peut invoquer la méthode d'interruption

interrupt ( ) de ce thread lorsque l'utilisateur sélectionne *Quitter* dans un menu ou indique qu'il souhaite que le programme se termine.

d- La jointure de thread

Il n'est pas rare qu'un thread ait besoin du résultat d'un autre thread. Par exemple, un navigateur Web chargeant une page HTML dans un thread peut engendrer un thread distinct pour récupérer toutes les images intégrées dans la page. Si les éléments IMG n'ont pas d'attributs HEIGHT et WIDTH, le thread principal peut attendre que toutes les images soient chargées avant de pouvoir terminer l'affichage la page. Java fournit trois méthodes *join ( )* pour permettre à un thread d'attendre qu'un autre thread se termine avant de continuer.

```
1  public final void join( ) throws InterruptedException
2  public final void join(long milliseconds) throws InterruptedException
3  public final void join(long milliseconds, int nanoseconds) throws InterruptedException
```

La première variante attend indéfiniment que le thread joigne pour finir. Les deux dernières variantes attendent le temps spécifié, après quoi elles continuent même si le thread devant joindre n'a pas fini. Comme pour la méthode *sleep ( ),* la précision en nanoseconde n'est pas garantie.

Le thread joignant - c'est-à-dire celui qui invoque la méthode *join ( )* - attend le thread joint - c'est-à-dire celui dont la méthode *join ( )* est invoquée-pour terminer. Par exemple, considérez ce fragment de code. Nous voulons trouver le minimum, le maximum et la médiane d'un tableau aléatoire de doubles. Il est plus rapide de le faire avec un tableau trié. Nous générons un nouveau thread pour trier le tableau, puis rejoignons le thread initial pour attendre ses résultats. Ce n'est que lorsque c'est fait que nous lisons les valeurs souhaitées.

Exemple 11

```
1   double[] array = new double[10000];
2   for (int i = 0; i < array.length; i++) {
3       array[i] = Math.random( );
4   }
5   SortThread t = new SortThread(array);
6   t.start( );
7   try {
8       t.join( );
9       System.out.println("Minimum: " + array[0]);
10      System.out.println("Median: " + array[array.length/2]);
11      System.out.println("Maximum: " + array[array.length-1]);
12  }
13  catch (InterruptedException e) {
14  }
```

Les premières lignes 1 à 4 exécutent le remplissage du tableau avec des nombres aléatoires. Ensuite, la ligne 5 crée un nouveau SortThread. La ligne 6 démarre le thread qui va trier le tableau. Avant de pouvoir trouver le minimum, la médiane et le maximum du tableau, nous devons attendre que le thread de tri soit terminé. Par conséquent, la ligne 8 rejoint le thread actuel au thread de tri. A ce stade, le thread exécutant ces lignes de code s'arrête à cette étape. Il attend que le thread de tri soit terminé. Le minimum, la médiane et le maximum ne sont pas récupérés dans les lignes 9 à 10 jusqu'à ce que le thread de tri ait fini de s'exécuter et soit mort. Notez qu'en aucun cas il n'y a de référence au thread qui fait une pause. Ce n'est pas l'objet Thread sur lequel la méthode *join ( )* est invoquée. Il n'est pas passé comme argument à cette méthode. Il n'existe implicitement que comme étant le thread actuel. Si cela se trouve dans le flux normal de contrôle de la méthode *main ( )* du programme, il se peut qu'il n'y ait aucune variable de thread quelconque qui pointe vers ce thread.

Un thread qui est joint à un autre thread peut être interrompu juste comme un thread en veille si un autre thread invoque sa méthode *interrupt ( )*. Le thread reconnaît cette invocation en tant *InterruptedException*. À partir de ce moment, il s'exécute normalement, à partir du bloc *catch* qui gère l'exception. Dans l'exemple précédent, si le thread est interrompu, il ignore le calcul du minimum, de la médiane et du maximum car ils ne seront pas disponibles si le fil de tri a été interrompu avant qu'il ne puisse finir.

On peut utiliser la méthode *join ()* pour résoudre le problème évoqué lorsqu'on utilise la technique de *race condition* pour retourner une valeur à la fin de l'exécution d'un thread. Le code suivant propose une réécriture de notre exemple sur le *race condition.*

Exemple 12

```
1  import java.io.File;
2  public class JoinDigestUserInterface {
3      public static void main(String[] args) {
4          ReturnDigest[] digestThreads = new ReturnDigest[args.length];
5          for (int i = 0; i < args.length; i++) {
6              // Calculate the digest
7              File f = new File(args[i]);
8              digestThreads[i] = new ReturnDigest(f);
9              digestThreads[i].start( );
10         }
11         for (int i = 0; i < args.length; i++) {
12             try {
13                 digestThreads[i].join( );
14                 // Now print the result
15                 StringBuffer result = new StringBuffer(args[i]);
16                 result.append(": ");
17                 byte[] digest = digestThreads[i].getDigest( );
18                 for (int j = 0; j < digest.length; j++) {
19                 result.append(digest[j] + " ");
20                 }
21                 System.out.println(result);
22             }
23             catch (InterruptedException e) {
24                 System.err.println("Thread Interrupted before completion");
25             }
26         }
27     }
28 }
```

Vous remarquez ici que tant le thread correspondant au résultat que l'on veut afficher n'est pas terminé (ligne 13 avec la méthode *join ()*) les résultats ne seront pas affichés.

e- Attendre un objet

Un thread peut attendre et en même temps verrouillé un objet. Dans son attente, il lève le verrou sur l'objet et s'arrête jusqu'à ce qu'il soit notifié par un autre thread que l'objet est à nouveau libre. Un autre thread peut modifier l'objet d'une façon ou d'une autre, avertit le thread en attente lorsqu'il finit de l'utiliser. L'attente est utilisée pour interrompre l'exécution jusqu'à ce qu'un objet ou une ressource atteigne un certain état. La jointure par contre est utilisée pour interrompre l'exécution jusqu'à ce qu'un autre thread finisse.

Attendre sur un objet est l'une des méthodes les moins connues pour en pause un Thread. Il n'y a aucune méthode propre à la classe Thread qui permet d'attendre un objet. Pour attendre un objet, le thread doit obtenir le verrou de cet objet à l'aide de la synchronisation puis invoquer l'une des méthodes de classe sur-définies *wait()* :

```
1  public final void wait( ) throws InterruptedException
2  public final void wait(long milliseconds) throws InterruptedException
3  public final void wait(long milliseconds, int nanoseconds) throws InterruptedException
```

Ces méthodes ne sont pas dans la classe *Thread*. Plutôt, ils sont dans la classe *java.lang.Object*. Par conséquent, ils peuvent être invoqués sur n'importe quel objet de toute classe. Lorsqu'une de ces méthodes est invoquée, le thread qui l'invoque libère son verrou sur l'objet (mais pas les verrous qu'il peut posséder sur d'autres objets) et va en veille. Il reste en veille jusqu'à ce que l'une des trois évènements se produise:

- Le délai expire.
- Le thread est interrompu.
- Une notification sur l'objet arrive.

L'expiration du délai et l'interruption du thread fonctionnent de la même manière que dans le cas de la jointure ou de la mise en veille. La notification se produit lorsqu'un autre thread invoque la méthode *notify ( )* ou *notifyAll ( )* sur l'objet sur lequel le thread est en attente. Ces deux méthodes sont dans la classe *java.lang.Object*:

```
1  public final void notify( )
2  public final void notifyAll( )
```

Ceux-ci doivent être invoqués sur l'objet sur lequel le thread était en attente, pas généralement sur le thread lui-même. Avant de notifier un objet, un thread doit d'abord obtenir le verrou sur l'objet à l'aide d'une méthode ou d'un bloc synchronisé. La méthode *notify ( )* sélectionne un thread au hasard dans la liste des threads en attente de l'objet et le réveille. La méthode *notifyAll ( )* réveille tous les threads en attente sur l'objet donné.

Une fois qu'un thread en attente est notifié, il essaie de retrouver le verrou de l'objet qu'il attendait. S'il réussit, son exécution reprend par là où elle était arrêtée après l'invocation de la

mise en attente. S'il échoue, il bloque l'objet jusqu'à ce que son verrou soit disponible, puis l'exécution reprendra avec l'instruction immédiatement après l'invocation de la mise en attente.

f- Préemption basée sur la priorité

Étant donné que les threads sont préemptifs entre les priorités, vous n'avez pas à vous soucier de l'exécution des threads prioritaires. Un thread prioritaire préempte les threads de priorité inférieure lorsqu'il est prêt à s'exécuter. Cependant, lorsque le thread prioritaire finit d'être exécuté, il ne passe généralement pas la main au même thread de priorité inférieure. Au lieu de cela, la plupart des machines virtuelles non-temps-réel utilisent un planificateur en *round-robin* pour que le thread de priorité inférieure qui ne soit pas exécuté depuis le plus longtemps possible soit exécuté ensuite.

Si vous préférez éviter un *yielding* explicite, vous pouvez utiliser un thread prioritaire pour forcer les threads à priorité inférieure à passer le temps CPU à un autre dans la liste. Essentiellement, vous pouvez utiliser un planificateur de thread basé sur la priorité de votre propre conception pour rendre tous vos threads préemptifs. L'astuce consiste à exécuter un thread de priorité élevée qui se met en veille périodiquement, disons toutes les 100 millisecondes. Cela va diviser les threads de priorité inférieure en tranches de temps de 100 millisecondes. Il n'est pas nécessaire que le thread qui fait le fractionnement ait des informations sur les threads est préemptés. L'exemple suivant démontre avec une classe *TimeSlicer* qui vous permet de garantir la préemption des threads de priorité inférieure à une valeur fixe chaque milliseconde.

Exemple 13

```
1  public class TimeSlicer extends Thread {
2      private long timeslice;
3      public TimeSlicer(long milliseconds, int priority) {
4          this.timeslice = milliseconds;
5          this.setPriority(priority);
6          // If this is the last thread left, it should not
7          // stop the VM from exiting
8          this.setDaemon(true);
9      }
10     // Use maximum priority
11     public TimeSlicer(long milliseconds) {
12         this(milliseconds, 10);
13     }
14     // Use maximum priority and 100ms timeslices
15     public TimeSlicer( ) {
16         this(100, 10);
17     }
18     public void run( ) {
19         while (true) {
20             try {
21                 Thread.sleep(timeslice);
22             }
23             catch (InterruptedException e) {
24             }
25         }
26     }
27 }
28
```

g- Fin d'un thread

La dernière façon par laquelle un thread peut céder le contrôle du CPU de manière ordonnée est lorsqu'il termine. Lorsque la méthode *run ( )* retourne, le thread meurt et les autres threads peuvent prendre le temps CPU.

Sinon, si votre méthode *run ( )* est vraiment si simple qu'elle finit toujours assez rapidement sans blocage, vous devez vous poser la question si vous devez le gérer avec un thread. L'utilisation des threads a une conséquence sur la machine virtuelle puis cela nécessite une planification et la gestion des différents états des threads. Si un thread se termine dans une petite fraction de seconde, il est probable qu'il finirait encore plus rapidement si vous procédez par appel simple de méthode plutôt qu'un thread distinct.

VII- Thread Pool

L'ajout de plusieurs threads à un programme améliore considérablement les performances, en particulier pour les programmes liés à l'E/S, tels que la plupart des programmes réseau. Cependant, les threads ont des conséquences sur la machines les exécutant. Démarrer un thread et le supprimer lorsqu'il finit sont exécution représente une quantité remarquable de travail pour la machine virtuelle, surtout si un programme génère des milliers de threads. Même si les threads finissent rapidement, cela peut surcharger la VM. Enfin, et surtout, bien que les threads aident à utiliser plus efficacement les ressources limitées d'un ordinateur, il n'y a encore qu'une quantité finie de ressources. Une fois que vous avez généré suffisamment de threads pour utiliser tous les temps d'inactivité disponibles de l'ordinateur, générer plus de threads gaspille le *MIPS* (*Million Instruction Per Second)* et la mémoire sur la gestion des threads.

Heureusement, vous pouvez obtenir le meilleur des deux mondes en réutilisant les threads. Vous ne pouvez pas redémarrer un thread une fois terminé, mais vous pouvez concevoir vos threads afin qu'ils ne terminent pas dès qu'ils ont fini d'exécuter une tâche. En lieu et place , vous mettez toutes les tâches que vous devez effectuer dans une file d'attente ou une autre structure de données et faites en sorte que chaque thread récupère une nouvelle tâche à partir de la file d'attente lorsqu'il est terminé sa tâche précédente. C'est ce qu'on appelle la poule (réservoir) de thread (*Thread Pool*), et la structure de données dans laquelle les tâches sont conservées s'appelle le *pool*.

La manière la plus simple de mettre en œuvre un *pool* de threads est d'utiliser un nombre fixe de threads configuré lorsque le *pool* est créé. Lorsque *le pool* est vide, chaque thread dans le *pool*. Lorsqu'une tâche est ajoutée au *pool*, tous les threads en attente sont notifiés. Lorsqu'un fil finit sa tâche assignée, il remonte au *pool* pour une nouvelle tâche. S'il n'en existe pas, il attend jusqu'à ce qu'une nouvelle tâche soit ajoutée au *pool*.

Une alternative est de mettre les threads eux-mêmes dans le *pool* et d'avoir le thread principal qui retire les threads du *pool* et leur attribue des tâches. Si aucun thread n'est dans le *pool* quand une tâche à exécuter venait à exister, le programme principal peut engendrer un nouveau thread. Lorsqu'un thread termine une tâche, il retourne dans le *pool*.

Il existe de nombreuses structures de données que vous pouvez utiliser pour faire un *pool*, bien qu'une file d'attente (*queue)* soit probablement la plus raisonnable pour que les tâches soient exécutées *FIFO*. Quelle que soit la structure de données que vous utilisez pour implémenter le *pool*, vous devez être extrêmement prudent quant à la synchronisation, car de nombreux threads interagiront étroitement entre eux en même temps. Le moyen le plus simple d'éviter les problèmes consiste à utiliser un *java.util.Vector* (qui est entièrement synchronisé) ou une collection synchronisée de l'API Java Collections.

Par exemple, supposons que vous souhaitez compresser (*gzip)* chaque fichier dans le répertoire actuel en utilisant un *java.util.zip.GZIPOutputStream*. D'une part, il s'agit d'une opération lourde d'E/S car tous les fichiers doivent être lus et écrits. D'autre part, la compression de données est une opération très intensive en CPU, donc vous ne voulez pas avoir beaucoup de threads s'exécutant en même temps. Nous pouvons alors utiliser le *thread pool*. Chaque thread de client compressera des fichiers tandis que le programme principal déterminera les fichiers à compresser. Dans cet exemple, le programme principal est susceptible de d'aller plus vite que les threads clients de manière significative car tout ce qu'il doit faire est de répertorier les fichiers du répertoire. Par conséquent, il n'est pas question de remplir le *pool* d'abord, avant de démarrer les threads clients. Cependant, pour rendre cet exemple aussi général que possible, nous allons permettre au programme principal de s'exécuter en parallèle avec les threads de fermeture.

La classe *GZipThread* contient un champ privé appelé *pool* contenant une référence au *pool*. Ici, ce champ est déclaré avec le type de liste, mais il est toujours accessible en *FIFO*. La méthode *run ( )* supprime les objets File du *pool* et les compresse. Si le *pool* est vide lorsque le thread est prêt, le fil attend sur l'objet *pool*.

Exemple 14

```
1  import java.io.BufferedInputStream;
2  import java.io.BufferedOutputStream;
3  import java.io.File;
4  import java.io.FileInputStream;
5  import java.io.FileOutputStream;
6  import java.io.IOException;
7  import java.io.InputStream;
8  import java.io.OutputStream;
9  import java.util.List;
10 import java.util.zip.GZIPOutputStream;
11
12 public class GZipThread extends Thread {
13     @SuppressWarnings("rawtypes")
14     private List pool;
15     private static int filesCompressed = 0;
16
17     @SuppressWarnings("rawtypes")
18     public GZipThread(List pool) {
19         this.pool = pool;
20     }
21
22     private static synchronized void incrementFilesCompressed() {
23         filesCompressed++;
24     }
25
26     public void run() {
27         while (filesCompressed != GZipAllFiles.getNumberOfFilesToBeCompressed()) {
28             File input = null;
29             synchronized (pool) {
30                 while (pool.isEmpty()) {
31                     if (filesCompressed == GZipAllFiles
32                             .getNumberOfFilesToBeCompressed())
33                         return;
34                     try {
35                         pool.wait();
36                     } catch (InterruptedException e) {
```

```
37                  }
38              }
39              input = (File) pool.remove(pool.size() - 1);
40          }
41          // don't compress an already compressed file
42          if (!input.getName().endsWith(".gz")) {
43              try {
44                  InputStream in = new FileInputStream(input);
45                  in = new BufferedInputStream(in);
46                  File output = new File(input.getParent(), input.getName()
47                          + ".gz");
48                  if (!output.exists()) { // Don't overwrite an existing file
49                      OutputStream out = new FileOutputStream(output);
50                      out = new GZIPOutputStream(out);
51                      out = new BufferedOutputStream(out);
52                      int b;
53                      while ((b = in.read()) != -1)
54                          out.write(b);
55                      out.flush();
56                      out.close();
57                      incrementFilesCompressed();
58                      in.close();
59                  }
60              } catch (IOException e) {
61                  System.err.println(e);
62              }
63          } // end if
64      } // end while
65  } // end run
66  } // end ZipThread
```

GZipAllFiles est le programme principal. Il construit le pool avec un objet de type *Vector*, transmet ce *pool* à quatre objets *GZipThread*, il commence ensuite tous les quatre threads, puis va rechercher tous les fichiers du répertoire cible. Ces fichiers sont ajoutés au *pool* pour un éventuel traitement par l'un des quatre threads.

Exemple 15

```
1  import java.io.*;
2  import java.util.*;
3
4  public class GZipAllFiles {
5      public final static int THREAD_COUNT = 4;
6      private static int filesToBeCompressed = -1;
7
8      @SuppressWarnings("unchecked")
9      public static void main(String[] args) {
10         @SuppressWarnings("rawtypes")
11         Vector pool = new Vector();
12         GZipThread[] threads = new GZipThread[THREAD_COUNT];
13         for (int i = 0; i < threads.length; i++) {
14             threads[i] = new GZipThread(pool);
15             threads[i].start();
16         }
17         int totalFiles = 0;
18         for (int i = 0; i < args.length; i++) {
19             File f = new File(args[i]);
20             if (f.exists()) {
21                 if (f.isDirectory()) {
22                     File[] files = f.listFiles();
23                     for (int j = 0; j < files.length; j++) {
24                         if (!files[j].isDirectory()) { // don't recurse
25                                                        // directories
26                             totalFiles++;
27                             synchronized (pool) {
28                                 pool.add(files[j]);
29                                 pool.notifyAll();
30                             }
31                         }
32                     }
33                 } else {
34                     totalFiles++;
35                     synchronized (pool) {
36                         pool.add(0, f);
```

```
37                          pool.notifyAll();
38                      }
39                  }
40              } // end if
41          } // end for
42          filesToBeCompressed = totalFiles;
43          // make sure that any waiting thread knows that no
44          // more files will be added to the pool
45          for (int i = 0; i < threads.length; i++) {
46              threads[i].interrupt();
47          }
48      }
49
50      public static int getNumberOfFilesToBeCompressed() {
51          return filesToBeCompressed;
52      }
53  }
```

Conclusion

Le multitâche est incontournable dans le monde de la programmation aujourd'hui. Il n'existe quasi plus de programme informatique mono-thread. Il existe des API qui permettent de rendre transparent cette partie de la programmation pour les amateurs. La maîtrise du fonctionnement des threads permet de mieux les adapter à nos besoins dans nos projet d'application.

Référence :

https://docs.oracle.com/javase/8/docs/api/

Harold E.R. - Java Network Programming, 4th Edition

Collins W. - Data Structures and the Java Collections Framework, 3rd Edition