Programmation avancée et répartie en Java : les processus légers

Frédéric Gava

L.A.C.L
Laboratoire d'**A**lgorithmique, **C**omplexité et **L**ogique

Cours de M1 MIAGE
(d'après les notes de cours de Fabrice Mourlin)

Plan

Plan

Déroulement du cours

Problématique (1)

Introduction

Comment faire réaliser plusieurs tâches en même temps ? En confiant ces tâches à des "processus" différents :

- Concurrence/parallèle (mémoire partagée) :
  - sur un même processeur (ordonnacement)
  - multi-coeurs, multi-processeur, GPU, *etc.*
- Distribué/Répartie (plusieurs machines)
- Un mixe des 2 (hybride)
- Mobile/agents (calcules qui se déplacent)

En Java

Java dispose d'un mécanisme de "processsus légers" (threads) qui s'exécutent en parallèle au sein d'une même JVM (concurrence). Parfois sur plusieurs coeurs/processeurs. Il existe aussi des bibliothèques standards (ou non) pour le calcul réparti/mobiles/etc.

Problématique (2)

Dans ce cours

- La notion de Thread
- Création de Threads, gestion du code et des données
- Cycle de vie d'un Thread et contrôles de l'exécution

Par la suite

- Accès concurrents
- Appels distants
- Échanges de messages (valeurs) : réseau et MPI
- Codes natif, références faibles, Programmation dynamique
- E/S et accès SBGD
- Sécurité et internationalisation

L'environnement d'exécution en Java

L'environnement d'exécution d'une JVM est disponible dans la JVM elle-même sous la forme d'un objet de type java.lang.Runtime. Il n'existe qu'un seul exemplaire d'un tel objet (Singleton) ; impossible de créer un objet de cette classe.

L'instance unique peut être retrouvée par appel à la méthode statique Runtime getRuntime();

De nombreuses méthodes sont disponibles dans la classe Runtime. Entre autres celles permettant de demander au système hôte de créer un processus concurrent (l'exécution d'un programme en dehors de la JVM elle-même) : c'est la famille des méthodes

Process exec(...);

Les processus en Java

les processus dont l'exécution (externe) a été commandée par une JVM sont représentés dans celle-ci sous la forme d'objet java.lang.Process. Ces objets permettent de

- communiquer avec les processus externes correspondants
- se synchroniser avec leur exécution

On peut obtenir les flux d'entrée/sorties du processus externe :

```
InputStream getErrorStream();
InputStream getOutputStream();
OutputStream getIntputStream();
```

Une entrée du processus correspond à une sortie depuis la JVM (et vice-versa). Il est possible de se synchroniser avec l'exécution du processus externe avec les méthodes :

```
int exitValue();
int waitFor();
```

Qui renvoient la valeur de terminaison du processus externe, en mode bloquant (waitFor()) et nonbloquant (exitValue()).

L'ordonnanceur (1)

Comment un OS permet-il d'avoir plusieurs tâches qui s'exécutent en même temps ? Même s'il ne dispose que d'un seul processeur pour réellement exécuter une tâche (processus), il est capable de donner une illusion de parallélisme : p. ex., pendant qu'il pilote une imprimante, il continue à dialoguer avec l'utilisateur.

Chaque processus exécute un certain code et dispose de ses données propres. L'illusion de l'exécution en parallèle est obtenue en n'exécutant une partie de code de chaque processus que pendant un laps de temps très court. Si à chaque tâche n'est attribuée, à son tour, qu'une partie de son code, on aura l'impression que plusieurs programmes s'exécutent en parrallèle.

L'ordonnanceur (2)

L'ordonnanceura un rôle essentiel : c'est lui qui décide quel sera le processus actif à un moment donné et pour combien de temps.

Il existe différents algorithmes d'ordonnancement : p. ex. le "time-slicing" alloue des tranches de temps pour chaque processus. Dans certain OS, un processus actif garde "la main" jusqu'à ce qu'il se bloque sur un appel système (p. ex. une opération d'E/S). Ces algorithmes sont complexes car ils tiennent compte de facteurs supplémentaires comme des priorités entre processus.

On trouve différent ordonnanceurs : celui de l'OS (ou de l'hyperviseur) qui gère les processus (programmes) et donc la JVM. Celui de la JVM pour les threads. Le processeur, par hyper threading, peut modifier l'ordre d'exécution des instructions. Quid des multi-coeurs ? Les ordonnanceurs modernes (JVM, Linux) sont capables d'utiliser les différents processeurs d'un multi-coeurs. Impossible de faire des hypothèses sur leurs comportements.

Les threads

Dans un processus, il existe des tâches (threads, processus "légers") qui partagent les mêmes données. Une fois créée cette tâche va disposer d'une pile d'exécution qui lui est propre et partager des codes et des données des classes. La JVM pourra soit associer un thread Java à un thread système soit utiliser son propre algorithme d'ordonnancement. Le programmeur ne doit donc pas faire d'hypothèses sur le comportement de ordonnanceur.

Voici un exemple de mauvaise conception :

```
Image img = getToolkit().getImage(urlImage) ;//dans un composant AWT
// méthode asynchrone: le chargement est réalisé par une tâche de fond
while(−1 == img.getHeight(this)) { /* rien : polling */ }
// tant que l'image n'est pas chargée sa hauteur est −1
```

- peut étouffer le CPU à case d'une boucle vide
- peut se bloquer si arrêt à cause d'un appel système

Création (1)

La classe Java Thread permet d'exécuter une tâche. Une instance de Thread va disposer de caractéristiques propres comme un nom permettant de l'identifier ou une priorité. Le code exécuté par le Thread est fourni par une autre classe qui réalise le contrat d'interface Runnable au travers de sa méthode run() :

```
import java.io.* ;
public class Compteur implements Runnable {
 public final int max ;
 public Compteur( int max){this.max = max;}
 public void run () {
  for(int ix = 0 ; ix < max ; ix++) {
  try {
   File temp = File.createTempFile("cpt","");
   System.out.println("#"+this+":"+temp);
   temp.delete() ;
   } catch (IOException exc){/*test */}
...
```

Ceci nous permet la définition d'une tâche comme :

```
Thread tache = new Thread(new Compteur(33)) ;
```

Création (2)

Ici il s'agit d'une tâche de démonstration sans portée pratique (la boucle crée des fichiers temporaires), mais le fait de faire un appel système lourd dans la boucle va probablement mettre en lumière le comportement de l'ordonnanceur.

Exercice : Recopiez et testez le code ! Écrivez une code qui ajoute à l'infinie des nouveaux threads dans un ArrayList avec un code runnable "vide". Que se passe t'il ?

Déroulement du cours

Début et fin de vie

Une fois créé le Thread est prêt à être activé : ceci se fait par l'invocation de la méthode start() sur l'instance. A partir de ce moment le code du run() est pris en charge par l'ordonnanceur. En fonction des décisions de l'ordonnanceur le Thread passe par une série d'états : actif (le code tourne) ou éligible (le code ne tourne pas, mais l'ordonnanceur peut le réactiver à tout moment).

Quand le code du run() est terminé, le Thread devient un "Zombie", l'instance du Thread existe jusqu'à ce qu'elle soit éventuellement récupérée par le garbage-collector. Mais on ne peut plus lui demander de re-exécuter la tâche (IllegalThreadStateException).

Vivant ?

Pour tester si un Thread est "vivant" : **if** tache.isAlive() { ...}

Cette méthode indique que start() a été exécutée et que l'instance cible n'est pas devenue "zombie". Cette méthode n'indique pas que le Thread tâche est en état actif car :

- dans le cas d'un mono-processeur/coeur, c'est le Thread demandeur qui exécute le code qui est actif.
- sinon, le temps de récupérer le booléen d'un (imaginaire) isActif(), le thread pourrait finalement ne plus l'être à cause de l'ordonnanceur ; cela n'aurait donc pas de sens.

Exercice :Reprendre le Thread Compteur décrit précédemment. Créer un main qui lance plusieurs Compteurs en parallèle et observer l'ordonnancement (quel thread est actif à quel moment). Utiliser la commande "top" dans le shell ou un moniteur de resources sous Linux/Windows pour observer les processus.

Retrait de l'état actif

Dans le code exéctué par un thread il est possible de demander à passer en état d'inéligibilité pendant un certain temps. Le code qui s'exécute va s'interrompre, et le thread est retiré du pool des threads éligibles par l'ordonnanceur :

```
try {Thread.sleep(1000);}
 catch (InterruptedException exc) {...// message?}
```

Thread.sleep(**long** millis) rend le thread qui l'exécute inéligible pendant au moins millis millisecondes. Remarque : cette méthode est susceptible de propager une exception si on invoque sur l'instance courante de ce thread la méthode interrupt().

La méthode yield() permet au thread qui l'exécute de demander à l'ordonnanceur de bien vouloir le faire passer à l'état éligible. Si il existe d'autres threads de priorité au moins équivalente qui sont en attente il est possible que l'ordonnanceur décide de rendre un de ces autres threads actif. Mais c'est sans garantit ! L'ordonnanceur est maître de sa stratégie.

Se rejoindre

La méthode d'instance join() permet au Thread qui l'exécute d'attendre la fin de l'exécution d'un Thread cible :

```
public class CompteurChaine extends Compteur{
 private Thread autre ;
 public CompteurChaine( int max, Thread autre){
  super(max) ;
  this.autre = autre ;
  }
 public void run () {
  System.out.println("prêt:" + this ) ;
  try {autre.join();}
   catch (InterruptedException exc) {/*test*/}
  super.run() ;
  }
}
```

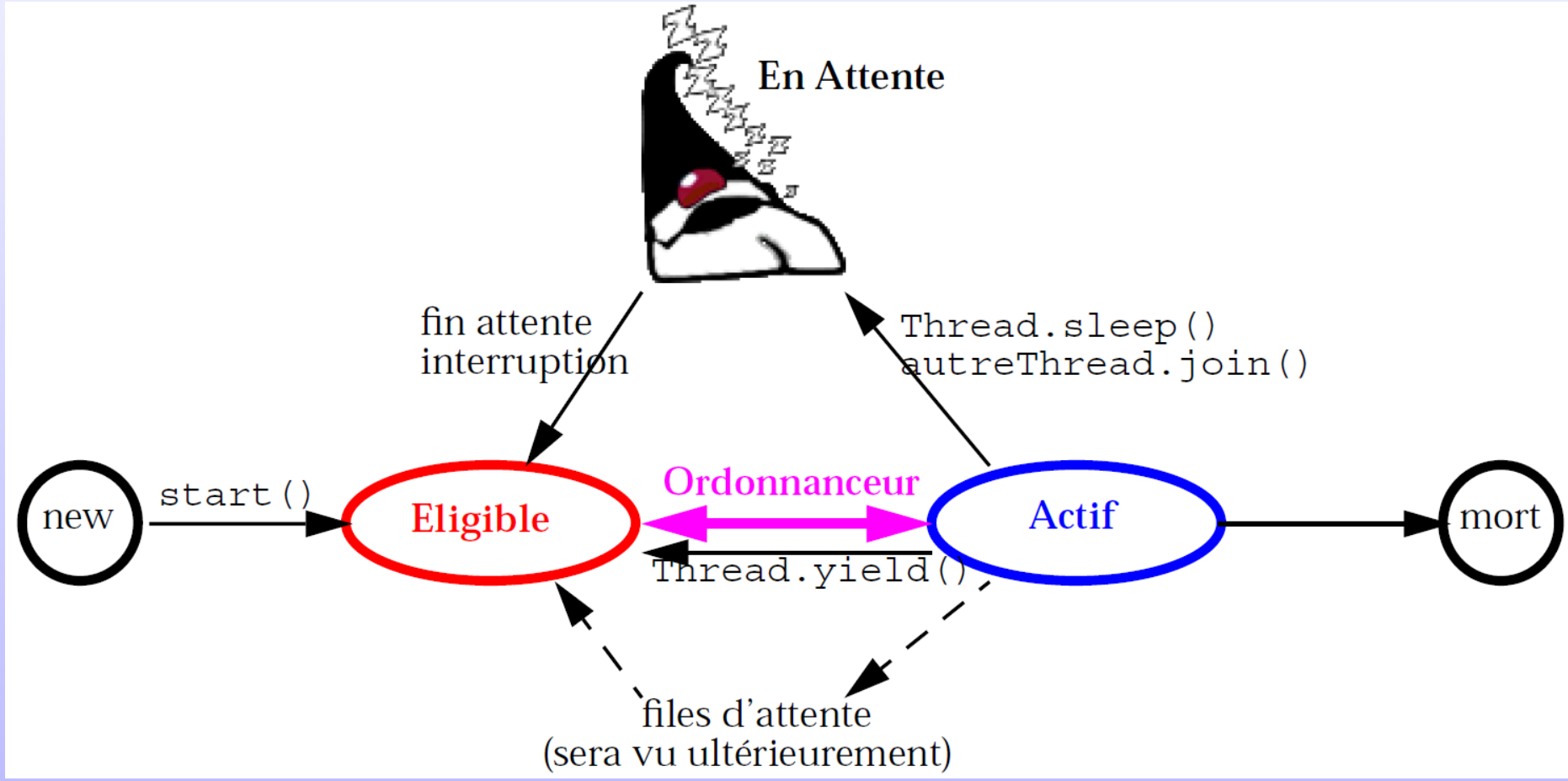
Le thread qui va exécuter ce Runnable va attendre la fin de l'exécution de l'autre Thread puis exécuter la suite du code.

Interruption et priorité

```
Thread patience = new Thread() {
                    public void run() {
                     while(!isInterrupted()) {System.out.print('.') ;}
                    }
                   };
patience.start();
..... //on fait des tas de choses
patience.interrupt() ;
try {patience.join() ;}
 catch(InterruptedException xc) {...}
```

- setPriority(**int** prior) : permet de fixer la priorité du Thread (valeur comprise entre MIN_PRIORITY et MAX_PRIORITY)
- setDaemon(**boolean** b) : permet de spécifier si le Thread est un "daemon" ou un Thread "utilisateur". Par défaut, le thread est utilisateur. La JVM s'arrête automatiquement quand les seuls threads actifs sont des "daemon". Attention à l'existence du thread de l'interface graphique !

Cycle de vie


En Attente
fin attente
interruption
Thread.sleep()
autreThread.join()
new
start()
Eligible
Ordonnanceur
Actif
mort
Thread.yield()
files d'attente
(sera vu ultérieurement)

Attributs d'un thread

- String name [rw] : son nom
- **long** id [ro] : son identité
- **int** priority [rw] : sa priorité (les Threads sont ordonnancés à l'aide de cette priorité)
- **boolean** daemon [rw] : son mode d'exécution
- Thread.State state [ro] : son état parmi :
  - NEW, RUNNABLE, BLOCKED, WAITING,
  - TIMED_WAITING, TERMINATED
- sa pile (mais dont on ne peut qu'observer la trace)
- son groupe de Thread
- quelques autres attributs mineurs ...

Variables partagées liées à un Thread

Des fois, on peut avoir besoin d'une variable partagée qui ne soit pas liée à une classe (membre static) mais liée à un contexte de thread. Typiquement une telle variable pourrait être un identifiant de session utilisateur sur un serveur : un ensemble de codes pourraient avoir besoin d'une variable partagée qui soit spécifique au thread qui les exécute. Pour répondre à cette situation on peut mettre en place un objet partagé de type ThreadLocal.

```
public class Session {
 public static ThreadLocal uid = new ThreadLocal() ; ...
}
public class TacheUtilisateur implements Runnable {
   Utilisateur ut ;
   public TacheUtilisateur(Utilisateur u) {ut = u ;}
   public void run() {Session.uid.set(ut) ; ...} ...
}
```

et maintenant dans un code de l'application :

```
utilisateurCourant= (Utilisateur) Session.uid.get();
```

Image de la mémoire

Le partage de données par plusieurs threads pose des problèmes. Il faut savoir qu'en Java les variables sont stockées dans une mémoire principale qui contient une copie de référence de la variable. Chaque Thread dispose d'une copie de travail des variables sur lesquelles il travaille. Les mises à jour réciproques de ces versions d'une même variable reposent sur des mécanismes complexes qui permettent des réalisations performantes de JVM.

Il est possible de forcer des réconciliations de valeurs en déclarant des variables membres comme **volatile**.

```
public class Partage {
 public volatile int valeurCourante;
 ...
}
```

Dans ce cas un Thread sait que lorsqu'il "prend" une copie de cette variable, sa valeur peut subir des modifications et les compilateurs doivent s'interdire certaines optimisations.

Volatile

Rôle

Utilisation sur des variables modifiables de manière asynchrone (plusieurs threads y ont accès “simultanément”). Le fait d’employer **volatile** oblige la JVM à rafraîchir son contenu à chaque utilisation. Ainsi, on est sûr qu’on n’accède pas à la valeur mise en cache mais bien à la valeur correcte.

Déclaration

public volatile Integer = 5;

Les executor (1)

Définition

Un executor est une interface permettant d'executer des tâches.

public interface Executor { **void** execute (Runnable command); } // *Asynchrone*

Un Executor permet de découpler ce qui doit être fait (une tâche définie par un Runnable) de quand et comment le faire explicitement (par un Thread) :

- Pool de threads (réeutilisation de threads, schémas prod/cons)
- Contraintes d'exécution (gestion des resources) :
  - dans quel thread exécuter quelle tâche ?
  - dans quel ordre (FIFO, LIFO, par priorité) ?
  - combien de tâches peuvent être exécutée en concurrence ? Combien de Thread au maximum ?
  - combien peuvent être mises en attente ?
  - en cas de surcharge, quelle tâche sacrifier ?

Les executor (2), méthodes statiques

ExecutorService newFixedThreadPool(**int** n)

La méthode renvoie un pool de taille fixée n, rempli par ajout de threads jusqu'à atteindre la limite, les threads qui meurent sont remplacés au fur et à mesure.

ExecutorService newCachedThreadPool() :

La méthode renvoie un pool dynamique qui se vide ou se remplit selon la charge. On crée un nouveau Thread que si aucun thread n'est disponible. Les Thread sans tâche pendant une minute sont détruits.

ExecutorService newSingleThreadExecutor()

renvoie un pool contenant un seul thread, remplacé en cas de mort inattendue. La différence avec newFixedThreadPool est que le nombre de Threads ne pourra pas être reconfiguré.

ExecutorService newScheduledThreadPool(**int** n)

renvoie un pool de taille fixée n qui peut exécuter des tâches avec délai ou périodiquement.

ExecutorService et cycle de vie

L'ExecutorService offre un service et propose des méthodes de gestion fines :

- **void** shutdown() : arrêt propre ; aucune nouvelle tâche ne peut être acceptée mais toutes celles soumises sont exécutées.
- List<Runnable> shutdownNow() : tente d'arrêter les tâches en cours (par InterruptedException) et renvoie la liste des tâches soumises mais n'ayant pas débutées.

L'exécution est asynchrone : une tâche est successivement :

$$\text{soumise} \Rightarrow \text{en cours} \Rightarrow \text{terminée}$$

Au travail !