Algorithmique parallèle et distribuée :
Processus légers, implémentation en langage Java

Julien Rossit

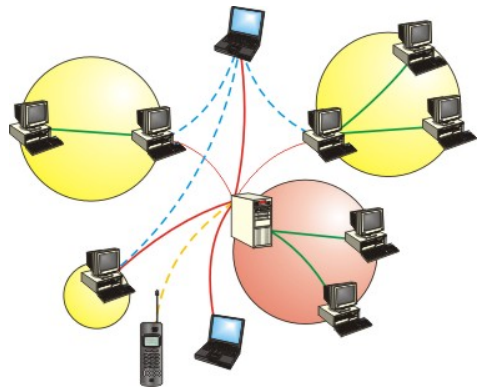
julien.rossit@parisdescartes.fr

IUT Paris Descartes

UNIVERSITÉ PARIS DESCARTES
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE

De **gros besoin actuels** en *programmation parallèle et distribuée* :

- réseaux locaux et à large échelle (web) ;
- systèmes *multi-processus* ;
- machines *multi-processeurs* (ou multi-cœurs) ;
- applications **multi-threadées**.



Motivations et problèmes

Motivations :

- échange d'informations et partage de ressources ;
- amélioration des performances (parallélisassion) ;
- amélioration de la sûreté (réplication).

Difficile à mettre en oeuvre ?

Problèmes :

- perte, corruption, duplication et/ou déséquencement des messages ;
- temps de communication ;
- homogénéité ou non des différents traitements parallèles ;
- détection de terminaison ;
- allocation de ressources ;
- exclusion mutuelle ;
- détection des inter-blocages et résolution ;
- etc.

Motivations et problèmes

Motivations :

- échange d'informations et partage de ressources ;
- amélioration des performances (parallélisassion) ;
- amélioration de la sûreté (réplication).

Difficile à mettre en oeuvre ?

Problèmes :

- perte, corruption, duplication et/ou déséquencement des messages ;
- temps de communication ;
- homogénéité ou non des différents traitements parallèles ;
- détection de terminaison ;
- allocation de ressources ;
- exclusion mutuelle ;
- détection des inter-blocages et résolution ;
- etc.

Outils disponibles

Système :

- sémaphore ;
- moniteurs ;
- pipes.

Réseau :

- le modèle OSI ;
- le modèle IEEE.

Langage :

- expression du parallélisme ;
- expression de l'indéterminisme ;
- expression des interactions.

Difficultés

Absence de la connaissance de l'état global du système :

- délais de transmission ;
- état des moyens de communication ;
- vitesse d'exécution relative de chacun des traitements ;
- politique d'ordonnancement.

Problème :

Non déterminisme !

Déroulement

1. Le langage Java
   - Philosophie
   - Historique
   - Environnement et outils
2. Les processus légers
   - Principe
   - Cycle de vie
   - Threads vs processus
3. Implémentation en Java
   - La classe Thread
   - L'interface Runnable
   - Remarques

Petits rappels sur le langage Java

Un bref petit tour du langage...

Un langage orientée objet, structuré et impératif,
sous licence GNU GPL.

Philosophie

- simple, orienté objet et familier ;
- robuste et sûr (*Garbage Collector*) ;
- indépendant de la machine employée pour l'exécution (bytecode) : *compile once, run everywhere* ;
- très performant ;
- interprété, multi-tâches et dynamique.

Historique

1990 projet *Stealth*, puis *Green Project* (*J. Gosling*, *M. Sheridan*) ;
1992 *OAK* (*J. Gosling*, *B. Joy*) ;
1994 lancement de *Java Development Kit* (JDK 1.0) ;
1995 premier lancement commercial ;
1998 lancement de *Java 2 Software Development Kit* (J2SDK) ;
1999 lancement de *Java 2 Enterprise Edition* (J2EE) ;
2000 lancement de *Java 2 version 1.3* (J2SE 1.3) ;
2002 lancement de (J2SE 1.4) ;
2004 lancement de *Java SE version 5.0* (J2SE 5.0) ;
2006 annonce de passage sous *licence GPL* ;
lancement de *Java SE version 6* (J2SE 6) ;
2009 rachat de *Sun* par *Oracle* ;
2010 *J. Gosling* démissionne ;
2011 lancement de *J2SE 7*;
2014 lancement de *J2SE 8*.

3 éditions

- *J2SE (Standard Edition)* : librairies, machine virtuelle et compilateur ;
- *J2EE (Enterprise Edition)* : librairies et serveur pour des applications d'entreprise ;
- *J2ME (Micro Edition)* : environnement de développement pour des systèmes embarqués.

De nombreuses technologies pour les applications d'entreprise autour de Java : JSP, servlet, JDBC, JMS, JavaIDL, JavaMail, RMI, JCE, JSSE, JNDI, JTS, JAX, JavaSpeech, Java3D, JavaCard, JavaPhone, JavaTV, JMX, JMI, etc.

Outils

- nombreux IDEs : Eclipse, NetBeans, JBuilder, IntelliJ, etc ;
- machines virtuelles : HotSpot, KaffeVM, LaTTe, Jikes RVM, etc ;
- compilateurs : Javac, Jikes, ECJ, GCJ, etc.

Les processus légers (threads)

Des traitements parallèles

Les *processus* (dits lourds) :

- communications locales par fichier (pipe, . . .) ;
- communications distantes par envoi de messages (socket).

Les *threads* (dits processus légers) :

- sont en quelque sorte un processus à l'intérieur d'un processus ;
- partagent la même zone mémoire ;
- communiquent par variables partagées ;
- possèdent leur propre environnement d'exécution ;

De cette façon, un processus :

- partage ses ressources entre les threads qui le composent (temps processeur, mémoire) ;
- possède au moins un thread qui exécute le programme principal (fonction *main()*).

Des traitements parallèles

Les *processus* (dits lourds) :

- communications locales par fichier (pipe, ...) ;
- communications distantes par envoi de messages (socket).

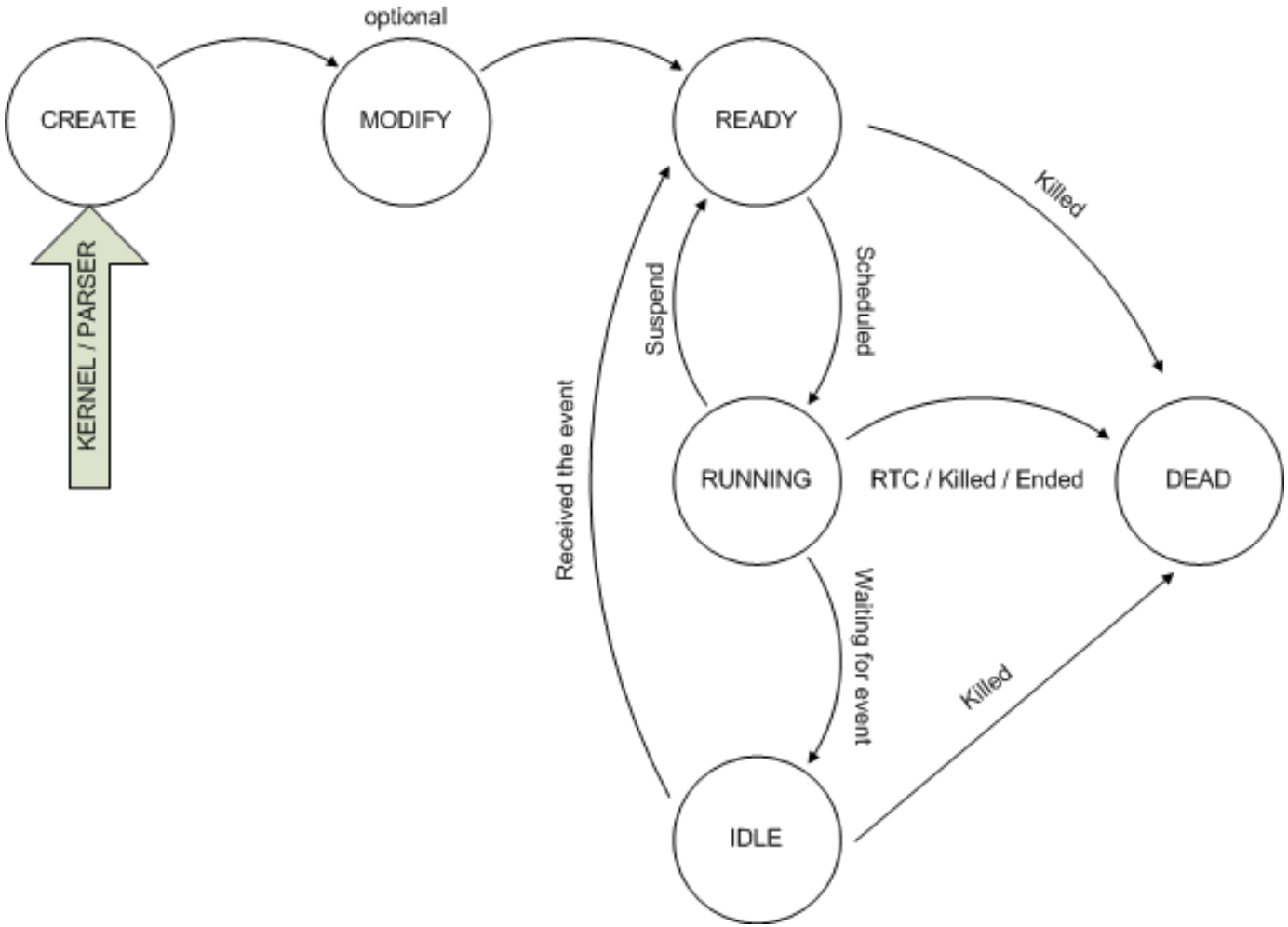
Les *threads* (dits processus légers) :

- sont en quelque sorte un processus à l'intérieur d'un processus ;
- partagent la même zone mémoire ;
- communiquent par variables partagées ;
- possèdent leur propre environnement d'exécution ;

De cette façon, un processus :

- partage ses ressources entre les threads qui le composent (temps processeur, mémoire) ;
- possède au moins un thread qui exécute le programme principal (fonction *main()*).

Cycle de vie


optional
CREATE
MODIFY
READY
KERNEL / PARSER
Killed
Scheduled
Suspend
Received the event
RUNNING
RTC / Killed / Ended
DEAD
Waiting for event
Killed
IDLE

Threads vs processus

Partage de mémoire

- processus : espaces mémoire séparés ;
- threads : espace mémoire partagé (mais piles d'exécution différentes).

En résumé

- plus efficace ;
- moins robuste :
  - le plantage d'un thread peut perturber les autres ;
  - le plantage d'un processus n'a pas (normalement) d'incidence sur les autres.

Utiliser des approches mixtes :

plusieurs processus ayant plusieurs threads chacun.

Implémentation des threads en Java

Implémentation des threads en Java

Le langage Java et la machine virtuelle offrent la possibilité de programmer des traitements concurrents :

- simplification la programmation dans de nombreux cas :
  - programmation événementielle (ex. GUI) ;
  - entrée/sorties non bloquantes ;
  - timers, déclenchements périodiques ;
  - servir plusieurs clients simultanément (serveur Web, BD, ...).
- meilleure utilisation des capacités de la machine, utilisation des temps morts.

La classe Thread

1^{ere} méthode :

- hériter de la classe *Thread*;
- surcharger la méthode *run()* :

Code

```
class MonThread extends Thread {
  MonThread() {
    ... code du constructeur ...
  }

  public void run() {
    ... code a exécuter dans le thread ...
  }
}
```

Instanciation et lancement

Un appel à la méthode *start()* (classe *Thread*) lance le thread :

Code

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

Déroulement :

1. passage de l'état du thread à *prêt* ;
2. la machine virtuelle décide du moment d'exécution ;
3. appel de la méthode *run()*.

L'interface Runnable

2^{eme} méthode :

- implémenter l'interface *Runnable* ;
- implémenter la méthode *run()* :

Code

```
class MonThread implements Runnable {
  MonThread() {
    ... code du constructeur ...
  }

  public void run() {
    ... code a exécuter dans le thread ...
  }
}
```

Instanciation et lancement

Le constructeur de la classe *Thread* prend en argument instance implémentant *Runnable* :

1. instanciation la classe implémentant l'interface ;
2. création d'une instance de *Thread* ;
3. appeler la méthode *start()* :

Code

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

Contexte d'utilisation

Quelques principes généraux :

- les instructions du thread sont définies dans la méthode *run()* (seule signature possible *public void run()*) ;
- exécution concurrente du programme lanceur et du thread ;
- méthode *main()* est associée automatiquement à un thread ;
- création d'autant de threads que nécessaire, soit de à la même classe, soit de classes différentes ;
- appel de *start()* une seule et unique fois pour chaque thread ;
- un thread meurt lorsque sa méthode run() se termine ;
- jamais d'appel direct à la méthode *run()* !
- deux (ou plus) threads peuvent exécuter la même méthode simultanément :
  - des flux d'exécutions distincts (une piles par thread) ;
  - mais même espace mémoire partagé !

Contexte d'utilisation (suite)

Pas de passage de paramètre via la méthode *start()* !

- définir des variables d'instance ;
- les initialiser lors de la construction.

Exemple :

```
public class MonThread implements Runnable {
  int unEntier ;
  Object unObjet ;

  public MonThread(int unEntier,Object unObjet) {
    this.unEntier=unEntier;
    this.unObjet=unObjet;
  }

  public void run() {... unEntier ... unObjet ...}
}

new MonThread(123,uneReference).start();
```

Pour aller plus loin

- Les groupes de thread (*ThreadGroup*)

Exemple :

```
ThreadGroup monGroupe  = new ThreadGroup("groupe de threads");
Thread t1 = new Thread(monGroupe,new MonThread(), "thread 1");
Thread t2 = new Thread(monGroupe,new MonThread(), "thread 2");
```

permettent (entre autres) de restreindre l'accès au threads, de lancer une action sur tous les threads d'un même groupe ;

- les démons (*daemons*) sont des thread ne nécessitant pas d'intéraction avec l'utilisateur. Ils se terminent automatiquement lorsque plus aucun thread classique n'est actif ;

- etc.

Des questions ?