Programmation concurrente en java

M. Belguidoum

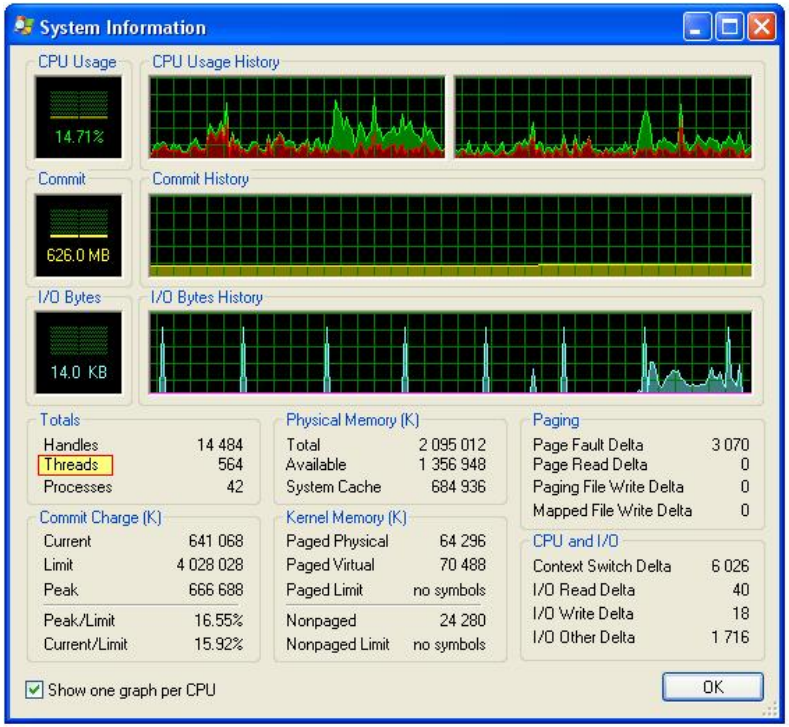
Université Mentouri de Constantine
Département Informatique

Plan

Rappel : les processus

- Un système **multi-tâches** est capable d'exécuter plusieurs programmes en parallèle sur une même machine
- La plupart du temps, une machine n'a qu'un seul processeur qui ne peux exécuter qu'un seul programme à la fois
- Un programme en cours d'éxécution est appelé **un processus**
- La plupart des systèmes d'exploitation sont équipés d'un ordonnançeur de tâches qui donne à tour de rôle le processeur aux processus.
- Chaque processus est activé de façon cyclique et pendant une courte durée ce qui donne à l'utilisateur l'impression que plusieurs processus sont en cours d'exécution.

Rappel : les processus



- **Processus** = **ensemble d'instructions + état d'exécution**
  (pile, registres, pc, tas, descripteurs d'E/S, gestionnaires de signaux...)
- Deux classes principales de processus
  - **Processus lourd** (ou tâche ou processus) : ne partage pas son état Sauf des espaces mémoire partagés déclarés explicitement
  - **Processus léger** (ou *thread*) : partage son tas, ses descripteurs et ses gestionnaires

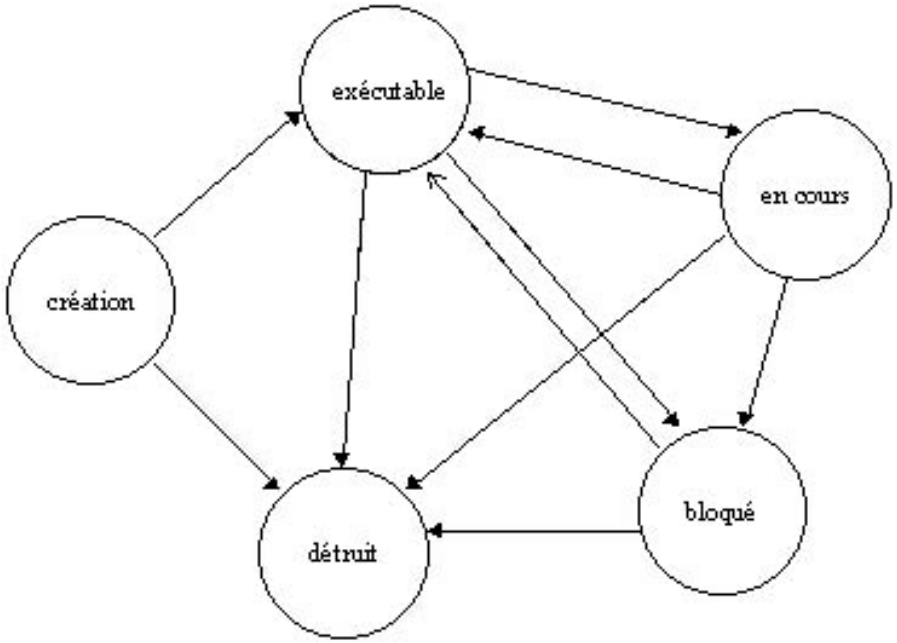
Qu'est ce qu'un thread ?

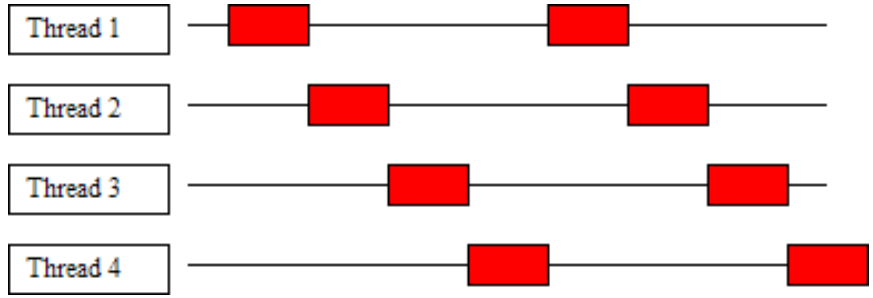
Thread

Un thread (appelée aussi processus léger ou activité) est un fil d'instructions (un chemin d'exécution) à l'intérieur d'un processus.

- Les ressources allouées à un processus (temps processeur, mémoire) vont être partagées entre les threads qui le composent.
- Les threads d'un même processus partagent le même espace d'adressage, les mêmes variables d'environnement, les mêmes données, etc contrairement aux processus.
- Un processus possède au moins un thread (qui exécute le programme principal main()).
- Les programmes qui utilisent plusieurs threads sont dits multithreadés.
- Les threads font partie intégrante du langage JAVA. Elles sont gérées grâce à la classe Thread.

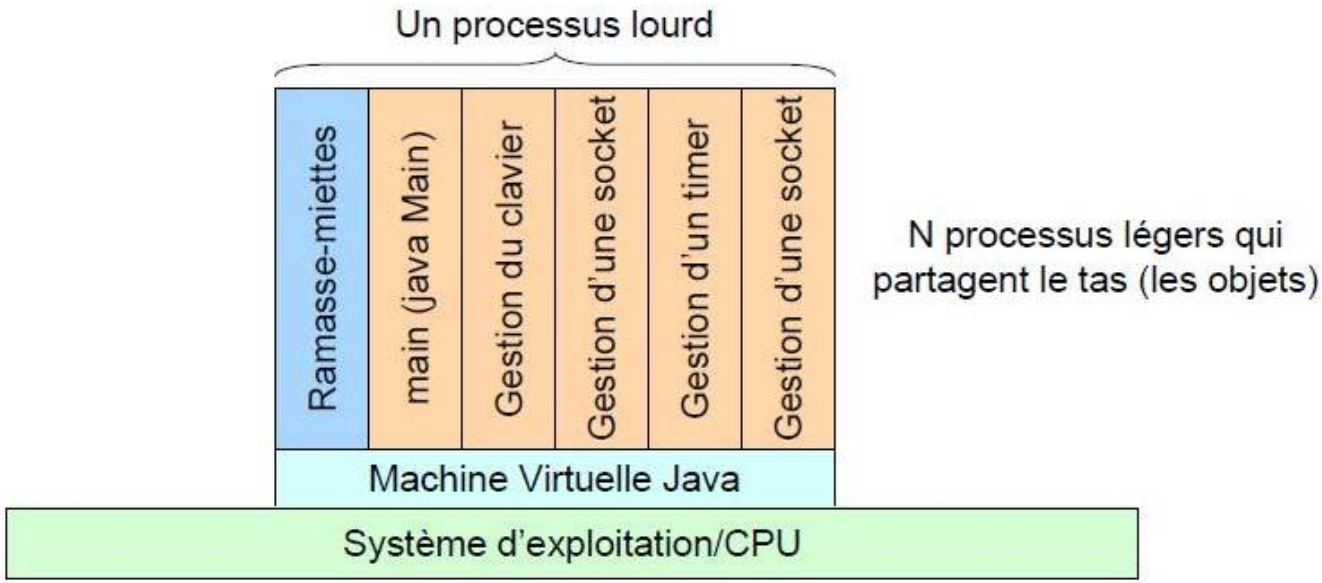
Etats des threads





Les threads en Java

- **1 JVM = un processus lourd**
- thread principal : main() et les autres pour : ramasse-miettes, gestion du clavier, etc.
- tous les threads Java partagent la mémoire (les objets)



Les threads en Java

- pour réaliser plusieurs tâches en parallèle, il est possible de créer des threads au sein des applications Java
- deux techniques existent :
  - créer un thread par un **héritage** de la classe `java.lang.Thread` (l'héritage multiple n'est pas autorisé en java)
  - créer un thread en **implémentant l'interface** `java.lang.Runnable`.
- aborder la gestion de la synchronisation des threads

Création et démarrage d'un thread : héritage

- créer une classe qui hérite de la class Thread
- redéfinir la méthode `run()` pour y inclure les traitements à exécuter par le thread

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

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

- Pour créer et exécuter un tel thread, il faut instancier un objet et appeler sa méthode start() qui va créer le thread et elle-même appeler la méthode run().

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

Création et démarrage d'un thread : héritage

- L'appel de la méthode `start()` passe le thread à l'état "prêt"
- Lorsque le thread démarre, JAVA appelle sa méthode `run()`.
- Un thread se termine lorsque sa méthode run() termine.

Création et démarrage d'un thread : l'interface Runnable

Exemple

```
class MonThread implements Runnable {
public void run() {
System.out.println("I'm a thread!");
}
public static void main(String args[]) {
// Un thread possède optionnellement un nom symbolique
Thread t = new Thread(new MonThread(), "My Thread");
// MonThread.run() est démarré dans un nouveau thread après
//l'appel de start()
t.start();
System.out.println("Ce code s'exécute en // de run()");
}
}
```

Création et démarrage d'un thread : l'interface `Runnable`

- Déclarer une classe qui implémente l'interface `Runnable`
- Cette interface déclare seulement une méthode : `run()`.
- redéfinir sa seule et unique méthode run() pour y inclure les traitements à exécuter dans le thread. La classe Thread a un constructeur new Thread(Runnable).
- L'argument du constructeur est donc toute instance de classe implémentant cette méthode `run()`.

Création et démarrage d'un thread : l'interface Runnable

- La classe se déclare comme dans l'exemple précédent, mais on implémente Runnable au lieu d'hériter de Thread :

```
class MonThread2 implements Runnable {
    MonThread2() {
        ... code du constructeur ...
    }
    public void run() {
        ... code à exécuter dans le thread ...
    }
}
```

- Pour créer et lancer un thread, on crée d'abord une instance de MonThread2, puis une instance de Thread sur laquelle on appelle la méthode start() :

```
MonThread2 p = new MonThread2();
Thread T = new Thread(p);
T.start();
```

Création et démarrage d'un thread : l'interface Runnable

Exemple

```
class MonThread extends Thread {
public void run() {
System.out.println("I'm a thread!");
}
public static void main(String args[]) {
// MonThread sera un Thread nommé automatiquement "Thread-1"
Thread t = new MonThread();
// MonThread.run() est démarré dans un nouveau thread après
//  l'appel de start()
t.start();
}
}
```

Quelle technique choisir ?

	Avantages	**Inconvénients**
extends java.lang.Thread	Chaque thread a ses données qui lui sont propres	On ne peut plus hériter d'une autre classe
implements java.lang.Runnable	L'héritage reste possible. En effet, on peut implémenter autant d'interfaces que l'on souhaite.	Les attributs de votre classe sont partagés pour tous les threads qui y sont basés. Dans certains cas, il peut s'avérer que cela soit un atout.

Thread : Quelques propriétés et méthodes

- `void destroy()` : met fin brutalement au thread.
- `int getPriority()` : renvoie la priorité du thread.
- `void setPriority(int)` : modifie la priorité d'un thread
- `ThreadGroup getThreadGroup()` : renvoie un objet qui encapsule le groupe auquel appartient le thread.
- `boolean isAlive()` : renvoie un booléen qui indique si le thread est actif ou non.
- `boolean isInterrupted()` : renvoie un booléen qui indique si le thread a été interrompu.
- `void start()` : démarrer le thread et exécuter la méthode `run()`.
- `currentThread()` : donne le thread actuellement en cours d'exécution.
- `setName()` : fixe le nom du thread.
- `getName()` : nom du thread.

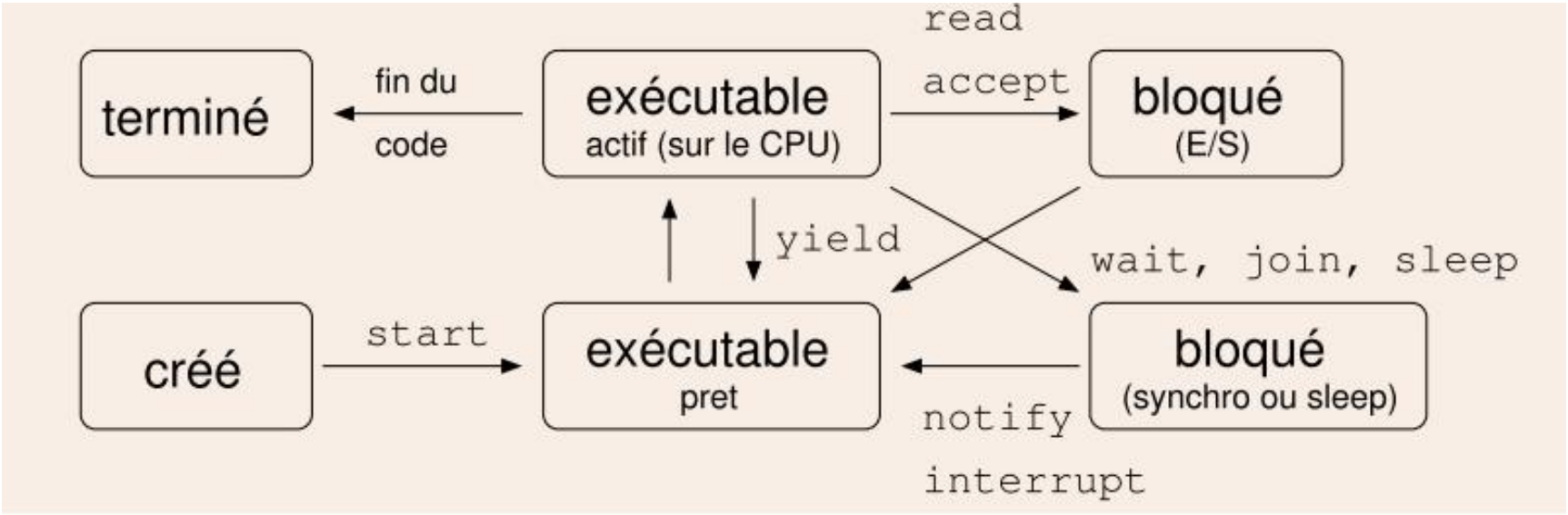
Thread : Quelques propriétés et méthodes

- `isAlive()` : indique si le thread est actif(true) ou non (false).
- `void resume()` : reprend l'exécution du thread() préalablement suspendu par `suspend()`.
- `void run()` : méthode déclarée par l'interface Runnable : elle doit contenir le code qui sera exécuté par le thread.
- `void start()` : lance l'exécution d'un thread
- `void suspend()` : suspend le thread jusqu'au moment où il sera relancé par la méthode `resume()`.
- `void yield()` : indique à l'interpréteur que le thread peut être suspendu pour permettre à d'autres threads de s'exécuter.
- `void sleep(long)` : mettre le thread en attente durant le temps exprimé en millisecondes fourni en paramètre. Cette méthode peut lever une exception de type InterruptedException si le thread est réactivé avant la fin du temps.
- `void join()` : opération bloquante - attend la fin du thread pour passer à l'instruction suivante

Cycle de vie d'un thread

- Le comportement de la méthode `start()` de la classe Thread dépend de la façon dont l'objet est instancié
  - si l'objet qui reçoit le message `start()` est instancié avec un constructeur qui prend en paramètre un objet Runnable, c'est la méthode `run()` de cet objet qui est appelée.
  - si l'objet qui reçoit le message `start()` est instancié avec un constructeur qui ne prend pas en paramètre une référence sur un objet Runnable, c'est la méthode `run()` de l'objet qui reçoit le message `start()` qui est appelée.
- Un thread en cours de traitement s'exécute jusqu'à ce qu'il soit : achevé, ou stoppé par un appel à la méthode `stop()`, ou en sortant de la méthode `run`, ou interrompu pour passer la main par `yield()`, ou mis en sommeil par `sleep()`, ou désactivé temporairement par suspend() ou wait().

Cycle de vie d'un thread



Gestion de la propriété d'un thread

- La priorité du nouveau thread est égale à celle du thread dans lequel il est créé.
- un thread ayant une priorité plus haute recevra plus fréquemment le processeur qu'un autre thread
- La priorité d'un thread varie de 1 à 10. La classe Thread définit trois constantes :
  - `MIN_PRIORITY` : priorité inférieure (0)
  - `NORM_PRIORITY` : priorité standard (5 : la valeur par défaut)
  - `MAX_PRIORITY` : priorité supérieure (10)
- Pour déterminer la priorité d'un thread on utilise la méthode `getPriority()` pour la modifier on utilise `setPriority(int)`

Gestion d'un groupe de thread

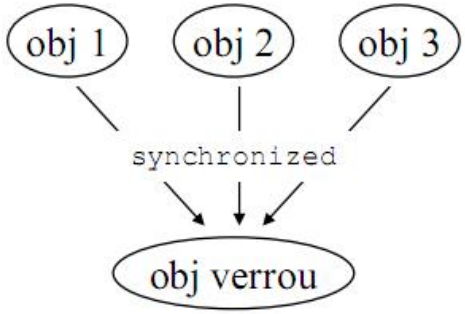
- La classe ThreadGroup représente un ensemble de threads, il est ainsi possible de regrouper des threads selon différents critères.
- Il suffit de créer un objet de la classe ThreadGroup et de lui affecter les différents threads.
- Un objet ThreadGroup peut contenir des threads mais aussi d'autres objets de type ThreadGroup.
- La notion de groupe permet de limiter l'accès aux autres threads : chaque thread ne peut manipuler que les threads de son groupe d'appartenance ou des groupes subordonnés.
- La classe ThreadGroup possède deux constructeurs :
  - `ThreadGroup(String nom)` : création d'un groupe avec attribution d'un nom
  - `ThreadGroup(ThreadGoup groupe_parent, String nom)` :création d'un groupe à l'intérieur du groupe spécifié avec l'attribution d'un nom
- Pour ajouter un thread à un groupe, il suffit de préciser le groupe en paramètre du constructeur du thread.

Synchronisation des threads

- Plusieurs threads peuvent accéder à un objets conccurent (problème d'accès concurrent) => Introduction de la notion de section critique
- On ajoute le mot clé `synchronized` dans l'en-tête des méthodes.
- Ces méthodes servent à construire ce que l'on appelle des "moniteurs" c'est-à-dire des structures de données qui sont protégées de telle manière que seules des procédures qui travaillent en exclusion mutuelles puissent accéder aux objets.

Synchronisation des threads : exemple

Plusieurs processus veulent accéder à un compte en banque => utiliser la commande synchronized qui fait en sorte que les méthodes soient exécutées en exclusion mutuelle.




```
public class Compte
int solde = 0;
public synchronized void deposer(int s) {
int so = solde + s;
solde = so;
}
public synchronized void retirer(int s) {
int so = solde - s;
solde = so;
}
}
```

Concurrence d'accès : exercice

Exercice : crédit sur un compte commun

Soit une classe gérant les sommes créditées sur un compte commun. Écrire le programme formé d'un programme principal et de 2 acteurs apportant respectivement 35000 DA et 25000 DA. À la suite de tout ajout, le solde du compte est affiché par les acteurs. À la fin de l'exécution des acteurs, le solde du compte commun est affiché par le programme principal.

Concurrence d'accès : exercice

Une solution

```
class PartageMemoire extends Thread {
private static int cpteCommun = 0;
private int salaire;
PartageMemoire ( int monSalaire ) {
salaire = monSalaire;
}
public void run() {
cpteCommun = cpteCommun + salaire;
System.out.println( "Cpte Commun "+ cpteCommun + " DA" );}
public static void main(String args[]) {
Thread t1 = new PartageMemoire ( 35000 );
Thread t2 = new PartageMemoire ( 25000 );
t1.start();
t2.start();
try {
t1.join();
t2.join();}
catch (InterruptedException e){}
System.out.println( "Cpte Commun final "+ cpteCommun + " DA" );
}}
```

Notion de verrous

- Si d'autres threads cherchent à verrouiller le même objet, ils seront endormis jusqu'à que l'objet soit déverrouillé => mettre en place la notion de section critique.
- Pour verrouiller un objet par un thread, il faut utiliser le mot clé `synchronized`.
- Il existe deux façons de définir une section critique.
  - soit on synchronise un ensemble d'instructions sur un objet :

```
synchronized(object) {
    // Instructions de manipulation d'une ressource
partagée.
}
```

  - soit on synchronise directement l'exécution d'une méthode pour une classe donnée.

```
public synchronized void meth(int param) {
    // Le code de la méthode synchronizée.
}
```

L'exclusion mutuelle

- Chaque objet Java possède un verrou dont la clé est gérée par la JVM.
- lorsqu'un thread souhaite accéder à une méthode synchronisée d'un objet, il demande la clé de cet objet à la JVM, entre dans la méthode, puis ferme le verrou à clé.
- De cette façon, aucun autre thread ne peut accéder aux méthodes synchronisées de cet objet.
- Lorsque le thread sort de la méthode synchronisée, il ouvre de nouveau le verrou et rend la clé à la JVM.
- Un autre thread peut alors accéder aux méthodes synchronisées de l'objet.

Synchronisation temporelle : wait et notify

- Les méthodes `wait()`, `notify()` et `notifyAll()` permettent de synchroniser différents threads.
- Ces méthodes sont définies dans la classe Object (car elles manipulent le verrou associé à un objet), mais ne doivent s'utiliser que dans des méthodes `synchronized`.
- `wait()` : le thread qui appelle cette méthode est bloqué jusqu'à ce qu'un autre thread appelle `notify()` ou `notifyAll()`.
- `wait()` libère le verrou, ce qui permet à d'autres threads d'exécuter des méthodes synchonisées du même objet.
- `notify()` et `notifyAll()` permettent de débloquer une tâche bloqué par `wait()`.
- si une tâche T1 appelle wait dans une méthode de l'objet O, seule une autre méthode du même objet pourra la débloquer ; cette méthode devra être synchronisée et exécutée par une autre tâche T2.

Bibliographie

- cours Lionel Seinturier, Gael Thomas...
- La documentation officielle et le tutoriel chez Sun : `http://download.oracle.com/javase/6/docs/index.html`
- Les tutoriels chez développez.com : `http://java.developpez.com/cours/`
- la paquetage Thread de java : `http://doc.java.sun.com/DocWeb/api/java.lang.Thread`
- Apprendre Java - Cours et exercices, Irène Charon
- Apprentissage du langage java, Serge Tahé
- Penser en Java 2nde édition, Bruce Eckel traduit en francais (Thinking in Java) : `http://bruce-eckel.developpez.com/livres/java/traduction/tij2/`
- ...