INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

Java

benjamin.nguyen@insa-cvl.fr

La concurrence en Java : les Threads

D’après J.-M. Doudoux,

Culler, Singh, Gupta (Parallel Computer Architecture: a Hardware/Softwaer Approach), 1998

Hennessy & Patterson (Computer Architecture, 5th edition), 2011

But du cours (compétences)

- Savoir coder une application complexe en Java (avec des tests unitaires)
- Savoir débugger une application complexe en Java
- Savoir gérer les fichiers en Java
- Savoir gérer la persistence en Java avec la sérialisation
- Savoir utiliser les threads en Java
- Savoir créer une interface graphique simple en Java (approche modèle MVC)

Si le temps le permet …

- Connaître les bibliothèques cryptographiques en Java
- Comprendre et avoir utilisé le mécanisme d'introspection en Java
- Savoir gérer la persistence en Java avec une base de données

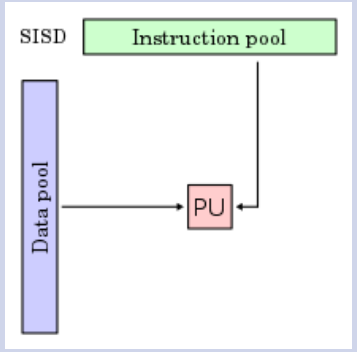
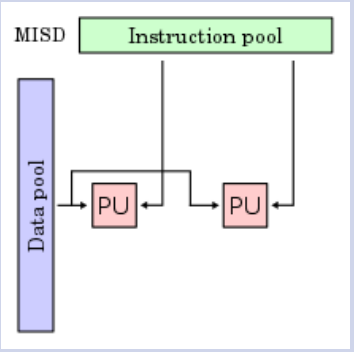
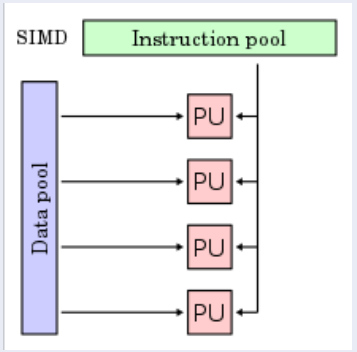
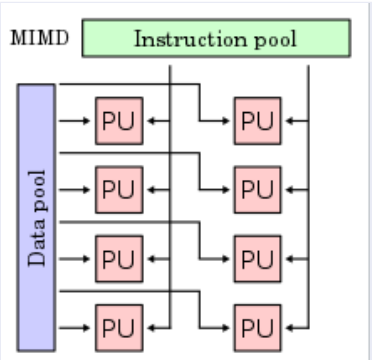
Multi-tâches et parallélisme : objectifs

- Certaines “tâches” (algorithmes ou fonctions) peuvent être effectuées indépendamment des autres
- Certaines “tâches” peuvent être effectuées indépendamment des autres mais l’ordre a une importance
- Certaines tâches doivent échanger des informations avec d’autres tâches

- Les ordinateurs actuels disposent tous de plusieurs processeurs ou coeurs permettant d’effectuer des tâches en simultané. Pour que cela apporte une accélaration au programme, il faut que ces tâches puissent s’exécuter de manière **indépendante**

Parallélisme 101 : Types de machines

- Taxonomie de Flynn

	Une seule instruction	Plusieurs instructions
Donnée unique		
Données différentes		

Images CBurnett, 2007, CC-BY-SA

Taxonomie de Flynn

- **SISD** : ordinateur classique, architecture Von Neumann, approche fetch/pipeline.
- **SIMD** : cartes graphiques (depuis 1976 avec le CRAY-1)
- **MISD** : pas vraiment utilisé (vérification et redondance dans les systèmes critiques)

- **MIMD** : machines multiprocesseur

  Mémoire partagée pour échanger des données ou mémoire distribuée :

  - Sémaphores
  - Verrous / mutex
  - Barrières de synchronisation

Parallélisme réel ou simulé

- La programmation parallèle (concurrente) est un paradigme de programmation
- On considère :
  - Une pile d'exécution
  - Un ensemble de données privées
- On peut écrire un programme parallèle … et l'exécuter sur une machine non parallèle !

Type de concurrence

- **Disjointe** : les entités concurrentes ne communiquent pas et n'interagissent pas entre elles
- **Compétitive** : un ensemble d'entités concurrentes en compétition pour l'accès à certaines ressources partagées (par exemple le temps CPU, un port d'entrées/sorties, une zone mémoire)
- **Coopérative** : un ensemble d'entités concurrentes qui coopèrent pour atteindre un objectif commun. Des échanges ont lieu entre les processus.

Problème de deadlock (interblocage)

- Peut se produire si deux tâches s'attendent mutuellement
- Comment empêcher qu'un interblocage ne se produise ? (approche *préventive*)
- Comment résoudre une situation d'interblocage si celle-ci se produit ? (approche *curative*)

Et en Java ?

- Utilisation de
  - l’interface java.lang.Runnable
  - la classe java.lang.Thread

Cycle de vie d’un Thread

Valeur	Description
NEW	Le thread n'est pas encore démarré. Aucune ressource système ne lui est encore affectée. Seules les méthodes de changement de statut du thread start() et stop() peuvent être invoquées
RUNNABLE	Le thread est en cours d'exécution : sa méthode start() a été invoquée
BLOCKED	Le thread est en attente de l'obtention d'un moniteur qui est déjà détenu par un autre thread
WAITING	Le thread est en attente d'une action d'un autre thread ou que la durée précisée en paramètre de la méthode sleep() soit atteinte. Chaque situation d'attente ne possède qu'une seule condition pour retourner au statut Runnable : •si la méthode sleep() a été invoquée alors le thread ne retournera à l'état Runnable que lorsque le délai précisé en paramètre de la méthode a été atteint •si la méthode suspend() a été invoquée alors le thread ne retournera à l'état Runnable que lorsque la méthode resume sera invoquée •si la méthode wait() d'un objet a été invoquée alors le thread ne retournera à l'état Runnable que lorsque la méthode notify() ou notifyAll() de l'objet sera invoquée •si le thread est en attente à cause d'un accès I/O alors le thread ne retournera à l'état Runnable que lorsque cet accès sera terminé
TIMED_WAITING	Le thread est en attente pendent un certain temps d'une action d'un autre thread. Le thread retournera à l'état Runnable lorsque cette action survient ou lorsque le délai d'attente est atteint
TERMINATED	Le thread a terminé son exécution. La fin d'un thread peut survenir de deux manières : •la fin des traitements est atteinte •une exception est levée durant l'exécution de ses traitements

Interface `Runnable`

- Cette interface doit être implémentée par toute classe qui contiendra des traitements à exécuter dans un thread.
- Cette interface ne définit qu'une seule méthode : `void run()`.
- Dans les classes qui implémentent cette interface, la méthode `run()` doit être redéfinie pour contenir le code des traitements qui seront exécutés dans le thread.
- **Important :** Intérêt d'utiliser un `Runnable` : utiliser si la seule chose qu'on veut faire c'est définir la méthode `run()`. Si on veut modifier d'autres méthodes, il faut hériter de `Thread`.

- Exemple fichier `Compteur.java`

Lancement d'un `Thread`

- Idée générale :
  - On doit créer une nouvelle instance d'une classe `Thread`, avec comme paramètre de construction une classe qui hérite de l'interface `Runnable`
  - On lance le Thread avec la méthode `start()`

Voir `ExempleThread.java`

Constructeurs de `Thread`

Constructeur	Rôle
Thread()	Créer une nouvelle instance
Thread(Runnable target)	Créer une nouvelle instance en précisant les traitements à exécuter
Thread(Runnable target, String name)	Créer une nouvelle instance en précisant les traitements à exécuter et son nom
Thread(String name)	Créer une nouvelle instance en précisant son nom
Thread(ThreadGroup group, Runnable target)	Créer une nouvelle instance en précisant son groupe et les traitements à exécuter
Thread(ThreadGroup group, Runnable target, String name)	Créer une nouvelle instance en précisant son groupe, les traitements à exécuter et son nom
Thread(ThreadGroup group, Runnable target, String name, long stackSize)	Créer une nouvelle instance en précisant son groupe, les traitements à exécuter, son nom et la taille de sa pile
Thread(ThreadGroup group, String name)	Créer une nouvelle instance en précisant son groupe et son nom

Autres méthodes de `Thread`

Méthode	Rôle
static int activeCount()	Renvoyer une estimation du nombre de threads actifs dans le groupe du thread courant et ses sous-groupes
void checkAccess()	Déterminer si le thread courant peut modifier le thread
static Thread currentThread()	**Renvoyer l'instance du thread courant**
static void dumpStack()	Afficher la stacktrace du thread courant sur la sortie standard d'erreur
static int enumerate(Thread[] tarray)	Copier dans le tableau fourni en paramètre chaque thread actif du groupe et des sous-groupes du thread courant
static Map<Thread,StackTraceElement[]> getAllStackTraces()	Renvoyer une collection de type Map qui contient pour chaque thread actif les éléments de sa stacktrace
int getPriority()	**Renvoyer la priorité du thread**
ThreadGroup getThreadGroup()	Renvoyer un objet qui encapsule le groupe auquel appartient le thread
static boolean holdsLock(Object obj)	Renvoyer un booléen qui précise si le thread possède le verrou sur le monitor de l'objet passé en paramètre
void interrupt()	Demander l'interruption du thread
static boolean interrupted()	Renvoyer un booléen qui précise si une demande d'interruption du thread a été demandée
boolean isAlive()	**Renvoyer un booléen qui indique si le thread est actif ou non**
boolean isInterrupted()	Renvoyer un booléen qui indique si le thread a été interrompu
void join()	Attendre la fin de l'exécution du thread
void join(long millis)	Attendre au plus le délai fourni en paramètre que le thread se termine
void join(long millis, int nanos)	Attendre au plus le délai fourni en paramètres (ms + ns) que le thread se termine
void run()	Contenir les traitements à exécuter
void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)	Définir le handler qui sera invoqué si une exception est levée durant l'exécution des traitements
static void sleep(long millis)	**Endormir le thread pour le délai exprimé en millisecondes précisé en paramètre**
static void sleep(long millis, int nanos)	Endormir le thread pour le délai précisés en paramètres
void start()	**Lancer l'exécution des traitements : associer des ressources systèmes pour l'exécution et invoquer la méthode run()**
String toString()	**Renvoyer une représentation textuelle du thread qui contient son nom, sa priorité et le nom du groupe auquel il appartient**
static void yield()	Demander au scheduler de laisser la main aux autres threads

Création d’un démon

- Thread qui va tourner en continu
- On doit le déclarer en tant que `daemon` avant de le lancer : `setDaemon(true)`
- N’empêche pas la JVM de s’arrêter

- Voir exemple `Demon.java`

Problèmes de concurrence ?

- Voir l'exemple `Exemple2Thread.java`
- Que se passe-t-il ?
- On peut résoudre le problème en *verrouillant* certaines méthodes de la classe pour empêcher son exécution par un autre `Thread` concurrent. Nous avons mis un *mutex*.

- Voir `Sync.java`

Protection des attributs entre threads

- Utilisation de `private` (sans setter) et utilisation de `final`

Méthodes deprecated

- stop()
- interrupt()
- resume()

/!\ Ces arrêts sont trop brutaux et peuvent engendrer des incohérences (voir cours MIBD)

- Que faire ?
  - Utiliser une variable *volatile* (= qui est mise à jour rapidement, et non cachée par le CPU) que le processus vérifie continuement et déclanche un arrêt “contrôlé”.

Attente de Threads : méthode join

- La méthode `join` d'un `Thread` attend jusqu'à ce que ce `Thread` soit terminé.
- Notons que d'autres choses peuvent se passer entretemps (exécution d'autres instructions d'autres threads en particulier)

- Voir exemple : `ExempleJoinThread.java`

Envoi d’informations à un Thread : méthode interrupt()

Priorité d’un Thread

- Voir SynchAvecArret.java en rajoutant la priorité (ne “force” rien …)

Synchronization entre Threads

- La classe Object contient les méthodes wait(), notify() et notifyAll() pour permettre de synchroniser des threads grâce à l'envoi de messages. Ces méthodes permettent la mise en oeuvre d'un mécanisme de communication par échanges de messages visant à synchroniser l'exécution de threads.
- La méthode wait() met le thread courant en attente jusqu'à ce que l'objet reçoive une notification par les méthodes notify() ou notifyAll() : cette attente peut donc être potentiellement infinie.
- La méthode wait() possède deux surcharges :
  - wait(long *timeout*) : attend au plus la durée en millisecondes fournie en paramètre
  - wait(long *timeout*, int *nanos*) : attend au plus la durée en millisecondes cumulée avec celle en nanosecondes fournies en paramètres
- La méthode notifyAll() avertit tous les threads dont les méthodes wait() de la même instance sont invoquées.
- La méthode notify() avertit un des threads dont la méthode wait() de la même instance est invoquée.
- Il est important que les méthodes wait() et notifyAll() ne soient invoquées que par le thread qui possède le verrou sur le moniteur de l'instance.

Un cas classique d'utilisation de la synchronisation de threads est la mise en oeuvre du modèle de conception producer/consumer.

Voir `ExempleProducerConsumer.java`

Groupe de Threads

- A voir en TD