Programmation concurrente en Java

Licence Professionnelle
Iut d'Aubière
Salva sébastien

1

Prelude

- Processus en Java
- Runtime runtime = Runtime.getRuntime(); runtime.exec(**new** String[] {"app.exe" });
  - Exec retourne un objet Process qui fournit
    - InputStream et OutputStream
    - waitFor()
- Mais aussi: ProcessBuilder (Process avec args), Desktop (navigateur, mail)

2

Plan

- Threads,
- Techniques de synchronisation (synchonize, join, notify)
- Autres techniques (semaphores, barrier, pool de threads,etc.)
- Collections
- Présentation de quelques paradigmes de programmation

Qu'est ce que la parallélisation

- Technique d'optimisation pour obtenir des résultats plus rapidement
- Utilisation de plus d'un processeur
  - Gestion de plusieurs flots d'exécution
- Le temps d'exécution devrait diminuer mais le temps cpu augmente

Qu'est ce qu'un thread ?

- Un thread consiste en une série d'instructions avec son propre compteur de programme, sa propre pile et identificateur (voir cours système précédent).
  - (le reste est partagé entre les threads d'un même processus)
- Un programme // exécute des threads en //
- Ces threads sont ensuite en file d'attente sur les processeurs

Thread en Java

- un thread est une instance de la classe Thread qui implémente l'interface Runnable
- On a toujours un Thread =>celui de la méthode main
- Un Thread qui n'est plus référencé n'est pas détruit car il est enregistré par un contrôleur d'exécution.

Thread en Java

- 2 façons pour créer des Threads:
- 1ère façon :
  - utiliser le constructeur Thread(Runnable) de la classe Thread
  - créer un Runnable
  - le passer au constructeur de Thread
- 2ème façon : créer une instance d'une classe fille de la classe Thread

Thread en Java

- Interface Runnable (1ère façon)
  - La classe doit implémenter l'interface Runnable

```
public interface Runnable {
void run();
}
```

Méthode qui contient le code exécuté par un Thread

Thread en Java

- Interface Runnable (1ère façon)

```
public class Voiture implements
Runnable{

private String modele;
public Voiture(String modele){
        this.modele = modele;}
public void run(){
        try{
        System.out.println(modele + "
roule");
        Thread.sleep(5000);
        }
catch(InterruptedException e){
System.out.println(e.getMessage());}
}}
```


Thread en Java

- Interface Runnable
- Lancement:

```
Thread t1 = new Thread(new Voiture("ford"));
Thread t2 = new Thread(new Voiture("volwagen"));

Voiture v = new Voiture("ford");
Thread t1 = new Thread(v);
Thread t2 = new Thread(v);

T1.start();
T2.start();
```

Ici sur même instance d'objet

Thread en Java

- Classe fille à Thread (2ème façon)

```
public class Voiture extends Thread{
private String modele;
public Voiture(String modele){
        this.modele= modele;
        }

public void run(){
try{
        System.out.println(modele + " roule");
        Thread.sleep(5000);
}}
}
catch(InterruptedException e){System.out.println(e.
getMessage());}
```

- Lancement:

```
Voiture v1 = new Voiture("ford");
Voiture v2 = new Voiture("wolvagen");
V1.start();
V2.start();
```


Thread en Java

- Choix ?
- Méthode 2 plus simple à écrire (certes) mais:
- Si la classe qui contient la méthode run() doit hériter d'une autre classe, utiliser la méthode 1
- Il est aussi plus simple d'utiliser la 1ère méthode pour partager des données entre plusieurs threads

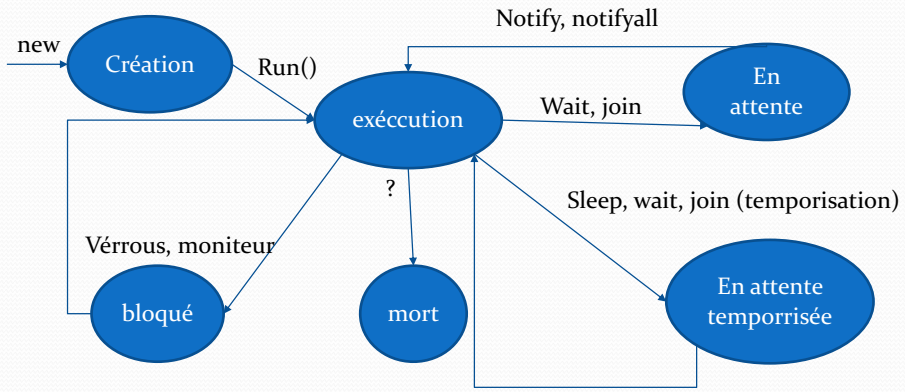
Thread en Java

- D'autres possibilités si code source est court

```
Thread t = new Thread() {
. . .
public void run() {

...code  ...

}
};
t.start();
```

```
new Thread(
new Runnable() {
. . .
public void run() {

. . .code...

}
});
```


Cycle de vie d'un thread en Java



Attention: destroy, resume, suspend et stop sont utilisable (deprecated !) mais peuvent produire de l'interblocage ou des terminaisons incorrectes : **A banir**

Partage de données entre instances de Thread

- Chaque thread peut avoir sa propre instance d'une donnée.
- Pour partager une donnée:
  - Recours à une variable de classe (ex. plus loin)
  - Ou passage instance objet à partager dans les constructeurs (ex. plus loin)
  - Ou initialisation de Threads avec même instance de Runnable (pred.)

Méthodes de Thread

- Nom du thread courant:
  - Thread.currentThread().getName()
  - Très utile pour repérer les threads
- Lancement d'un thread
  - T.start()
  - (et pas la méthode run()!)

Méthodes de Thread

- Destruction d'un thread
  - intervient quand la méthode run() se termine.
  - (pas de stop ou destroy donc)
- Interruption
  - Méthode interrupt() qui crée une *InterruptedException* si le thread est en attente.

Méthodes de Thread

- Relancer un thread?
  - On ne peut relancer un thread qui a déjà été lancé
  - Si l'exécution de la méthode run() du thread n'est pas encore terminée, on obtient une java.lang.IllegalThreadStateException
  - Si elle est terminée, aucune exception n'est lancée mais rien n'est exécuté

Méthodes de Thread

- Mise en sommeil:
  - Thread.sleep(long millis)
  - Si cette méthode est appelée dans un bloc *synchronized* le thread ne perd pas le moniteur (voir plus loin)
  - État du thread ? getState()
- Attention : isAlive() retourne vrai si le thread est actif ou endormi

Méthodes de Thread

- Mise en sommeil:

```
Public class TestSleep extends Thread{
public TestSleep(String name){super(name);}
public void run(){
try{
        System.out.println(this.getName()+" a ete
lance");
        Thread.sleep(100);
        System.out.println(this.getName()+" est
termine");}
catch(InterruptedException e){
        System.out.println(this.getName()+" a ete
interrompu");
}}
```

```
public static void main(String arg[]){
try{
        TestSleep st = new TestSleep("sasa");
        st.start();
        Thread.sleep(100);
        st.interrupt();
}
catch(InterruptedException ie){}
}}
```

Commentaires ?
Sur les threads, la synchro?

Méthodes de Thread

- Priorité:
  - Chaque Thread a une priorité définissant sa priorité à l'exécution (ordonnanceur)
  - Possibilité de récupérer et de modifier la priorité
    - setPriority(int i)
    - Int getPriority()
  - méthode yield() suspend l'exécution du thread courant pour donner la main à un autre.
    - l'appel de cette méthode peut redonner la main au thread courant

Autre

- Méthode setDaemon(boolean on) appelée avant start() permet de faire du thread un démon, processus de basse priorité qui tourne en tâche de fond
- Méthode currentThread, placée dans une méthode de n'importe quelle classe, retourne l'objet Thread qui modélise le thread courant

Synchronisation

Mots clés synchronized, join, notify

Pb de dépendance des données

- For (i=0, i<n, i++)
  a[i]=a[i+M]+b[i]

⇒Si M=0, on peut paralléliser, si M=1 ?

⇒Que faire si on ne connaît rien sur M ?

Synchronisation en Threads

- Comme pour les processus, il est nécessaire de synchroniser des threads (dans un même processus, donc JVM) lorsque des données sont partagées
  - Exclusion mutuelle
- Synchronisation sur terminaison de thread
- Moniteurs

Synchronisation en Threads

- Synchronisation sur terminaison de thread: méthode join()

```
public class exempleConcurrent extends Thread {
    private static int compte = 0;

  public void run() {
        int tmp = compte;
        try {
            Thread.sleep(100); // ms
        } catch (InterruptedException e) {
            System.out.println("ouch!\n");
            return;}
        tmp = tmp + 1;
        compte = tmp;
```

```
public static void main(String args[]) throws
InterruptedException {
        Thread T1 = new exempleConcurrent();
        Thread T2 = new exempleConcurrent();
        T1.start();
        T2.start();
        T1.join();
        T2.join();
    System.out.println( "compteur=" + compte );
  }
}
```

Attente fin T1

Attente fin T2

Commentaires ?

Moniteurs

- D'un point de vue général, Un moniteur =
- module exportant des procédures permettant l'exclusion mutuelle
- Moniteur passif, ce sont les processus ou threads qui invoquent ces procédures
- 2 procédures principales générales: wait, signal
- (en java wait, notify)

Moniteurs

- Moniteur en java
- Objet « verrou » qui ne laisse l'accès à une ressource partagée que par un seul thread
  - (contient var. d'état, méthodes d'accès, méthode de synchro, des conditions (hoare74))
- En Java, tout objet peut jouer le rôle de moniteur.
- On pose un moniteur sur un bloc de code à l'aide du mot clé *synchronized*

```
synchronized(objetMonitor){
... //code en section critique
}
```

Moniteurs

- un thread n'accède à la section critique que si le moniteur est disponible
  - un thread qui entre en section critique bloque l'accès au moniteur
  - un thread qui sort de section critique libère l'accès au moniteur
- Sleep ne fait pas perdre le moniteur (contrairement à wait)
- Attention: si le moniteur est remplacé dans la section critique, la synchronisation n'est plus garantie.

```
synchronized(maListe) {
maListe = new ArrayList<String>();
}
```


Moniteurs

- Méthodes synchronisées (sections critiques)

on peut déclarer qu'une méthode est en section critique sur le moniteur *this*

```
synchronized void methode()
{
//section critique
}
```

```
void methode() {
synchronized(this) {
//section critique
}
}
```

Moniteurs

- Méthodes synchronisées (sections critiques)

```
public class Counter{
long count = 0;

public synchronized void add(long value){
this.count += value;
  }
}
```


Moniteurs

- Si 2 threads doivent être en exclusion mutuelle sur une même ressource, ils doivent se synchroniser sur le même moniteur (logique…)
- Les sections critiques obtenues par *synchronized* doivent être les plus petites possibles (performance)
- Un constructeur ne peut pas être synchronised (exécuté par un seul thread pour la création d'un objet)
- Attention aux interblocages si utilisation de plusieurs moniteurs

Synchronisation sur moniteur

- 2 méthodes permettent d'effectuer des synchronisations de threads
  - Wait()
  - Notify()

- Si *objet* est l'objet moniteur (verrou), alors
  - Le thread qui appelle **wait** ou **notify** doit posséder *objet*

  - le thread qui appelle la méthode **objet.wait()** perd le moniteur et attend

Synchronisation sur moniteur

- Si *objet* est l'objet moniteur (verrou), alors
  - Le thread redevient actif dans les cas suivants:

    - si la méthode **objet.notify()** est appelée et qu'il est choisi parmi les threads du wait-set (activation d'un des threads du wait-set, sélection plus ou moins aléatoire)

    - si la méthode **objet.notifyAll()** est appelée (activation de tous les threads du wait-set)

    - si la durée spécifiée pour le **wait** est écoulée (cas où **wait(timeout)** est appelée)

Le producteur / Consommateur en Java

- Un ou des producteurs ajoutent des objets dans une file (FIFO).
- Un ou des consommateurs enlèvent ses objets de la file.
- Implémentation par une Arraylist() liste

Le producteur / Consommateur en Java

- 1er problème: il faut gérer les accès concurrents à la liste
- 2eme problème: si la liste est vide, un consommateur doit attendre qu'un producteur notifie l'ajout d'un élément dans la liste-> synchro
- Solution:
  - Les consommateurs se mettent en attente à chaque fois que la liste est vide
  - Les producteurs font une notification aux threads en attente lorsqu'ils ajoutent un objet

Le producteur / Consommateur en Java

```
public class FIFO {
        private ArrayList liste = new ArrayList();

public synchronized void poser(Object obj) {
this.notify();
liste.add(obj);}

public synchronized Object prendre() {
while(liste.size()== 0) {
        try {this.wait();}
catch (InterruptedException exc) {} }
Obj ret=liste.get(0) ;
Liste.remove(0); return ret; }
```


Le producteur / Consommateur en Java

Remarques:

- Bloc synchronized englobe tout le code concurrent (size et get(o)). De même, notify avant le add n'est donc pas incorrect.
- wait() peut etre interrompu (méthode interrupt()) et exige donc d'être placé dans un bloc try-catch.

Le producteur / Consommateur en Java

Remarques:

- Le thread qui exécute wait() est mis dans un pool d'attente ;
- quand il en sort il "revient" dans le bloc synchronized et entre en compétition avec d'autres threads pour acquérir le verrou.
- Il n'est donc pas certain que la condition du notify soit encore remplie au moment ou le thread acquiert le verrou :
- le test sous forme while est obligatoire.

Le producteur / Consommateur en Java

- Exemple :

```
public class File extends ArrayList<Object>{

public synchronized void Ajouter(Object o){
        System.out.println("Ajout de " +
        o.toString());
        this.add(o);
        if(this.size()>=1){
                this.notify();

        }
}
```

```
public synchronized Object Prendre(){
        while(this.size()==0){
        try{ this.wait(); }
        catch(InterruptedException e){}
Object o = this.get(0);
System.out.println("Prise de  " + o.toString());
this.remove(o);
return o;
}}
```

Le producteur / Consommateur en Java

```
Public class Prod extends Thread{
        private File f;
        private int cpt;

        public Prod(File f){ this.f = f; this.cpt = 0;
}

public void run(){
try{
while(this.cpt < 20){
                f.ajouter("element " + this.cpt);
                Thread.sleep(1000);
                this.cpt ++; }
catch(InterruptedException e){}
}}
```

```
Public class Cons extends Thread{
private File f;

   public Cons(File f){
      this.f = f;
}
   public void run(){
      while(true){
            f.prendre();

            }
}}
```


Moniteurs

- Inconvénients
  - un thread en attente d'un moniteur est bloqué
  - Impossible d'abandonner après un certain temps
- Solutions
  - Verrous (lock), Sémaphores, Futures, etc.
  - Diverses collections concurrentes

Les collections

43

Collections synchronisées

- De base une collection pose des problèmes dans le cas de partage entre Threads
  - Nécessité d'atomicité (écriture)
  - Consistence (valeur partagée doit être la même à tout instant)
- Solution de base
  - Collections non thread safe : utiliser Synchronized (verrous, sépaphore, etc.)
    - Arraylist, linkedlist, Hashset, Treeset, etc.
    - Map, hashmap, treeMap
  - Mot clé *Volatile* pour visibilité <> synchronized (pas de blocage mais consistence)

44

Collections synchronisées

- Collections *thread safe*
- Un objet immuable (final) est toujours thread-safe.
- Collections qui utilisent des verrous implicites
  - Facilitent le travail (encore fau-il bien lire la doc !)
  - Déclarées par *threadsafe*
- Généralement (un peu) moins performant (synchronisation coute du temps)
- A utiliser impérativement avec des threads
  - Attention aux threads cachés !

Collections synchronisées

- Exemple: Vectors<E>
- Méthode:
  - boolean add(E elt) : ajoute elt en fin de tableau,
  - boolean add(int i,E elt) : ajoute elt en position i
  - E get(int index) : renvoie l'élément en position i du tableau
  - boolean remove(E elt) : supp. la première occurence de elt et retourne vrai s'il y en a une
  - E remove(int i) : supp. l' élément en position i et retourne celui-ci.

Collections synchronisées

Stack<E> extends Vector<E>: pile

- E peek(): renvoie le premier élément de la pile sans l'enlever,
- E pop(): renvoie le premier élément de la pile et l'enlève de celle-ci,
- E push(E elt): met elt sur la pile et retourne elt

Hashtable<K,V>: table de hachage

- V get(Object clef): renvoie l'objet indexé par clef,
- V put(K clef, V valeur): associer valeur à clef dans la table.

Collections synchronisées

- Interface BlockingQueue
  - Implémentations de files d'attente bloquantes (i.e. les accès bloquent si la file est vide ou pleine)
  - ArrayBlockingQueue<E> file de taille bornée. Méthodes put (avec attente), offer (sans attente et sans ajout si pleine), take, poll(take non bloquant si vide)
- Mais aussi
  - LinkedBlockingQueue<E>
  - SynchronousQueue<E>
  - ConcurrentLinkedQueue<E>

Variables Atomiques

- **Package java.util.concurrent.atomic**

- AtomicBoolean,
- AtomicInteger,
- AtomicLong,
- AtomicReference,
- AtomicLongArray,
- AtomicReferenceArray.

- Accès: get, set (et autres voir doc)

Synchronisation avancée

Sémaphores en Java

- **java.util.concurrent.Semaphore**
- **Semaphore(int i)** crée un sémaphore avec un compteur initial égal à i
- la méthode **acquire() = P**
  - Permet de diminuer le compteur et bloque le thread demandeur si compteur <0 jusqu'à ce que le compteur passe à >=0 ou que le thread soit interrompu.
  - **acquire(int)** permet de requérir plusieurs permis.
- la méthode **release() = V**
  - augmente le compteur de 1. (**release(int i)** augmente le compteur de i.
- la méthode **tryAcquire()** n'est pas bloquante.

Sémaphores

```
public class MonThread extends Thread{
private String s; private Semaphore sem; public

MonThread(String s, Semaphore sem){this.s = s;this.sem = sem;}
public void run(){ try{
            sem.acquire();
            System.out.println(s+" P compteur= "+sem.availableCompt());
            Thread.sleep(10);
            sem.release();
System.out.println(s+" fin "+sem.availableCompt()+" restants)");
}
catch(InterruptedException e){System.out.println(e.getMessage());}
}}

public static void main(String[] a){
Semaphore sem = new Semaphore(3);
            for(int i = 0;i<5;i++){ MonThread mt = new MonThread("toto"+i,sem); mt.start(); }
}
```

Les verrous

- **java.util.concurrent.locks.Lock** permet de réaliser des moniteurs complexes.
- Les **Lock** permettent une synchonisation plus souple, mais plus exigante que les blocs **synchronized**.
  - il ne faut pas oublier d'ouvrir le verrou après l'avoir utilisé.
  - Il est conseillé de mettre le code en section critique dans un **try/finally**, pour être sur de déverrouiller le **Lock** dans tous les cas.

Les verrous

- Des objets **Condition** peuvent être ajoutés au verrou pour faire de la synchronisation sur plusieurs verrous croisés.
- Peut devenir (trop) complexe et (trop) difficile à tester

Les verrous

Schéma de fonctionnement:

```
Lock l = new ReentrantLock();
l.lock();
try{ //section critique }
finally{
        l.unlock();
        }
```

possibilité de tenter d'acquérir le verrou et s'il est déjà pris, de faire autre chose : méthode **tryLock()**

Les verrous

- Les conditions sur verrous

  Remplace l'utilisation des méthodes *wait, notify* du moniteur.

- Méthodes :
  - void await() : provoque le blocage du thread appelant jusqu'`a réception d'un signal ou bien d'une interruption.
  - void await (long time, TimeUnit unit) : provoque le blocage du thread appelant jusqu'`a réception d'un signal, d'une interruption ou bien le temps unit est dépassé.
  - void signal() : réveille un thread bloqué.
    void signalAll() : réveille tous les threads bloqués.

Les verrous

Exemple:

```
class BoundedBuffer {
Lock lock = new ReentrantLock();
Condition notFull = lock.newCondition();
Condition notEmpty = lock.newCondition();

Object[] items = new Object[100];
int putptr, takeptr, count;

public void put(Object x)throws IE {
lock.lock();
try {while (count == items.length)
            notFull.await();
items[putptr] = x;
if (++putptr == items.length)
            putptr = 0; ++count;
notEmpty.signal();
}
finally {lock.unlock();} }
```

```
public Object take() throws IE {
lock.lock();
try {
            while (count == 0)
                        notEmpty.await();
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0;
            --count;
            notFull.signal();
            return x;
}
finally { lock.unlock(); }
```


Les barrières

- Mécanisme permettant de bloquer un thread en un point jusqu'à ce qu'un nombre prédéfini de threads atteigne également ce point.
- Barrière CountDownLatch : Attente sur compteur à décrémenter
  voir exemple plus loin
- Barrière cyclique: CyclicBarrier(int n) : barrière pour n threads,

Les barrières

- Barrière cyclique: CyclicBarrier(int n) : barrière pour n threads,
- CyclicBarrier(int N, Runnable barrierAction)

- int await() : attendre que tous les threads aient atteint la barrière ou
  - que l'un des threads en attente soit interrompu
  - l'un des threads en attente atteigne son délai d'attente
  - la barriere soit réinitialisée
  - La valeur de retour = le nombre de threads restant à attendre.

- void reset() : réinitialise la barrière , i.e. les éventuels threads en attente reçoivent BrokenBarrierException

Les barrières

Barrière initialisée une seule fois

CyclicBarrier barriere = new CyclicBarrier (int N); …

```
// pour chacun des N threads
try {barriere.await();}
catch (InterruptedException ex) {return ;}
catch (BrokenBarrierException ex) {return;}

 // réinitialisation de la barrière
barriere.reset();
```

Les barrières

Java Doc :http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html

```
class Solver {
final int N;  final float[][] data; final CyclicBarrier barrier;

class Worker implements Runnable {
int myRow;
Worker(int row) { myRow = row; }

public void run() {
while (!done()) {
        processRow(myRow);
        try { barrier.await(); }
        catch (InterruptedException ex) { return; }
        catch (BrokenBarrierException ex) { return; } } } }
```


Les barrières

```
public Solver(float[][] matrix) { data = matrix; N = matrix.length;
Runnable barrierAction = new Runnable() { public void run() { mergeRows(...); }};
barrier = new CyclicBarrier(N, barrierAction);

List<Thread> threads = new ArrayList<Thread>(N);
for (int i = 0; i < N; i++) {
Thread thread = new Thread(new Worker(i));
threads.add(thread);
 thread.start(); }

 // wait until done
for (Thread thread : threads) thread.join(); } }
```

Chaque worker fait une ligne de la matrice et attend à la barrière
Lorsque toutes les lignes sont faites, le runnable donné à la barrière s'exécute

Tubes entre threads

- Java fournit des classes permettant de créer des tubes de communication entre threads.

- Deux classes :
  - PipedWriter pour l'entrée du tube (écriture)
  - PipedReader pour la sortie du tube.

Tubes entre threads

- Mise en place :

```
PipedWriter out = new PipedWriter();
PipedReader in;
try {in = new PipedReader(out);} catch (IOException e) { ... }
```

- PipedWriter est une sous-classe de Writer, et possède donc une méthode write
- PipedReader, sous-classe de Reader, a diverses méthodes read permettant de lire un ou plusieurs caractères depuis le tube.
- Le programme crée ensuite des threads en leur donnant l'extrémité du tube (l'objet in ou out) dont elles ont besoin.

Tubes entre threads

```
public class PipedIO {

public static void main(String[] args) throws Exception {
   Sender sender = new Sender();

   Receiver receiver = new Receiver(sender);

   sender.start();
   receiver.start();
   new Timeout(4000, "Terminated");
 }
}
```


Tubes entre threads

```
class Sender extends Thread {
  private Random rand = new Random();
  private PipedWriter out = new PipedWriter();
  public PipedWriter getPipedWriter() { return out;
}
  public void run() {
    while(true) {
      for(char c = 'A'; c <= 'z'; c++) {
        try {
          out.write(c);
          sleep(rand.nextInt(500));
        } catch(Exception e) {
          throw new RuntimeException(e);
        }}}}
```

```
class Receiver extends Thread {
  private PipedReader in;
  public Receiver(Sender sender) throws IOException {
    in = new PipedReader(sender.getPipedWriter());
  }
  public void run() {
    try {
      while(true) {
        // Blocks until characters are there:
        System.out.println("Read: " + (char)in.read());
      }
    } catch(IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

Classe Timer

- Timer permet de relancer des objets spécialisés Runnable = Timertask, avec délais d'attente et de répétition

```
public class TestTimer extends TimerTask{
public void run(){
try{
        System.out.println("En cours");
        Thread.sleep(10);}
catch(InterruptedException e){System.out.println(e.getMessage());}
}}

Timer t = new Timer();
t.schedule(new TestTimer(),100,200);
```

Tache démarre dans 100ms et se répète toutes les 200ms
3eme argument optionnel

Pool de Threads via Executor

- Un executor est une interface permettant d'exécuter des tâches.
- Une tâche est définie par un Runnable qui peut être lancé via un Thread

```
public interface Executor{
        void execute(Runnable command); }
```

Executor

- Possibilité de définir un pool de Thread qui prennent des tâches à exécuter
  - Schémas Producteur / consommateur
- Contraintes sur lesquelles réfléchir :
  - Ressources (attention aux partages, interblocages, etc.)
  - Choix des tâches (reception via FIFO, LIFO, priorité)
  - Nombre de threads

Executor

- Méthodes statiques:
- void execute(Runnable command) :
  - soumettre une tâche à l'exécuteur
  - L'exécution est asynchrone : une tâche est successivement soumise, en cours, terminée.

Executor

- Méthodes statiques:
- ExecutorService newFixedThreadPool(int) retourne un pool de taille fixée, rempli par ajout de threads jusqu'à atteindre la limite, les threads qui meurent sont remplacés au fur et à mesure.
- ExecutorService newCachedThreadPool() : 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 Threads sans tache pendant une minute sont détruits.

Executor

- ExecutorService newSingleThreadExecutor() renvoie un pool contenant un seul thread, remplacé en cas de mort inattendue.
  - La différence avec newFixedThreadPool(1) est que le nombre de Threads ne pourra pas être reconfiguré.
- ExecutorService newScheduledThreadPool(int) : renvoie un pool de taille fixée qui peut exécuter des tâches avec délai ou périodiquement

Executor

- L'ExecutorService propose des méthodes d'arrêt :
- void shutdown(): 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 à l'executor mais n'ayant pas débutées.

Executor

- Exemple:

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

class SimpExec {
 public static void main(String args[]) {
  CountDownLatch cdl = new CountDownLatch(5);
  CountDownLatch cdl2 = new CountDownLatch(5);
  CountDownLatch cdl3 = new CountDownLatch(5);
  CountDownLatch cdl4 = new CountDownLatch(5);
  ExecutorService es =
Executors.newFixedThreadPool(2);

  es.execute(new MyThread(cdl, "A"));
  es.execute(new MyThread(cdl2, "B"));
  es.execute(new MyThread(cdl3, "C"));
  es.execute(new MyThread(cdl4, "D"));

try {
   cdl.await();
   cdl2.await();
   cdl3.await();
   cdl4.await();
  } catch (InterruptedException exc) {
   System.out.println(exc);
  }

  es.shutdown();
 }
}
```

DOC: A CountDownLatch is initialized with a given count. The await methods block until the current count reaches zero due to invocations of the countDown()

Executor

- Exemple:

```
class MyThread implements Runnable {
  String name;
  CountDownLatch latch;

  MyThread(CountDownLatch c, String n)
{
    latch = c;
    name = n;
}
```

```
public void run() {
   for (int i = 0; i < 5; i++) {
    latch.countDown();
   }}}
```


Executor

- Autre alternative: Tâches avec Résultats : **Callable et Future**
- Callable équivalent de Runnable mais permettant de renvoyer une valeur. Méthode run() -> call() throws Exception
- Pour la soumettre à un ExecutorService
  - Besoin de FutureTask:
  - new FutureTask<V>(instance de Callable); *ou bien*
  - Future<V> submit(Callable<V> tache) soumettre tâche à l'exécuteur, le Future<V> renvoyé permet de manipuler de manière asynchrone la tâche.

Executor

Méthodes de FutureTask<V> :

- V get(): renvoie la valeur du résultat associée à la tâche en bloquant jusqu'`a ce que celui-ci soit disponible,
- boolean isDone() : renvoie true si la tâche est terminée.
- boolean cancel(boolean même_en_cours) : annule la tâche si celle n'a pas encore commencée. Si même en cours, tente de l'arrêter aussi même si elle a déjà débutée,
- boolean isCancelled() : renvoie true si la tâche a été annulée.

Exemple Avec Callable et FutureTask

```
Package com.journaldev.threads;
 import java.util.concurrent.Callable;

public class MyCallable implements Callable<String> {
   private long waitTime;

   public MyCallable(int timeInMillis){
      this.waitTime=timeInMillis;
   }

   public String call() throws Exception {
      Thread.sleep(waitTime);
      //return the thread name executing this callable task
      return Thread.currentThread().getName();
   }
}
```

Executor

```
public class FutureTaskExample {

    public static void main(String[] args) {
        MyCallable callable1 = new MyCallable(1000); //2 Taches à exécuter
        MyCallable callable2 = new MyCallable(2000);

        FutureTask<String> futureTask1 = new FutureTask<String>(callable1);
        FutureTask<String> futureTask2 = new FutureTask<String>(callable2);

        ExecutorService executor = Executors.newFixedThreadPool(2);
        executor.execute(futureTask1);
        executor.execute(futureTask2);
```


Executor

```
while (true) {
        try {
            if(futureTask1.isDone() &&
futureTask2.isDone()){
                System.out.println("Done");
                //Arret executor service
                executor.shutdown();
                return;
            }
```

```
if(!futureTask1.isDone()){

            System.out.println("FutureTask1
output="+futureTask1.get()); //attente bloquante
            }

            System.out.println("Waiting for FutureTask2
to complete");
            String s = futureTask2.get(200L,
TimeUnit.MILLISECONDS); //attente non bloquante
            if(s !=null){
                System.out.println("FutureTask2
output="+s);
            }
        } catch (InterruptedException |
ExecutionException e) {
            e.printStackTrace();
        }catch(TimeoutException e){
            //do nothing
        }}}}
```

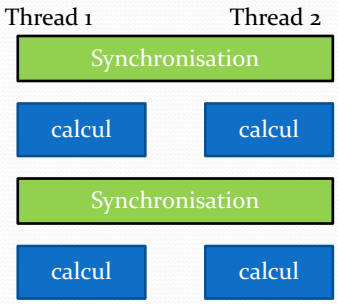
Paradigmes de programmation

81

Parallélisme de phases

- Programme = suite d'étapes
  - Classiquement: calcul et synchronisation
- Pas de recouvrement entre calcul et synchronisation
- Souvent la 1ère idée !
- => sync coutent cher en temps
- => difficile de maintenir l'équilibrage de charges

synchronisation: sem ou barrier ou créations de pool de threads, etc.



82

Maitre/esclave

- Ferme de processus ou maître/esclave
  - Un processus maître = coordinateur
  - exécute le code séquentiel
  - initie des processus esclaves
  - transmet du travail à ces processus esclaves
  - attend les résultats des esclaves
- Des processus esclaves
  - attend du travail du maître, exécute le travail
  - retourne le résultat au maître
- Paradigme simple
- Maître = goulot d'étranglement

Arbre de calcul

- Découverte dynamique d'un arbre de calcul
  - Un thread maitre divise le travail entre thread fils
  - Une fois le calcul effectué, regroupement des résultats (on remonte dans l'arbre en le supprimant éventuellement)
- Succession de division et de regroupement (on crée l'arbre de calcul puis on remonte les résultats)

Arbre de calcul

- Ex: calcul de y=(x1 op x2 op … xn)
- Op(x1,…, xn) se décompose en (x1, …,x(n/2)) et (x(n/2)+1,…, xn)
- Et ensuite ?

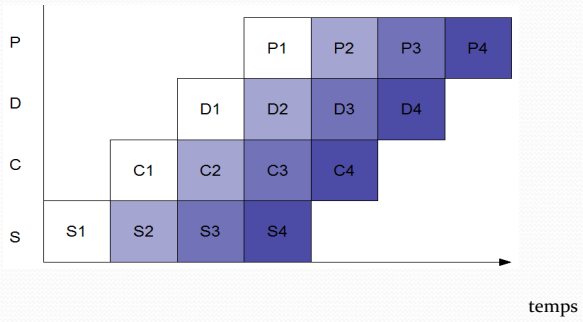
Le pipeline

- Technique employé dans les processeurs
- Plusieurs threads réalisent un pipeline virtuel
- Un flot de données parcourt le pipeline
- Recouvrement des calculs entre threads
- Et recouvrement possible entre calcul et synchronisations
- => le but est d'obtenir un pipeline équilibré sans « trous »

Le pipeline

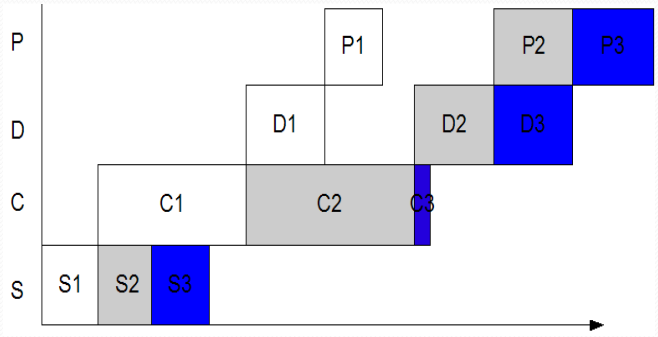
- On considère qu'une tache peut être découpée en : T= S+C+D+P (sérialisation, connexion, desérialisation, persistance)
- Le pipeline idéal

Nb étages = nb threads



Le pipeline

- Et ce que l'on peut obtenir réellement:



Conclusion

- Thread
  - Nécessaires au dvp car PC= machine parallèles (facilement 6 cœurs aujourd'hui)
  - Difficile à tester, à dvp
- API
  - De plus en plus riches
  - Pour faciliter l'utilisation, la maintenance