Programmation Concurrente

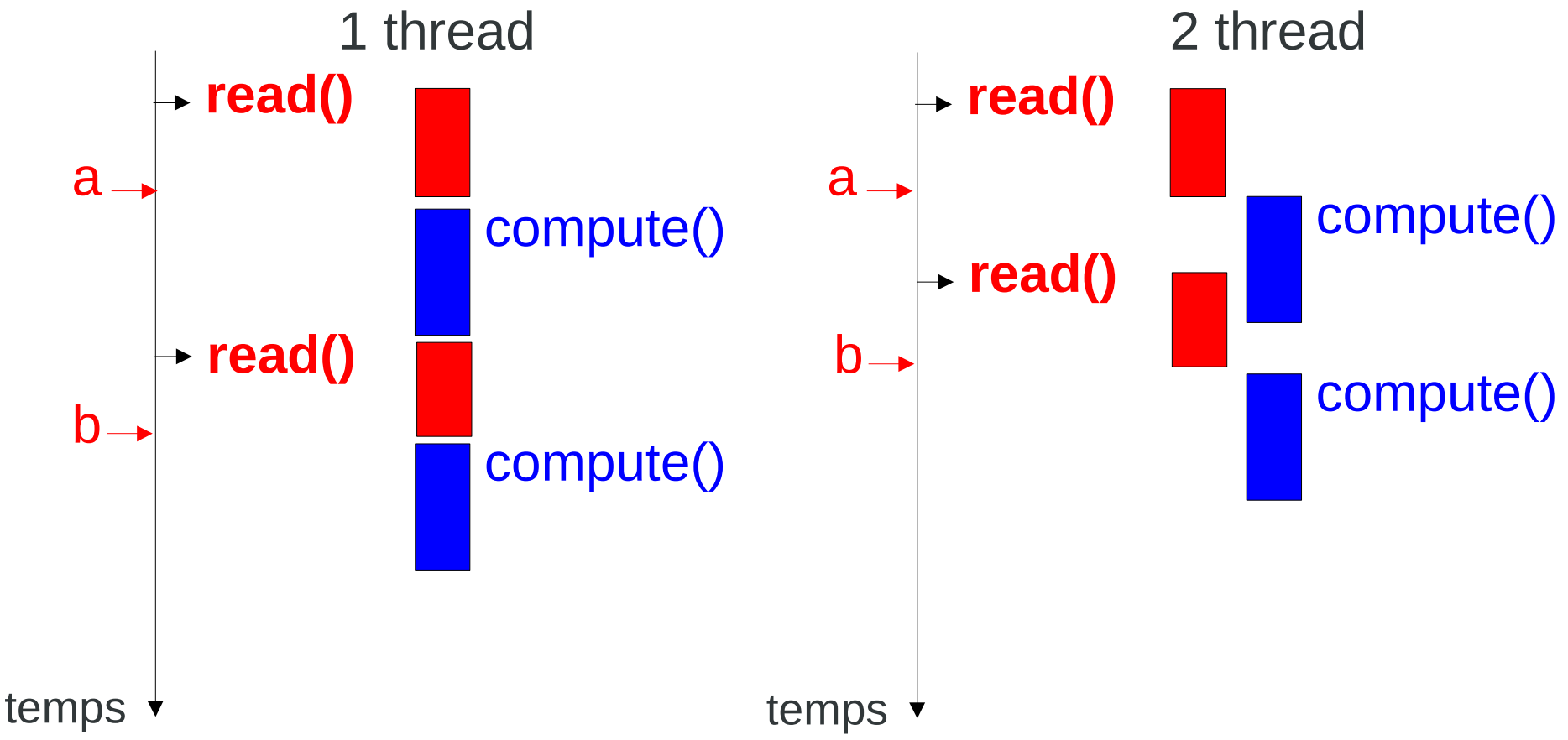
Rémi Forax
forax@univ-mlv.fr

Programmation concurrente

- Permettre d'effectuer plusieurs traitements, spécifiés distinctement les uns des autres,« **en même temps** »
- En général, dans la spécification d'un traitement, beaucoup de temps est passé à « attendre »
  - Idée: exploiter ces temps d'attente pour réaliser d'autres traitements, en exécutant **en concurrence** plusieurs traitements
  - Sur mono-processeur, simulation de parallélisme
  - Peut simplifier l'écriture de certains programmes (dissocier différentes activités)

Entrelacer calcul/entrée/sortie

- Utilisation des temps d'attente de saisie

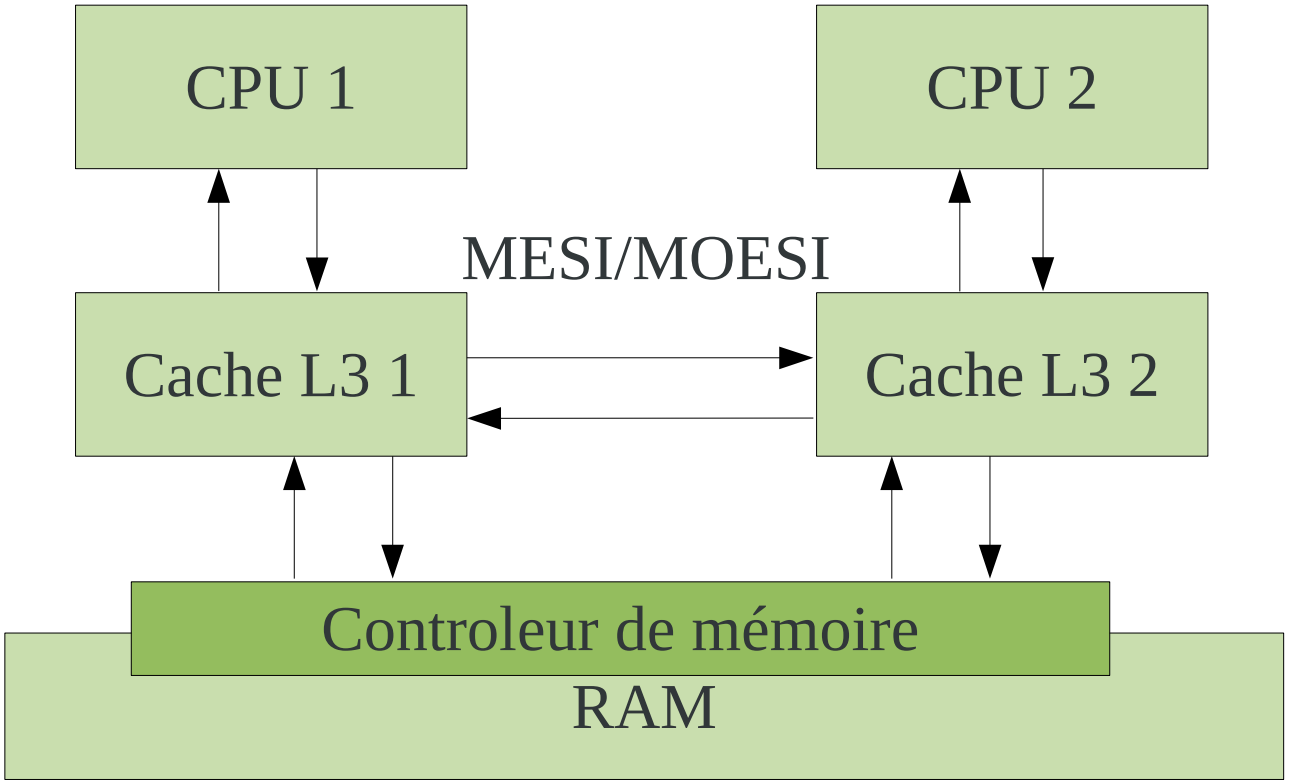


Problèmes de la concurrence

- Le programmeur n'a pas la main sur l'architecture des processeurs, des caches, ni sur le scheduler de l'OS.
  - Problèmes pour un thread qui regarde des variables modifiées par un autre thread (***visibilité***)
  - Le scheduler est preemptif
    - Il arrête les thread où il veut (opération pas atomique !)
- Les instructions des différents threads peuvent, *a priori*, être exécutées dans n'importe quel ordre.
  - Notion d'**indépendance** des traitements ?

Multi-coeur & cache

- Le cache le plus loin du proc (L3) est partagé

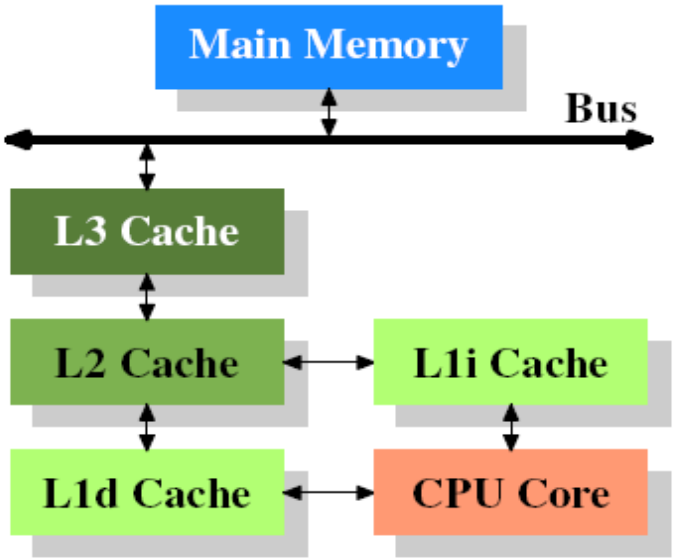


- Dé-cohérence entre le cache et la mémoire pour la lecture et l'écriture entre CPUs

Cache & cache miss

```
ld rax [rbx+16]
```

- Accès en cycle d'horloge:
  - Registre <1 clock
  - L1 ~3 clocks
  - L2 ~15 clocks
  - L3 ~50 clocks
  - RAM ~ 200 clocks



- Si donnée pas dans le cache, il faut attendre

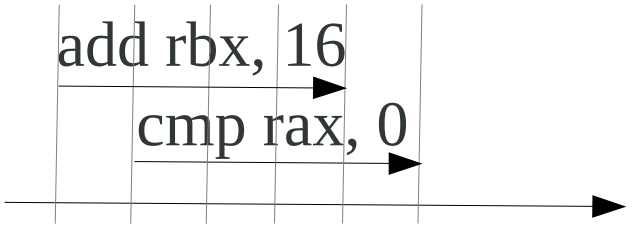
Solution au cache-miss

- Problème actuelle: *cache miss* domine le temps d'exécution
- ILP (Instruction Level Parallelism)
  - Pipelining, wide-issue
  - Prédiction de branche
  - Exécution spéculative
  - Out-Of-Order execution
- Pas utilisé sur les processeurs anciens ou les nouveaux peu gourmand en Watts (GPU ou Intel Atom)

Pipelining + wide issue

- Partage des unités entre plusieurs instructions

lecture de la mémoire +
decoding +
fetch des opérandes +
opération +
écriture de la mémoire




```
add rbx, 16       #ajoute 16 à rbx
cmp rax, 0        #compare rax à 0
```

- Wide issue: 1 clock == 2 read/2 write exécution en parallèle
- Instructions sur différents registres/endroits de la mémoire

Out of Order execution

- Example avec cache miss, out-of-order, register renaming, exécution spéculative et branch prediction

```
ld rax, [rbx+16]    # cache miss (attend 200 clocks)
add rbx, 16         # register renaming, // dispatch
cmp rax, 0          # rax pas accessible, ajoute dans load buffer
jeq null_check      # branch prediction, on continue nvelle clock
st [rbx-16], rcx    # garde rcx dans load buffer
ld rcx, [rdx+0]     # autre cache miss (pas relié)
ld rax, [rax+8]     # rax pas encore arrivé, on attend
```

- Si l'on regarde la mémoire, les effets de bords des instructions n'apparaissent pas dans le bonne ordre !

Problèmes liés au processeur/optimiser

- Le JIT
  - stocke les variables dans des registres pour éviter les aller-retour à la mémoire
    - ré-ordonnance (lui aussi) les instructions pour de meilleur performance
- Les long et double sont chargés en 2 fois sur une machine 32bit
- Problème :
  - Pas de problème visible pour la thread courante
  - Problème pour une autre thread qui regarde (accède aux mêmes variables)

Modèle d'exécution

- La garantie du modèle d'exécution :
  - Deux instructions d'un même processus léger doivent respecter leur séquencement s'il elle dépende l'une de l'autre

```
a=3;
c=2;
b=a; // a=3 doit être exécuter avant b=a
```

  - Mais pour un autre processus léger, on ne sait pas si **c=2** s'exécute avant ou après **b=a**
  - Deux instructions de deux processus légers distincts n'ont pas a priori d'ordre d'exécution à respecter.

Solutions

- Les solutions :
  - Nécessité d'assurer la **cohérence** des données
    - En imposant un contrôle sur les lectures/écritures
    - En utilisant des opérations atomique du processeur
  - Organiser les threads entre-elles, "synchronisation", rendez-vous, exclusion mutuelle, échange...
- Il n'existe pas (encore ?) de solution automatique à la concurrence, on doit utiliser un ensemble mécanisme à la main
  - Ces solutions peuvent entrainer d'autres problèmes: famine, interblocage.

Programmation Concurrente

Processus système

- Runtime, Process, ProcessBuilder
- Échange de messages par pipe
- Ordonnacement par le système

Processus léger

- Thread, Runnable, Executors (pool de threads)
- Partage de la même mémoire (tas)
- Un processus possède plusieurs threads
- Ordonnancement par le système ou la VM

Les processus

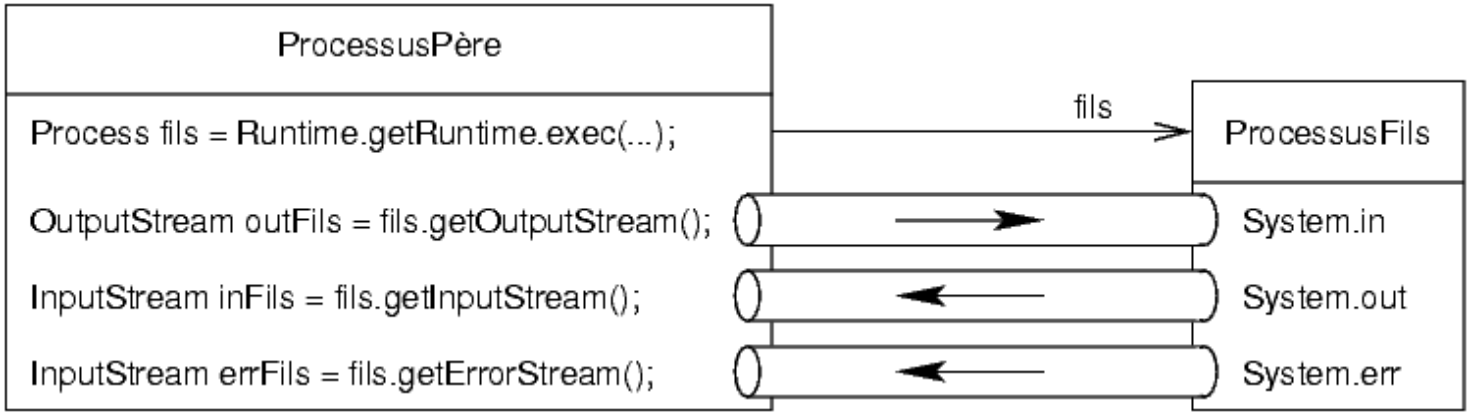
- Objet représentant une application qui s'exécute
  - java.lang.Runtime
    - Objet de contrôle de l'environnement d'exécution
      Objet courant récupérable par Runtime.getRuntime()
    - D'autres méthodes: [total/free/max]Memory(),
      gc(), exit(), halt(), availableProcessors()...
    - exec() crée un nouveau processus
  - java.lang.Process et ProcessBuilder
    - Objets de contrôle d'un (ensemble de) processus, ou commandes
    - Runtime.getRuntime().exec("cmd") crée un nouveau processus correspondant à l'exécution de la commande, et retourne un objet de la classe Process qui le représente

La classe Runtime

- Les différentes méthodes exec() créent un processus natif.
  - Exemple simple:
    ```
    Runtime.getRuntime().exec("javac MonProg.java");
    ```
  - Avec un tableau d'arguments
    ```
    Runtime.getRuntime()
          .exec(new String[]{"javac", "MonProg.java"});
    ```
  - Exécute la commande cmd dans le répertoire /tmp avec comme variable d'environnement vararіable la valeur value.
    ```
    Runtime.getRuntime().exec("cmd",
                                  new String[] {"VARIABLE=VALUE"},
                                  new File("/tmp/"));
    ```

La classe Process

- Objet retourné par méthode exec() de Runtime
  - Process fils= Runtime.getRuntime().exec("commande");
    fils.waitFor(); // attend la terminaison du processus fils
    System.out.println(fils.exitValue());
  - Toutes les méthodes sont abstraites dans Process:
    - destroy(), getInputStream(), getOutputStream(), getErrorStream()
  - Nécessité de lire et d'écrire dans les flots d'entrée/sortie



La classe ProcessBuilder

- Gère des "attributs de processus" communs
  - Commande (liste de chaînes de caractères)
    - Ex: ["javac", "-Xlint:unchecked", "Toto.java"]
  - Environnement (une Map<String, String> entre variables et valeurs)
    - Récupère par défaut les variables d'environnement du système
  - Répertoire de travail sous la forme d'un objet File
  - Propriété de redirection de flot d'erreur (vers le flot de sortie std)
    - redirectErrorStream(boolean redirect) (faux par défaut)
- Chainage des méthodes (*builder* au sens *design-pattern)*

Exemple de ProcessBuilder

- On souhaite ré-utiliser la même commande sur plusieurs répertoires différents

```
public List<Process> listAll(File... dirs) {
  ProcessBuilder pb=new ProcessBuilder("\bin\ls","-a")
  Map<String,String> env=pb.environment();
  env.put("newVariable","newValue");
  ArrayList<Process> procs=new ArrayList<Process>();
  for(File workDir:dirs) {
    pb.directory(workDir).redirectErrorStream(true);
    procs.add(pb.start());
  }
  return Collections.unmodifiableList(procs);
}
```

Processus léger

- Étant donnée une exécution de Java (une JVM)
  - **un seul processus** (au sens système d'exploitation)
  - disposer de **multiples** fils d'exécution (**threads**) internes
  - possibilités de **contrôle** plus fin (priorité, interruption...)
  - espace **mémoire commun** entre les différents threads
- Les processus légers s'exécutent en concurrence sur le nombre de processeurs disponibles
- Ils onts des piles différentes mais accède au même tas
  - => problèmes d'accès concurrent

Processus léger & concurrence

- java.util.concurrent.* existe depuis 1.5
    - Facilite l'utilisation des threads
    - implémentations spécifiques aux plate-formes/processeurs
  - Pour la gestion de l'atomicité des lectures/opérations/écriture
    - java.util.concurrent.atomics
  - Pour la gestion de la synchronisation, verrous, conditions d'attente/notification
    - java.util.concurrent.locks
  - Pour des classes utilitaires ou collection thread-safe
    - java.util.concurrent

java.lang.Thread

- La classe Thread contrôle l'exécution
  - Cohérence des valeurs, mise à jour inter-thread (volatile/final) *java memory model* (JMM, JSR 133)
  - Opérations atomiques
    - compare and set, java.util.concurrent.atomics
  - Section critique et exclusion mutuelle
    - synchronized, moniteurs, java.util.concurrent.locks
  - Point de rendez vous, attente passive
    - wait()/notify(), Condition
  - Classes de service:
    - java.util.concurrent.Executors, ThreadPoolExecutor

Threads de base

- Lorsqu'on exécute la commande `%java Prog`
  - La JVM démarre plusieurs threads, dont la thread "main"
  - La thread "main" est chargée d'exécuter le code de la méthode main()
  - Le code peut demander la création d'autres threads
  - Les autres threads servent à la gestion de la JVM (ramasse-miettes, etc).
    - "Signal Dispatcher", "Finalizer", "Reference Handler", etc.
    - on peut avoir ces informations en envoyant un signal QUIT au programme (Ctrl-\ sous Unix ou Ctrl-Pause sous Windows).
    - La commande jconsole permet de « monitorer » les programmes Java

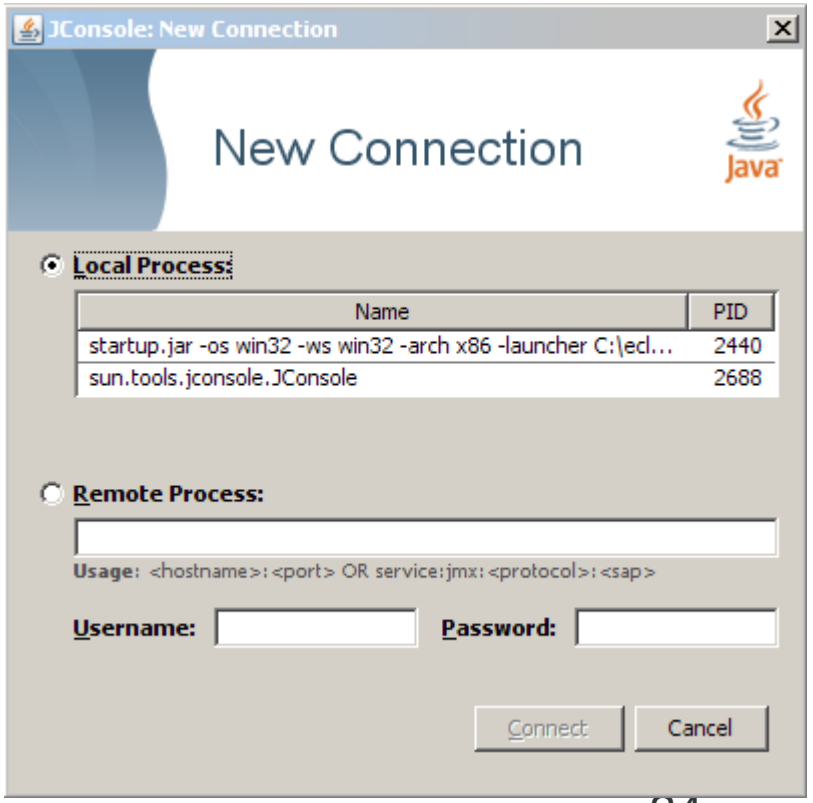
Exemple de SIGQUIT

- Programme simple (boucle infinie dans le main())
  - Taper Crtl-\ produit l'affichage suivant (extrait):
- Full thread dump Java HotSpot(TM) Client VM (1.5.0_05-b05 mixed mode, sharing):

```
"Low Memory Detector" daemon prio=1 tid=0x080a4208 nid=0x2899 runnable [0x00000000..0x00000000]
"CompilerThread0" daemon prio=1 tid=0x080a2d88 nid=0x2898 waiting on condition [0x00000000..0x45c9df24]
"Signal Dispatcher" daemon prio=1 tid=0x080a1d98 nid=0x2897 waiting on condition [0x00000000..0x00000000]
"Finalizer" daemon prio=1 tid=0x0809b2e0 nid=0x2896 in Object.wait() [0x45b6a000..0x45b6a83c]
        at java.lang.Object.wait(Native Method)
        - waiting on <0x65950838> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0x65950838> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)
"Reference Handler" daemon prio=1 tid=0x0809a600 nid=0x2895 in Object.wait() [0x45aea000..0x45aea6bc]
        at java.lang.Object.wait(Native Method)
        - waiting on <0x65950748> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:474)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
        - locked <0x65950748> (a java.lang.ref.Reference$Lock)
"main" prio=1 tid=0x0805ac18 nid=0x2892 runnable [0xbfffd000..0xbfffd4e8]
        at fr.umlv.td.test.Toto.main(Toto.java:5)
"VM Thread" prio=1 tid=0x08095b98 nid=0x2894 runnable
"VM Periodic Task Thread" prio=1 tid=0x080a5740 nid=0x289a waiting on condition
```

jconsole

- Outils de monitoring de la VM de façon externe
  - Pour la version 1.5, la VM doit être lancée par java -Dcom.sun.management.jmxremote ...
  - Pour la version 1.6, **rien à faire**
- Monitoring :
  - du CPU
  - des threads
  - de la mémoire/GC
  - du chargement des classes



La classe java.lang.Thread

- Chaque instance de la classe Thread possède:
  - un nom, [get/set]Name(), un identifiant
  - une priorité, [get/set]Priority(),
    - les threads de priorité plus haute sont exécutées plus souvent
    - trois constantes prédéfinies: [MIN / NORM / MAX]_PRIORITY
  - un statut *daemon* (booléen), [is/set]Daemon()
  - un groupe, de classe ThreadGroup, getThreadGroup()
    - par défaut, même groupe que la thread qui l'a créée
  - une cible, représentant le code que doit exécuter ce processus léger. Ce code est décrit par la méthode

    public void run() {...}

    qui par défaut ne fait rien (return;) dans la classe Thread

Thread et JVM

- La Machine Virtuelle Java continue à exécuter des threads jusqu'à ce que:
  - soit la méthode exit() de la classe Runtime soit appelée
  - soit toutes les threads non marquées "*daemon*" soient terminées.
  - on peut savoir si une thread est terminée via la méthode isAlive()
- Avant d'être exécutées, les threads doivent être créés:

  Thread t = new Thread(...);
- Au démarrage de la thread, par t.start();
  - la JVM réserve et affecte l'espace mémoire nécessaire avant d'appeler la méthode run() de la cible.

On implante Runnable.run()

- En utilisant une classe

```
class MyRunnable implements Runnable {
  @Override public void run() { /* code */ }
}
...
Runnable r = new MyRunnable();
Thread t = new Thread(r);    // création
t.start();                   // démarrage
```

- En utilisant la syntaxe des lambdas

```
Runnable r = () -> { /* code */  };
Thread t=new Thread(r);       // création
t.start();                    // démarrage
```

Ne pas hériter de Thread !!

- Il est possible d'hériter de Thread mais c'est une bétise !

```
public class Bleep {
  String name="Bleep";
  void setName(String name) {
    this.name=name;
  }
  void backgroundSetName() throws InterruptedException {
    Thread t=new Thread() {
      @Override public void run() { setName("Blat"); }  // attention !
    };
    t.start();
    t.join();  // attend la mort de t
    System.out.println(name);
  }
  public static void main(String[] args) throws InterruptedException {
    new Bleep().backgroundSetName();
  }
}
```

Thread et objet de contrôle

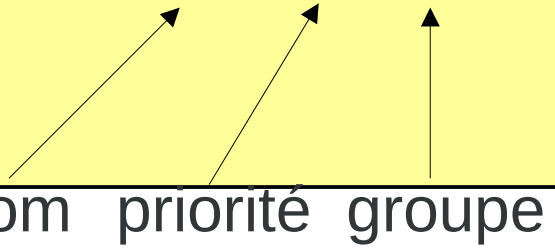
- À la fin de l'exécution de la méthode run() de la cible, le processus léger est terminé (mort):
  - il n'est plus présent dans la JVM (en tant que thread)
  - mais l'objet contrôleur (de classe Thread) existe encore
  - sa méthode isAlive() retourne false
  - il n'est pas possible d'en reprendre l'exécution
  - l'objet contrôleur sera récupéré par le ramasse-miettes
- L'objet représentant la thread qui est actuellement en train d'exécuter du code peut être obtenu par la méthode statique Thread.currentThread()

La thread courrante

- Par défaut la thread courante s'appel **main**.

```
public class ThreadExample {
  public static void main(String[] args) throws InterruptedException {
    // Affiche les caractéristiques du processus léger courant
    Thread t = Thread.currentThread();
    System.out.println(t);
    // Donne un nouveau nom au processus léger
    t.setName("Médor");
    System.out.println(t);
    // Rend le processus léger courant
    // inactif pendant 1 seconde
    Thread.sleep(1000);
    System.out.println("fin");
  }
}
```

```
% java ThreadExample
Thread[main,5,main]
Thread[Médor,5,main]
fin
%
```



Cycle de vie d'un processus léger

- Création de l'objet contrôleur: t = new Thread(...)
- Démarrage de la thread et allocation des ressources : t.start()
- Début d'exécution de run()
  - [éventuelles] suspensions temp. d'exéc: Thread.sleep()
  - [éventuels] relâchements voulus du proc. : Thread.yield()
  - peut disposer du processeur et s'exécuter
  - peut attendre le proc. ou une ressource pour s'exécuter
- Fin d'exécution de run(), eligible pour le GC

L'accès au processeur

- Différents états possibles d'une thread
  - exécute son code cible (elle a accès au processeur)
  - attend l'accès au processeur (mais pourrait exécuter)
  - attend un événement particulier (pour pouvoir exécuter)
- L'exécution de la cible peut libérer le processeur
  - si elle exécute un yield() (demande explicite)
  - si elle exécute une méthode bloquante (sleep(), wait()...)
- Sinon, c'est l'ordonnanceur qui répartit l'accès des threads au processeur.
  - utilisation des éventuelles priorités

Différents états d'un processus léger

- Depuis la 1.5, il est possible de connaître l'état d'un processus léger *via* la méthode getState(), exprimé par un type énuméré de type Thread.State :
  - NEW : pas encore démarré;
  - RUNNABLE : s'exécute ou attend une ressource système, par exemple le processeur;
  - BLOCKED : est bloqué en attente d'un moniteur;
  - WAITING : attente indéfinie de qq chose d'une autre thread;
  - TIMED_WAITING : attente bornée de qq chose d'une autre thread ou qu'une durée s'écoule;
  - TERMINATED : a fini d'exécuter son code.

Arrêt d'un processus léger

- Les méthodes stop(), suspend(), resume() sont **dépréciées**
  - Risquent de laisser le programme dans un « sale » état !
- La méthode destroy() n'est pas implantée
  - Specification trop brutale: l'oublier
- Terminer de manière douce...
- Une thread se termine normalement lorsqu'elle a terminé d'exécuter sa méthode run()
  - obliger proprement à terminer cette méthode

Interrompre une thread

- La méthode interrupt() appelée sur une thread t
  - Si t est en attente parce qu'elle exécute un wait(), un join() ou un sleep(), alors ce statut est réinitialisé et la thread reçoit une InterruptedException
  - Si t est en attente I/O sur un canal interruptible (java.nio.channels.InterruptibleChannel), alors ce canal est fermé, le statut reste positionné et la thread reçoit une ClosedByInterruptException
  - Sinon positionne un « statut d'interruption »
- Le statut d'interruption ne peut être consulté par les méthodes interrupted() et isInterrupted()

Consulter le statut d'interruption

- public static boolean interrupted()
  - retourne true si le statut de la thread actuellement exécutée a été positionné (méthode statique)
  - si tel est le cas, réinitialise ce statut à false
- public boolean isInterrupted()
  - retourne true si le statut de la thread sur laquelle est appelée la méthode a été positionné (méthode d'instance, non statique)
  - ne **modifie pas** la valeur du statut d'interruption

Exemple interrupt

```
public class InterruptionExample {
  public static void main(String[] args) throws InterruptedException {
    Thread[] threads=new Thread[5];

    for(int i=0;i<threads.length;i++) {
      final int id=i;
      Thread t=new Thread(new Runnable() {
        public void run() {
          for(int j=0;!Thread.interrupted();j++) {
            System.out.println(j + "ième exécution de " + id);
          }
          System.out.println("Fin d'exécution du code " + id);
          // interrupted() a réinitialisé le statut d'interruption
          System.out.println(Thread.currentThread().isInterrupted());
        }
      });
      threads[i]=t;
      t.start();
    }
    Thread.sleep(5);
    for(int i=0;i<threads.length;i++)
      threads[i].interrupt();
  }
}
```

Exemple interrupt (suite)

- L'exemple précédent affiche :

```
52ième exécution de 3
53ième exécution de 3
48ième exécution de 0
Fin d'exécution du code 0
false
1ième exécution de 2
Fin d'exécution du code 2
false
1ième exécution de 4
Fin d'exécution du code 4
false
0ième exécution de 1
Fin d'exécution du code 1
false
54ième exécution de 3
Fin d'exécution du code 3
false
```

currentThread.isInterrupted()
Le status d'interuption a été réinitialisé par **interrupted**()

- Certaines threads n'ont quasiment pas eut le temps de démarrées

Interrupt et InterruptException

- Si le status d'interruption est déjà positioné, un appel à une méthode bloquante sleep(), wait(), join() etc, lance directement l'exception InterruptException
- Lorsque l'exception est lancée, le status d'interruption est réinitialisée
- Il est possible de le repositionner en s'interrompant soit-même
  Thread.currentThread().interrupt()

Exemple interrupt (suite)

- Objectif: ralentir l'affichage pour que chaque thread attende un temps aléatoire
  - Proposition: écrire une méthode **slow**(), et l'appeler dans la boucle de la méthode run() :

```
public void run() {
  for(int j=0;!Thread.interrupted();j++) {
    System.out.println(j + "ième exécution de " + id);
    slow(); // ralenti l'affichage
  }
  ...
}
```

Exemple interrupt (fin)

- La méthode slow() :

```
private void slow() {
  try {
    Thread.sleep(random.nextInt(2));
  } catch (InterruptedException e) {
    // repositionne le status d'interruption
    Thread.currentThread().interrupt();
  }
}
```

- On repositionne le status d'interruption pour pouvoir le consulter plus tard

```
1ième exécution de 4
2ième exécution de 2
1ième exécution de 0
3ième exécution de 2
1ième exécution de 1
1ième exécution de 3
Fin d'exécution du code 1
false
Fin d'exécution du code 4
Fin d'exécution du code 2
Fin d'exécution du code 0
Fin d'exécution du code 3
false
false
false
false
```

Thread et champs d'objet

- Les différents threads ont des piles d'exécution et des registres propres, mais accèdent à un même tas
- Trois types champs :
  - Ceux qui sont consultés/modifiés par 1 seul thread
    - Pas de problème, même s'ils résident dans la mémoire centrale
  - Ceux qui sont **accédés en concurrence**
  - Ceux qui sont gérés **localement** au niveau d'un thread
    - Classe dédiée à ce type: **ThreadLocal,** champ partagé entre plusieurs threads mais dont la valeur est différente pour chaque thread

Le problème

- Scanner est une variable partagée !!

```
public class IOUtil {
  public void open(File file) {
    try {
      scanner=new Scanner(file);
    } catch (FileNotFoundException e) {
      throw new IllegalArgumentException(e);
    }
  }
  public String nextLine() {
    if (!scanner.hasNextLine())
      return null;
    return scanner.nextLine();
  }
  public void close() {
    scanner.close();
  }
  private Scanner scanner;
```

```
  public static void main(String[] args) {
    final IOUtil io=new IOUtil();
    for(String arg:args) {
      final File file=new File(arg);
      new Thread(new Runnable() {
        public void run() {
          io.open(file);
          int count=0;
          for(;io.nextLine()!=null;count++);
          io.close();
          System.out.println(count);
        }
      }).start();
    }
```

Variables locales à une thread

- Si plusieurs threads accède à une même variable, il peut disposer de variables locales à une thread

- ThreadLocal<T> et InheritableThreadLocal<T> permettent de simuler ce comportement
  - objet déclaré comme un champ dans l'objet partagé
  - existence d'une valeur encapsulée propre à chaque thread, accessible via les méthodes get() et set()

Une solution

- On utilise la classe ThreadLocal

```
public class IOUtil {
  public void open(File file) {
    try {
      scanner.set(new Scanner(file));
    } catch (FileNotFoundException e) {
      throw new IllegalArgumentException(e);
    }
  }
  public String nextLine() {
    Scanner s=scanner.get();
    if (!s.hasNextLine())
      return null;
    return s.nextLine();
  }
  public void close() {
    scanner.get().close();
    scanner.remove();
  }
  private final ThreadLocal<Scanner> scanner=
    new ThreadLocal<Scanner>();
```

Autre solution

- On rend l'objet non mutable :)

```
public class IOUtil {
  public IOUtil(File file) {
    try {
      scanner=new Scanner(file);
    } catch (FileNotFoundException e) {
      throw new IllegalArgumentException(e);
    }
  }
  public String nextLine() {
    if (!scanner.hasNextLine())
      return null;
    return scanner.nextLine();
  }
  public void close() {
    scanner.close();
  }
  private final Scanner scanner;
```

```
  public static void main(String[] args) {
    for(String arg:args) {
      final File file=new File(arg);
      new Thread(new Runnable() {
        public void run() {
          final IOUtil io=new IOUtil(file);
          int count=0;
          for(;io.nextLine()!=null;count++);
          io.close();
          System.out.println(count);
        }
      }).start();
    }
```

Hérité d'une variable thread local

- La classe InheritableThreadLocal<T> permet de récupérer la valeur associé à une thread mère dans une thread créer à partir de celle-ci.
  - initialisation des champs de classe InheritableThreadLocal<T> de la thread « fille » à partir des valeurs des champs de la thread «mère», par un appel implicite à la méthode T childValue(T) de la classe.
  - possibilité de redéfinir cette méthode dans une sous-classe de InheritableThreadLocal<T>.

V1: Problème des registres

- Le JIT stocke les variables dans des registres pour éviter les aller-retour à la mémoire

```
public class A implements Runnable {
  public boolean start=false;

  public void run() {
    while(!this.start) ; // attente active

    ...
  }
}
```

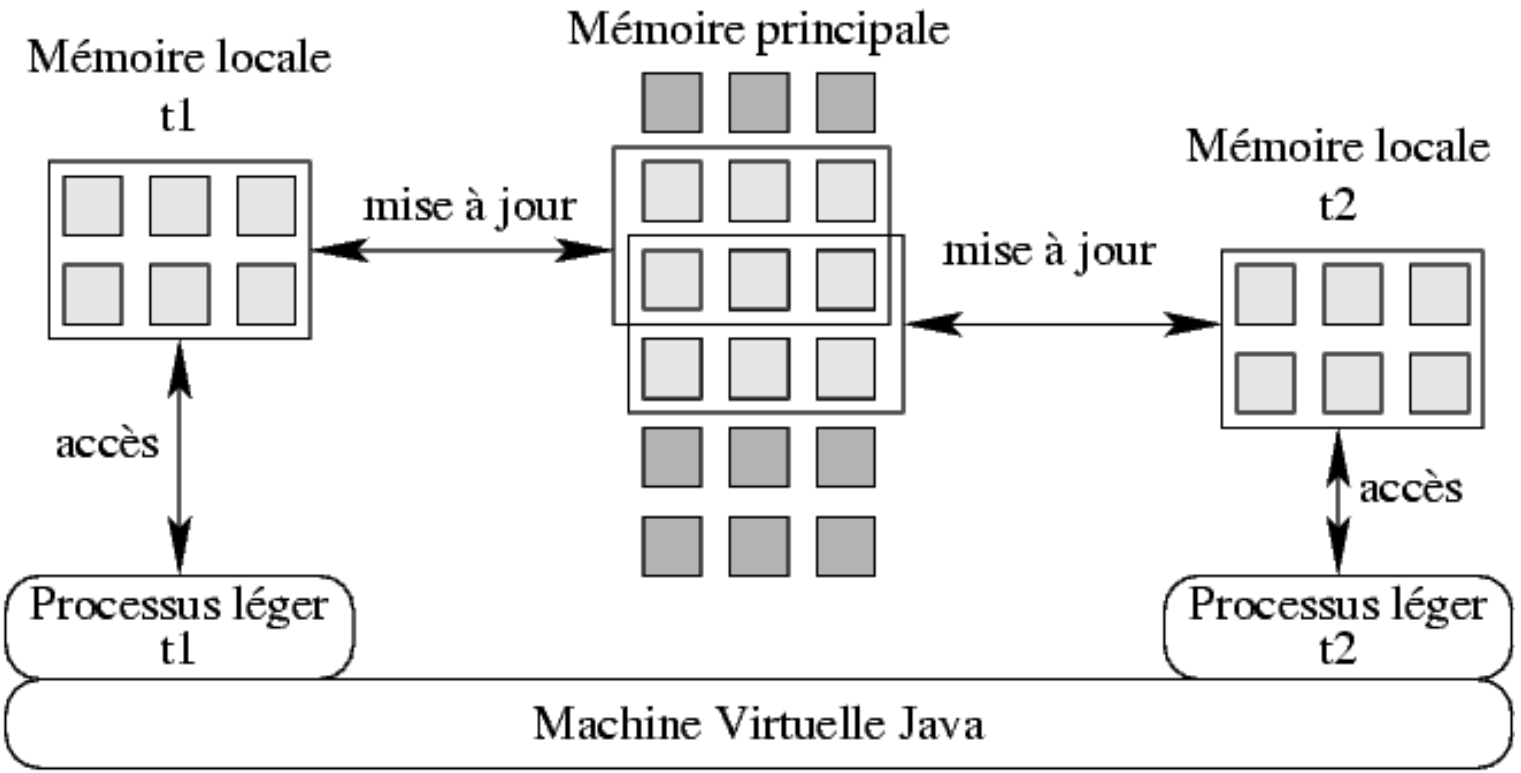
Ici start est mis dans un registre

Problème car start n'est plus rechargé de la mémoire

```
public void preinit(A a) {
  // init
  a.start=true;
}
```

Visibilité et mise à jour

- Chaque processus léger peut stocker ses valeurs dans les registres du proceseur qui lui sont allouées : la mise à jour est faite quand cela est nécessaire



V2: Problème d'ordonnancement

- Ré-ordonnacement est instruction à l'intérieur d'une thread par le JIT garantie juste qu'une variable est mise à jour avant d'être lu pour une même thread.

```
public class A implements Runnable {
  public int value;
  public boolean start=false;

  public void run() {
    while(!this.start) ; // attente active

    System.out.println(value); // affiche 0
  }
```

Réordonnancement en

```
a.start=true
value=3;
f(value);
```

Problème car value est changé après start

```
public void preinit(A a) {
  a.value=3;
  a.start=true;
  f(value);
```

Solution : le mot clé volatile

- Utilisé comme un modificateur de champ
  - oblige la cohérence entre les registres et la mémoire principale.
  - L'écriture d'une variable volatile force la sauvegarde de **tous** les registres locaux dans la mémoire.

```
public class A implements Runnable {
  public volatile boolean start=false;

  public void run() {
    while(!this.start) ; // attente active

    ...
  }
}
```

```
public void preinit(A a) {
  // init
  a.start=true;
}
```

Volatile (suite)

- Volatile et ordonnancement [JLS3]
  - L'écriture d'une variable volatile force la sauvegarde de **tous** les registres dans la mémoire.
  - Assure une barrière mémoire (barrière pour l'ordonnancement)

```
public class A implements Runnable {
  public int value;
  public volatile boolean start=false;

  public void run() {
    while(!this.start) ; // attente active

    System.out.println(value); // affiche 3
  }
```

```
public void preinit(A a) {
  a.value=3;
  a.start=true; // volatile write
```

V3: Problème des longs et doubles

- Le processeur peut charger et sauvegarder les données 64bits en deux opérations

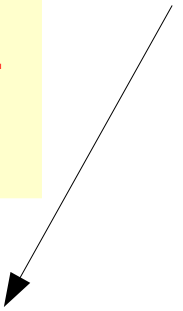
```
public class A implements Runnable {
  public long value=12;

  public void run() {
    // on attend 3 min
    System.out.println(value);
                        // affiche ni 12 ni -1
  }
}
```

Problème car l'assignation de value n'est pas une opération atomique

Scheduling au millieu de l'écriture

```
value_hi=0xffffffff;
value_lo=0xffffffff;
```



```
public void preinit(A a) {
  a.value=-1;
}
```

Volatile et long/double

- Assure de plus l'atomicité de la lecture et de l'écriture, y compris des champs de type double et long.
  - Par défaut, seules les lectures/écritures de 32 bits sont atomiques
- Attention: n'assure pas l'atomicité des opérations composites, (incrémentation/décrémentation)
  Les instructions composant ces opérations peuvent s'entrelacer entre plusieurs threads concurrentes
  - Ex: 1) Un champ x d'un objet o, déclaré volatile, vaut 0;
    2) Deux threads concurrentes font o.x++; sur le même objet o;
    3) Il peut arriver que, une fois les 2 threads exécutées,
    (o.x == 1) soit vrai! (et non 2 comme attendu)

F1: Problème de publication

- Le scheduler est **préemptif** et arrête les threads là ou il veux
- Il est possible d'accéder à la valeur d'un champs d'un objet alors que celui-ci n'a pas été initialisé.

```
public class A implements Runnable {
  public int value;
  public A(int value) {
    this.value=value;
  }
```

Correspond à

```
a=new A
a.value=3
```

\+ Scheduling au milieu

```
final A a=new A(3);
new Thread(new Runnable() {
  public void run() {
    System.out.println(a.value); // affiche 0
  }
}).start();
```

Le mot-clé final

- Les valeurs d'un champ final sont assignées dans le constructeur avant la publication de la référence sur l'objet
  - Une fois construit, le champ final peut être accédé par différentes threads sans synchronisation

```
public class A {
  public final int value;
  public A(int value) {
    this.value=value;
  }
}
```

F2: Publication unsafe de this

- Le réordonnancement à l'intérieur d'un constructeur est possible même si la variable est final !!!

```
public class A {
  public final int value;
  public A(B b,int value) {
    this.value=value;
    b.a=this; // oh oh
  }
}
```

```
public class B {
  public A a;
}
```

```
// une thread
B b=getSharedB();
A a=new A(b,3);
...
```

```
// une autre thread
B b=getSharedB();
System.out.println(b.a.value);
// peu afficher 0
```

Règles à respecter

- **Ne pas "publier"** la référence sur **this** dans le **constructeur**
  - Même si on croit que la publication est faite à la fin, rien ne l'assure (e.g. super() dans une sous-classe)
- Ne pas **exposer implicitement** la **référence this** dans le constructeur
  - Par exemple, l'exposition ou la publication de la référence à une classe interne non statique contient une référence implicite à this
- Ne pas démarrer de thread (start()) à l'intérieur d'un constructeur
  - Mieux vaut équiper la classe de sa propre méthode start() qui la démarre (après achèvement de la construction)

S1: Problème d'ordonnancement

- Le scheduler est **préemptif** et arrête les threads là ou il veux

```
public class A implements Runnable {
  public int value;
  public int value2;

  public void run() {
    ...
    System.out.println(value+" "+value2);
       // affiche 12 0
  }
```

Scheduling au millieu de l'écriture

Problème on ne choisi pas l'ordonnancement des threads

```
public void preinit(A a) {
  a.value=12;
  a.value2=24;
```

Exclusion Mutuelle/Section critique

- Écriture/lecture entrelacées provoquent des états incohérents de la mémoire
- Volatile et atomics ne peuvent rien à l'exemple précédent
- Impossible d'assurer qu'une thread ne « perdra » pas le processeur entre deux instructions atomiques
- On ne peut « que » exclure mutuellement plusieurs threads, grâce à la notion de moniteur

Les moniteurs

- N'importe quel objet (classe Object) peut jouer le rôle de moniteur.
  - Lorsqu'une thread « prend » le moniteur associé à un objet, plus aucune autre thread ne peut prendre ce moniteur.
  - Idée: protéger les portions de code « sensibles » de plusieurs threads par le même moniteur
    - Si la thread « perd » l'accès au processeur, elle ne perd pas le moniteur => une autre thread ayant besoin du même moniteur ne pourra pas exécuter le code que ce dernier protège.
  - Le mot-clef est **`synchronized`**

Solution en utilisant synchronized

- On utilise un bloc synchronized

```
public class A implements Runnable {
  public int value;
  public int value2;
  final Object lock=new Object();

  public void run() {
    synchronized(lock) {
      System.out.println(value+" "+value2);
    }
  }
```

Problème on ne choisi pas l'ordonnancement des threads

```
public void preinit(A a) {
  synchronized(a.lock) {
    a.value=12;
    a.value2=24;
  }
```

Synchronisation sur un objet

- L'objet utilisé pour l'exclusion mutuelle sert juste de point de rendez-vous, c'est un objet connu par les différentes threads qui veulent se synchronizer
- L'objet ne peut pas être **null** et ni un type primitif
- Exclusion mutelle vis à vis d'un moniteur
  - L'objet qui sert de moniteur est **inerte**
    (Il n'est pas possible est Java d'observer une différence avec un objet normal)

Choix du moniteur

- L'objet doit être connu par les deux threads mais
  - Ne doit pas être connu par d'autre pour éviter les deadlocks
- On utilise en rêgle général:
  - Une variable **privée** ou de paquetage **constante** (final)
  - Si elle n'existe pas, on en crée une
- Utiliser **this** n'est pas une bonne solution, risque d'utilisation externe du moniteur par d'autre thread

Exemple: Liste récursive

- Imaginons une liste récursive avec une structure constante (seules les valeurs contenues peuvent changer)
- Imaginons plusieurs threads qui parcourent, consultent et modifient les valeurs d'une même liste récurisve
- On souhaite écrire une méthode calculant la somme des éléments de la liste.

Une implantation possible

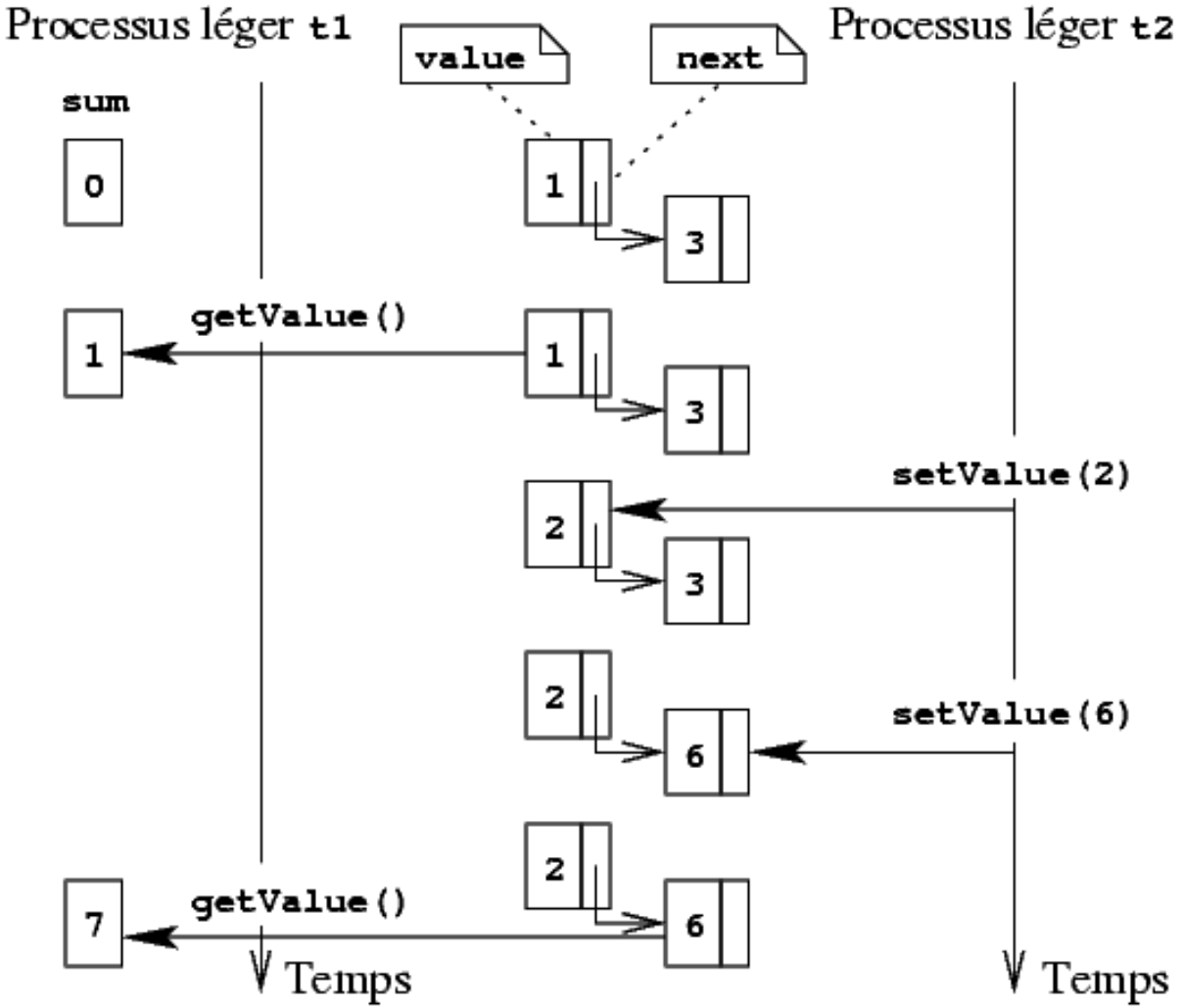
```
public class RecList {
  private double value;
  private final RecList next; // structure constante
  private final Object lock=new Object(); // protection accès à value
  public RecList(double value, RecList next){
    this.value = value;
    this.next = next;
  }
  public void setValue(double value) {
    synchronized(lock) { this.value = value; }
  }
  public double getValue() {
    synchronized(lock) { return value; }
  }
  public double sum() {
    double sum = getValue(); // pas value !!
    RecList n=next;
    if (n!=null)
      sum += n.sum();
    return sum;
  }
}
```

Mais volatile peut suffire

```
public class RecList {
  private volatile double value;
  private final RecList next; // structure constante
  public RecList(double value, RecList next){
    this.value = value;
    this.next = next;
  }
  public void setValue(double value) {
    this.value = value;
  }
  public double getValue() {
    return value;
  }
  public double sum() {
    double sum = value;
    RecList n=next;
    if (n!=null)
      sum += n.sum();
    return sum;
  }
}
```

Problème dans les deux cas

- La valeur retournée par la méthode sum() peut correspondre à une liste qui n'a jamais existé.



Solutions?

- Protéger la méthode sum() par un **moniteur propre au maillon** sur lequel on fait l'appel
  - Le moniteur sur le premier maillon est pris et conservé jusqu'au retour de la fonction et chaque appel récursif reprend le moniteur propre au nouveau maillon.

```
public double sum() {
  double sum = getValue();
  RecList n=next;
  if (n!=null) {
    synchronized(m) {
      sum += n.sum();
    }
  }
  return sum;
```

Semantique “weekly consistant”

- Les moniteurs sont donc pris récursivement :

```
public double sum() {
  double sum = getValue(); // pas value !!
  RecList n=next;
  if (n!=null) {
    synchronized(m) {
      sum += n.sum();
    }
  }
  return sum;
```

- N'empêche pas une modification en fin de liste entre le début et la fin du calcul de la somme.
- Sum est une vue "weakly-consistant" de la liste

Sémantique “snapshot”

- Pour une vue "snapshot" de la liste
  - Protéger les méthodes sur un seul et même moniteur global partagé par tous les maillons d'une même liste.
  - Chaque appel récursif **reprend le moniteur** global de la liste: les moniteurs sont **réentrants**
  - Ce moniteur sert à exclure mutuellement
    - le **calcul de la somme** de la liste et
    - les **modifications des valeurs** de la liste

Implantation snapshot

```
public class RecList {
  private volatile double value;
  private final RecList2 next;
  private final Object glock;
  public RecList(double value, RecList2 next){
    this.value = value;
    this.next = next;
    if (next==null) {
      glock=new Object();
    } else {
      glock=next.glock;
    }
  }

  public void setValue(double value) {
    synchronized (glock) { this.value = value; }
  }
  public double getValue() {
    return value; // volatile suffit
  }
```

```
  public double sum() {
    synchronized(glock) {
      double sum = value;
      RecList n=next;
      if (n!=null)
        sum += n.sum();
      return sum;
    }
  }
}
```

Comparaison

- “Weakly consistant” et” snapshot” sont deux sémantiques différentes
  - pas forcément meilleure ou moins bonne l'une que l'autre
- “Snapshot” est plus contraignant pour les autres threads et donc moins efficace mais il garantie que la liste ne changera pas pendant l'appel à sum()

Ré-entrance/héritage?

- Si une thread **t** détient un moniteur **m** alors
  si **t** exécute du code qui impose une synchronisation sur le même moniteur **m**, alors le **moniteur est pris deux fois**, il devra alors être libéré deux fois.
  On dit que les moniteurs sont **ré-entrant**.

- Il n'y a pas d'héritage de moniteur entre thread,
  si **t** crée démarre une thread **tBis**, alors **tBis** ne **détient pas** le moniteur **m**, mais entre en concurrence d'accès avec la thread **t** (**tBis** devra attendre que **t** libère **m** pour espérer pouvoir le prendre)

Synchronized comme mot clé

- Synchronized peut être utilisé comme mot-clé d'une méthode, cela revient a exécuter le code de la méthode dans un block synchronizé sur **this**

```
public synchronized double sum() {
  double sum = value;
  RecList n=next;
  if (n!=null)
    sum += n.sum();
  return sum;
}
```

```
public double sum() {
  synchronized(this) {
    double sum = value;
    RecList n=next;
    if (n!=null)
      sum += n.sum();
    return sum;
  }
}
```

- Les deux méthodes sont équivalentes

Protection en contexte statique

- Synchronized surr une méthode statique et équivalent à synchronizé sur l'objet Class correspondant à la classe.

```
public class C {
  public static synchronized  int m() {
    ...
  }
}
```

```
public class C {
  public static int m() {
    synchronized(Class.forName("C")) {
      ...
    }
  }
}
```

Tester la possession d'un moniteur

- La méthode **Thread.holdLock**(moniteur) permet de savoir si la thread courante possède ou non un moniteur

```
public class SynchronizedStack<E> {
  ...
  public boolean add(E element) {
    synchronized(array) {
      if (isFull())
        return false;
      array[top++]=element;
    }
    return true;
  }
  private boolean isFull() {
    assert Thread.holdsLock(array);
    return top==array.length;
  }
  private int top;
  private final E[] array;
```

java.util.concurrent.locks

- Paquetage disponible à partir de la version 1.5, l'interface **Lock** fourni des appels de méthode équivalentes à l'acquisition et au relachement d'un moniteur
  - L'acquisition et le relachement ne sont pas lié à un bloc
  - Deux implantations différentes :
    - **ReentrantLock** (une seul thread accède à la ressource)
    - **ReentrantReadWriteLock** (plusieurs threads en lecture, 1 seule en écriture)

Interface Lock

- Méthode permettant d'acquérir/relâcher un verrou :
  - **lock**() prend le verrou s'il est disponible et sinon endort la thread
  - **unlock**() relâche le verrou
  - **lockInterruptibly**() peut lever InterruptedException si le statut d'interruption est positionné lors de l'appel ou si la thread est interrompue pendant qu'elle le détient
  - **newCondition**() crée un objet **Condition** associé au verrou (cf plus tard: wait/notify)
- L'appel à **unlock**() doit se faire quoi qu'il arrive, par ex, dans un bloc finally.

tryLock()

- Il est possible d'essayer d'acquérir un verrou et si le verrou est déjà pris par une autre thread :
  - De ne pas être mis en attente
    - boolean tryLock()
  - Et d'être mis en attente mais seulement pendant un temps fixé
    - boolean tryLock(long time, TimeUnit unit)
      throws InterruptedException
- Attention, dans certain cas tryLock(long,TimeUnit) peut couter assez chère

ReentrantLock

- Implantation des verrous réentrant
  - Propose deux implantation différentes de la liste des threads en attente: équitable (faireness)/non équitable à choisir lors de la création
    ReentrantLock(boolean fairness)
  - Si l'équité choisi
    - La thread qui attend depuis le plus longtemps est favorisée.
    - Ce n'est pas l'équité de l'ordonnancement.
    - **tryLock**() ne repecte pas l'équité (si verrou dispo, retourne true).
    - Plus lent que l'algo non équitable

ReentrantLock (suite)

- Indique si le lock est en mode équitable
  - isFair()
- Indique si une thread a acquis le verrou
  - isLocked()
- Indique si une thread attend le relachement du verrou
  - boolean hasQueuedThread(Thread thread)
- Indique si la thread courante possède le verrou
  - **isHeldByCurrentThread**()
- Indique le nombre de fois que le verrou a été acqui
  - **getHoldCount**()

Exemple

- En reprenant l'exemple de la pile permettant les accès concurrent

```
public class LockedStack<E> {
  ...
  public boolean add(E element) {
    lock.lock();
    try {
      if (isFull())
        return false;
      array[top++]=element;
    } finally {
      lock.unlock();
    }
    return true;
  }
  private int top;
  private final E[] array;
  private final ReentrantLock lock=new ReentrantLock();
}
```

```
private boolean isFull() {
  assert lock.isHeldByCurrentThread();
  return top==array.length;
}
```

ReentrantReadWriteLock

- Permet de créer deux verrous distincts :
  - Verrou en lecture (readLock)/Verrou en écriture (writeLock)
- L'implantation garantie que
  - si une thread possède le verrou d'écriture aucune autre thread pourra acquérir le verrou de lecture
  - Plusieurs threads peuvent acquérir le verrou de lecture simultanément
  - L'obtention du verrou de lecture est garantie si l'on possède déjà le verrou d'écriture

Exemple

```
public class ReadWriteObjectCounter<E> {
  public ReadWriteObjectCounter() {
    ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    rLock = rwl.readLock();
    wLock = rwl.writeLock();
  }
  public void set(E element,int count) {
    wLock.lock();
    try {
      this.element=element;
      this.count=count;
    } finally {
      wLock.unlock();
    }
  }

  public int getIf(E element) {
    rLock.lock();
    try {
      return this.element.equals(element)?count:0;
    } finally {
      rLock.unlock();
    }
  }
```

```
  private E element;
  private int count;
  private final Lock rLock;
  private final Lock wLock;

  public static void main(String[] args) {
    ReadWriteObjectCounter<String> rwoc=
      new ReadWriteObjectCounter<String>();
    rwoc.set("toto",2);
    System.out.println(rwoc.getIf("toto")); //2
```

Exemple

- Il n'est pas possible de prendre un write-lock alors que l'on possède un read-lock, le contraire est par contre possible.

```
public E incrementIfNonNull() {
  rLock.lock();
  try {
    if (this.element==null)
      return null;
    rLock.unlock(); // attention
    wLock.lock();
    try {
      if (this.element==null)
        return null;
      this.count++;
      rLock.lock();
    } finally {
      wLock.unlock();
    }

    return this.element;
  } finally {
    rLock.unlock();
  }
}
```

Différence entre Lock et synchronized

- La gestion de la liste des threads en attente sur un verrou ou sur un moniteur n'est pas la même
- Pas de mode “fair” pour les moniteurs, en cas de forte contention, les verrous Lock sont plus efficaces
- Les moniteurs sont :
  - mieux intégrés à la VM (bias locking, lock coarsing)
  - occupe moins de mémoire que les Lock (lighweight lock/fat lock)
  - spin lock pas implanté pour les Lock en 1.5

Mémory modèle et synchro

- Barrière mémoire:
  - L'acquisition (***acquire***) d'un moniteur ou un Lock.lock() invalide le cache et force la relecture depuis la mémoire principale
  - Le relâchement (***release***) d'un moniteur ou un Lock.unLock() force l'écriture dans la mémoire principale de tout ce qui a été modifié dans le cache avant ou pendant le bloc synchronisé
- Il n'est donc pas nécessaire d'utiliser des variables volatiles si l'on utilise un moniteur ou un lock.

Interbloquage

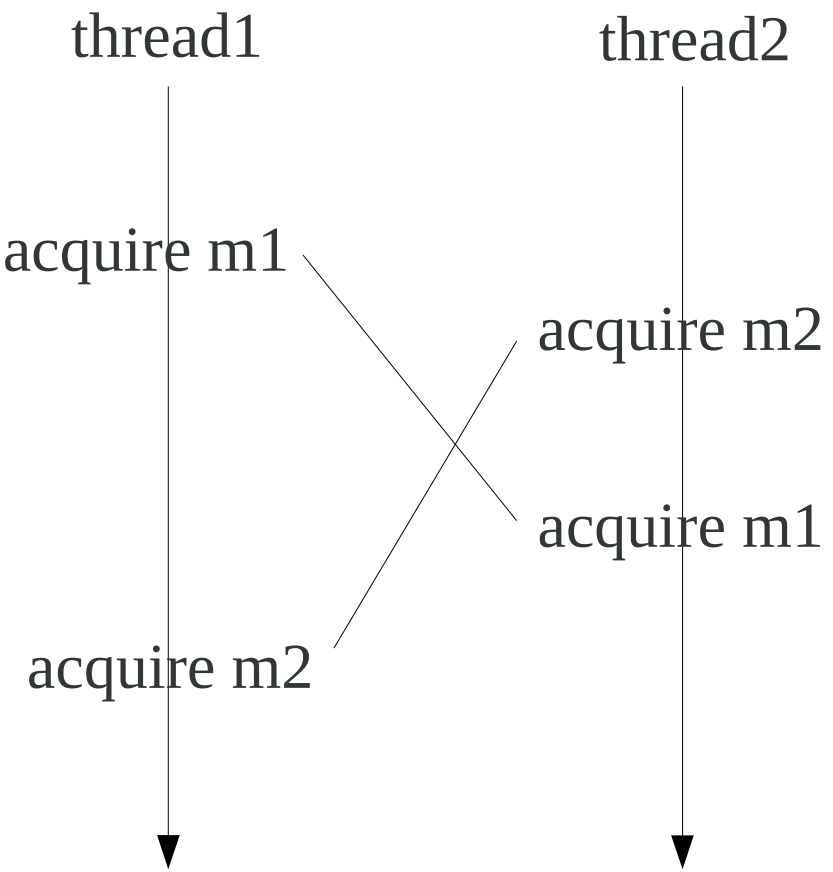
- Deux threads attendent chacune que l'autre libère le verrou. Un deadlock n'arrive pas forcément à tout les coups

```
public class Deadlock {
  Object m1 = new Object();
  Object m2 = new Object();
  public void ping() {
    synchronized (m1) {
      synchronized (m2) {
        // Code synchronisé sur
        // les deux moniteurs
      }
    }
  }
```

```
  public void pong() {
    synchronized (m2) {
      synchronized (m1) {
        // Code synchronisé sur
        // les deux moniteurs
      }
    }
  }
}
```

Représentation de l'interbloquage

- S'il y a un croisement, il y a un deadlock !!



Deadlock et initalisation de classe

- Le block statique est protégée par un verrou !!

```
public public class DeadlockLazy {
 static boolean initialized = false;
 static {
   Thread t = new Thread(new Runnable() {
     public void run() {
       initialized = true;
     }
   });
   t. start();
   try {
     t.join();
   } catch (InterruptedException e) {
     throw new AssertionError(e);
   }
 }
 public static void main(String[] args) {
   System.out.println(initialized);
 }
}
```

Efficacité & Interbloquages

- Eviter d'utiliser un verrou global
(comme dans l'exemple RecList)
- « diviser pour régner »
  - Découper en plusieurs verrous si pas de dépendance
- Faire un choix à la conception de l'ordre dans lequel on prend les verrous
  - toujours prendre les verrous dans le même ordre
- Réduire la taille des sections critiques au minimum

Singleton pas threadsafe

- Design Pattern *Singleton*
  - Eviter l'allocation d'une ressource coûteuse en la réutilisant

Deux risques :
1) Scheduling entre la création et l'assignation

2) publication de champs non initialisés

```
public class DB {
  private DB() { }
  public static DB getDB() {
   if (singleton==null)
     singleton=new DB();
   return singleton;
  }
  private static DB singleton;
}
```

ThreadSafe Singleton

- Mettre un synchronized globale, ok mais couteux

```
public class DB {
  private DB() { }
  public static DB getDB() {
   synchronized(DB.class) {
     if (singleton==null)
       singleton=new DB();
     return singleton;
   }

  }
  private static DB singleton;
}
```

- Et si on faisait le test à null avant le synchronized ?

Une bonne idée... qui finit mal

- Design Pattern *Double-Checked Locking*
  - Pour résoudre le problème, on peut essayer de "protéger" l'allocation, tout en limitant la nécessité de synchroniser:
  - force la mise à jour
  - assure que new n'est fait qu'1 fois

```
public class DB {
  public static DB getDB() {
   if (singleton==null) {
     synchronized(DB.class) {
       if (singleton==null)
         singleton=new DB();
     }
   }
   return singleton;
 }
 private static Resource singleton;
```

Reste le problème de la publication de champs non initialisés

Le DCL ne marche pas

```
public class DB {
  public static DB getDB() {
   if (singleton==null) {
     synchronized(DB.class) {
       if (singleton==null)
         singleton=new DB();
     }
   }
   return singleton;
 }
 private static DB singleton;
```

- Le problème :

- thread A entre dans getDb()
- thread B aussi et prend le moniteur et fait
  a) allocation par new
  b) appelle le constructeur **DB()** (initialisation des champs)
  c) affectation de la référence dans le champ **singleton**
- le problème vient du fait que thread A (qui n'est pas synchronisée) peut perçevoir les opérations a) b) c) dans le désordre.
  En particulier, A peut voir c) (disposer de la référence sur la singleton != null) avant de voir les effets de b) (et donc utiliser le singleton non initialisée)

La solution

- Utiliser le faite que les classes sont chargées de façon paresseuse (lazy)

```
public class DB {
  public static DB getDB() {
   return Cache.singleton;
  }
  static class Cache {
    static final DB singleton=new DB();
  }
}
```

- Le chargement d'une classe utilise un verrou et la VM garantie que la classe Cache ne sera chargée qu'une seule fois si nécessaire.

A1: Incrémentation

- **Problème** car l'incrémentation et l'assignation sont deux opérations différentes

Scheduling entre la lecture et de l'écriture

```
public class IDGenerator {
  private volatile long counter=0;
  public long getId() {
    return counter++;  // <==> counter=counter+1;
  }
}
```

- On peut utiliser synchronized/lock mais il y a mieux

```
public class IDGenerator {
  public long counter=0;
  public long getId() {
    synchronized(this) {
      return counter++;
    }
  }
```

A2: Test et assignation

- **Problème** car le test et l'assignation sont deux opérations différentes

```
public class NodeList<E> {
  private final E element;
  private volatile NodeList<E> next;
  public NodeList<E> addLast(E element) {
    NodeList<E> newNode=new NodeList<E>(element);
    for(NodeList<E> l=this;l.next!=null;l=l.next);
    l.next=newNode;
    return newNode;
  }
}
```

- Pourtant les processeurs modernes possèdent des instructions effectuant ces opérations atomiquement

Scheduling entre la lecture et de l'écriture

java.util.concurrent.atomics

- Opérations atomiques sur différents types de variable, étend la notion de *volatile*
  - Pour chacune des différentes classes AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference...
    - Autorise des implémentations plus efficaces en fonction des possibilités offertes par les processeurs
    - Opérations atomiques inconditionnelles:
      - get(): effet mémoire de la lecture d'un champ volatile
      - set(): effet mémoire de l'écriture d'un champ volatile
      - getAndSet(): effet mémoire de lecture et d'écriture d'un champ volatile
        - Comme getAndIncrement(), getAndAdd(), etc pour les entiers

Modification de valeur

- Si 2 threads exécutent `ai.getAndAdd(5);` sur un objet `AtomicInteger ai = new AtomicInteger(10);`
  - Alors on est assuré que les opérations ne sont pas entrelacées, qu'au final ai.get() vaut 20.
  - Pas de verrou, ni de synchronisation (moins **lourd**)

```
import java.util.concurrent.atomic.AtomicInteger;

public class ThreadSafeCounter {
  public int getNextUniqId() {
    return ai.getAndIncrement();
  }

  private final AtomicInteger ai=new AtomicInteger();
}
```

Opérations atomiques conditionnelles

- **`boolean compareAndSet(expectedValue, updateValue)`**
  - Affecte atomiquement la valeur updateValue dans l'objet atomique si la valeur actuelle de cet objet est égale à la valeur expectedValue.
    - Retourne true si l'affectation a eu lieu, false si la valeur de l'objet atomique est différente de expectedValue au moment de l'appel.
  - Effet mémoire: lecture/écriture d'un champ volatile
- **`boolean weakCompareAndSet(expectedValue, updateValue)`**
  - Même chose, mais peut échouer (retourne false) pour une raison «inconnue» (*spurious*) : on a le droit de ré-essayer.
  - Effet mémoire: ordonnancement correct des opérations sur cette variable, mais pas de "barrière"

Opération atomique paresseuse

- `void lazySet(value)`
  - Effet mémoire d'une écriture volatile, sauf qu'il autorise des réordonnancements avec les actions mémoires ultérieures n'ayant pas de contraintes avec les écritures non volatiles (ce qui n'est pas permit avec une écriture volatile)
  - Utilisation: mettre une valeur à null pour qu'elle soit garbage collectée

Compare and Set

- Un appel à **compareAndSet** est remplacé par la machine virtuelle par l'opération assembleur correpondante
- Avec compareAndSet, il est possible de ré-écrire getAndAdd(), getAndIncrement() etc
- L'implantation des Locks utilise aussi compareAndSet

```
public class ThreadSafeCounter2 {
  public int getNextUniqId() {
    for (;;) {
      int current=ai.get();
      int next=current+1;
      if (ai.compareAndSet(current,next))
        return current;
    }
  }
  private final AtomicInteger ai=new AtomicInteger();
```

CompareAndSet sur volatile

- Atomic[*Reference* | *Integer* | *Long*]FieldUpdater<T,V> permettent d'utiliser des opérations "*compareAndSet*" sur les champs **volatiles** d'une classe donnée.
  - static factory :
    static <T,V> AtomicReferenceFieldUpdater<T,V>
    newUpdater(Class<T> tclass, Class<V> vclass,
    String volatileFieldName)

  Utilisation :

  - updater.compareAndSet(T obj,V expect,V update)
  - updater.lazySet(T o,V value)

Exemple avec une liste chaînée

```
public class SafeList {
  private volatile Entry head;
  static class Entry {
    private final double value;
    private final Entry next;
    public Entry(double value,Entry next){
      this.value=value;
      this.next=next;
    }
    public double getValue() {
      return value;
    }
  }
  public void addFirst(double value) {
    for (;;) {
      Entry currentEntry=this.head;
      Entry newEntry=new Entry(value,currentEntry);
      if (updater.compareAndSet(this,currentEntry,newEntry))
        return;
    }
  }
  private final static AtomicReferenceFieldUpdater<SafeList,Entry> updater=
    AtomicReferenceFieldUpdater.newUpdater(
      SafeList.class,Entry.class,"head");
```

Autres atomics spéciaux

- Atomic[*Integer* | *Long* | *Reference*]Array permet de rendre atomique l'accès à une case d'un tableau
- AtomicMarkableReference<V> associe un booléen à une référence et AtomicStampedReference<V> associe un entier à une référence
  - boolean compareAndSet(V expectRef,
    V newRef,
    int expectStamp,
    int newStamp)

Atomicité *vs* exclusion mutuelle

- La notion de **publication** tend à **garantir** que l'**état** de la **mémoire** sera **cohérent**
  - volatile, final
  - java.util.concurrent.atomics.*
- La notion d'**exclusion mutuelle** (synchronized, lock) ne fait qu'assurer que du code n'aura **pas accès** à un **état** de la **mémoire incohérent**
  - La mémoire peut effectivement être dans un état incohérent
  - Rien n'empêche un code tierce d'avoir accès à ces incohérences s'il n'est pas protégé

W1:Data Race

- Il ya une "course aux données" (***data race***), lorsque :
  - Une thread **écrit** dans une variable
  - Une autre thread **lit** dans cette variable
  - L'**écriture** et la **lecture** ne sont **pas ordonnées**

```
public class DataRace {
  static int a=0;
  public static void main(String[] args) {
    new Thread(new Runnable() {
      public void run() {
        System.out.println(a); // peut afficher 1 comme 0
      }
    }).start();
    a=1;
  }
}
```

- **Thread.sleep** n'est pas une bonne façon de répondre au *data race* !!

Rendez-vous (Synchronisation)

- Nous connaissons déjà deux forme de RDV :
  - Attendre qu'une thread soit terminée=
    - thread.join()

      accepte éventuellement une durée (en milli ou nano sec)
  - Attente active
    - while(!interrupted()) et thread.interrupt()
- Existe une forme d'attente passive :
  - Attendre l'arrivée d'un événement particulier object.wait();
  - Notifier l'arrivée de cet événement: object.notify();

Object.wait()

- o.wait(); exécuté par une thread
  - requiert de détenir le moniteur sur o
  - suspend la thread courante et libère le moniteur o
  - la thread suspendue attend (passivement) d'être réveillée par une notification sur ce même moniteur
  - Lorsque qu'une autre thread envoie une notification associée à o, la thread courant doit à nouveau acquérir le moniteur associé à o afin de poursuivre son exécution et sortir du wait()

wait(timeout)

- Attente bornée par un délai
  - La thread est supendu en attente pendant le délai
  - passé ce délai, la thread peut reprendre son exécution mais doit auparavant reprendre le moniteur o.
- Deux formes différentes
  - Attente avec un délai en milliseconde
    void Object.**wait**(long timeout)
  - Attente avec un delai en nano seconde
    void Object.**wait**(long timeout,int nanos)

Object.wait() et reveil

- Il y a trois façons d'être reveillé dans un wait :
  - Par un **notify**()/**notifyAll**()
  - De façon “spurious”, sans le vouloir, du à l'implantation de wait/notify suivant les OS/processeur/carte mère
  - Par un interrupt(), dans ce cas une exception InterruptedException est levée par wait();
    - le statut d'interruption est réinitialisé par la levée de cette d'exception, i faut faire quelque chose
      - Arrêter le traitement
      - Repositionner le flag pour plus tard
        - Thread.currentThread().interrupt()

Notification

- Une notification peut être émise par une autre thread
  - par o.notify(): une seule thread en attente de notification sur o sera réveillée (la thread choisie dépend de l'implantation);
  - ou par o.notifyAll(): toutes les threads en attente de notification sur o seront réveillées (elles entrent alors en concurrence pour obtenir ce moniteur).
- L'exécution de o.notify() ou o.notifyAll()
  - requiert que la thread détienne le moniteur associé à o
  - ne libère pas le moniteur (il faut attendre d'être sorti du bloc de code protégé par un « synchronized (o) {...} »)

Notification perdue

- Absence de tout ordonnancement connu
  - possibilité de « perdre » une notification
    - par ex. si elle a été émise par t2 avant que t1 soit mise en attente

  dans ce cas, lorsque t1 rentre dans le wait(), il attend indéfiniment

- Solution:
  - Il faut utiliser une variable partagée que l'on testera avant le wait()

Spurious wake up

- wait() peut être reveillé sans notify(), on doit protéger wait() :
  - Par un while pour le rendormir
    - Oui mais un while sur quoi ??
      - On doit avoir une variable à tester
- **Permet aussi d'attraper les notifications perdues**

```
public void ...() {
  synchronized(lock) {
    while(!start) {
      lock.wait();
    }
  }
}
```

```
public void ...() {
  synchronized(lock) {
    start=true;
    lock.notify();
  }
}
```

Utiliser notifyAll() plutôt que notify()

- Le fait d'utiliser notifyAll() permet de reveiller toutes les threads en attente, dans ce cas, la thread la plus prioritaire pour le scheduler sera reveillé en premier

```
public void ...() {
  synchronized(lock) {
    while(!start) {
      lock.wait();
    }
    start=false;
  }
}
```

```
public void ...() {
  synchronized(lock) {
    start=true;
    lock.notifyAll();
  }
}
```

- Comme la sortie du wait() demande la ré-acquisition du moniteur, une seule thread sortira du bloc synchronized, les autres seront rendormis

Utiliser notify() plutôt que notifyAll()

- Comme notifyAll() réveille tout le monde, cela coûte cher, donc on l'utilise que lorsque l'on en a vraiment besoin

```
public void ...() {
  synchronized(lock) {
    while(!start) {
      lock.wait();
    }
    start=false;
  }
```

```
public void ...() {
  synchronized(lock) {
    start=true;
    lock.notify();
  }
}
```

Pile bloquante

- On souhaite avoir une pile qui met en attente les thread qui effectue un **pop**() sur une pile vide

```
public class BlockingStack<E> {
  public void push(E element) {
    synchronized(stack) {
      stack.push(element);
      notify();
    }
  }
  public E pop() throws InterruptedException {
    synchronized(stack) {
      while(stack.isEmpty()) {
        wait();
      }
      return stack.pop();
    }
  }
  private final ArrayDeque<E> stack=
    new ArrayDeque<E>();
}
```

Ce code ne marche pas !!!

Pile bloquante

- On doit faire un wait()/notify() sur le même objet que celui utiliser pour le moniteur si lève l'exception IllegalMonitorStateException

```
public class BlockingStack<E> {
  public void push(E element) {
    synchronized(stack) {
      stack.push(element);
      stack.notify();
    }
  }
  public E pop() throws InterruptedException {
    synchronized(stack) {
      while(stack.isEmpty()) {
        stack.wait();
      }
      return stack.pop();
    }
  }
  private final ArrayDeque<E> stack=new ArrayDeque<E>();
```

Interface Condition

- java.util.concurrent.locks.Condition
  - Supporte pour n'importe quel objet verrou de type Lock, l'équivalent des méthodes (wait(), notify() et notifyAll()) pour une condition particulière, i.e., plusieurs Conditions peuvent être associées à un Lock donné
  - Offre les moyens de
    - mettre une thread en attente de la condition: méthodes cond.await(), équivalentes à monitor.wait()
    - et de la réveiller: méthodes cond.signal()/signalAll(), équivalentes à monitor.notify()/notifyAll()
    - La thread qui exécute ces méthodes doit bien sûr détenir le verrou associé à la condition

Attente/Notification

- Relâche le verrou (Lock) associé à cette condition et inactive la thread jusqu'à ce que :
  - Soit une autre thread invoque la méthode signal()/signalAll sur la condition.
  - Soit qu'une autre thread interrompe cette thread (et que l'interruption de suspension soit supportée par l'implantation de await() de cette Condition)
  - Soit un réveil «illégitime» intervienne (*spurious wakeup)*
- Dans tous les cas, il est garanti que lorsque la méthode retourne, le verrou associé à la condition a été repris
- Si le statut d'interruption est positionné lors de l'appel à cette méthode ou si la thread est interrompue pendant son inactivité, la méthode lève une InterruptedException

Attente-notification (suite)

- void awaitUninterruptibly()
  - Ne tient pas compte du statut éventuel d'interruption
- long awaitNanos(long nanosTimeout) throws InterruptedException
  - Retourne (une approximation) du nombre de nanosecondes restant avant la fin du timeout au moment où la méthode retourne, ou une valeur négative si le timeout a expiré
- boolean await(long time, TimeUnit unit) throws InterruptedException
- boolean awaitUntil(Date deadline) throws InterruptedException
- void signal(), void signalAll()
  - Réactive une ou toutes les threads en attente de cette condition

Pile bloquante avec une condition

```
public class BlockingStack<E> {
  public void push(E element) throws InterruptedException {
    lock.lock();
    try {
      stack.push(element);
      condition.signal(); // car signalAll() trop couteux
    } finally {
      lock.unlock();
    }
  }
  public E pop() throws InterruptedException {
    lock.lock();
    try {
      while(stack.isEmpty()) {
        condition.await();
      }
      return stack.pop();
    } finally {
      lock.unlock();
    }
  }
  private final ReentrantLock lock=new ReentrantLock();
  private final Condition condition=lock.newCondition();
  private final ArrayDeque<E> stack=new ArrayDeque<E>();
}
```

LockInterruptibly ?

```
public class BlockingStack<E> {
  public void push(E element) throws InterruptedException {
    lock.lockInterruptibly();
    try {
      stack.push(element);
      condition.signal();
    } finally {
      lock.unlock();
    }
  }
  ...
  private final ReentrantLock lock=new ReentrantLock();
  private final Condition condition=lock.newCondition();
  private final ArrayDeque<E> stack=new ArrayDeque<E>();
}
```

Shutdown Hooks

- Plusieurs façons de sortir d'une exécution de JVM
  - Normales:
    - Fin de tous les processus légers (Thread) non *daemon*
    - Appel de la méthode exit() de l'environnement d'exécution
  - Plus brutales:
    - Control-C
    - Levée d'exception jamais récupérée
- Shutdown hooks
  - Code à exécuter par la JVM quand elle s'arrête
  - Sous la forme de processus légers

Enregistrement des crochets d'arrêt

- Sur l'objet runtime courant Runtime.getRuntime()
  - addShutdownHook(...) pour enregistrer
  - removeSutdownHook(...) pour désenregistrer
  - Avec comme argument un contrôleur de processus léger, héritant de la classe Thread, dont la méthode run() de la cible spécifie du code
  - Lorsqu'elle s'arrête, la JVM appelle leur méthode start()
    - Aucune garantie sur l'ordre dans lequel ils sont exécutés
    - Chacun est exécuté dans un processus léger différent
    - L'ensemble des threads à un temps maximum pour s'exécuter

Exemple de crochet d'arrêt

```
public static void main(String[] args)  throws IOException {
  final BufferedWriter out = new BufferedWriter(
      new FileWriter("typed.txt"));
  Thread hook=new Thread(new Runnable() {
    public void run() {
      try {
        try {
          out.write("Exec. aborted at "+new Date());
          out.newLine();
        } finally {
          out.close();
        }
      } catch(IOException ioe) {
        ioe.printStackTrace();
      }
    }
  });
  Runtime.getRuntime().addShutdownHook(hook);
  Scanner scanner=new Scanner(System.in);
  while (scanner.hasNextLine()) {
    String line=scanner.nextLine();
    if ("halt".equals(line))
      Runtime.getRuntime().halt(1);
    out.write(line);
      out.newLine();
  }
  scanner.close();
  out.close();
  Runtime.getRuntime().removeShutdownHook(hook);
}
```

Résultat

- Si le flot d'entrée est fermé normalement (Ctrl-D)
  - Le crochet est désenregistré (jamais démarré par JVM)
  - Les flots sont fermés (le contenu purgé dans le fichier)
- Si la machine est arrêtée par un Ctrl-C ou par une exception non récupérée
  - Le crochet est démarré par la JVM
  - Le fichier contient : « **`Exec. aborted at...`** »
- Si la JVM est arrêtée par halt() ou par un **kill**
  - Le crochet n'est pas démarré
  - Potentiellement, le flot n'a pas été purgé dans le fichier