Aperçu du modèle mémoire JAVA

Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

Année 2017-2018

Version du 21 février 2018

Un problème de vue

```
public static void main(String[] args) throws Exception {
  A a = new A() ;              // Création d'un objet a de la classe A
  a.start() ;                  // Lancement du thread a
  a.valeur = 1 ;               // Modification de l'attribut valeur
  a.fin = true ;               // Modification de l'attribut fin
}
static class A extends Thread {
  public int valeur = 0 ;
  public boolean fin = false ;

  public void run() {
    while(! fin) {} ;          // Attente active
    System.out.println(valeur);
  }
}
```

Ce programme termine-t-il ? Peut-il afficher 0 ?

À quoi sert le modèle mémoire Java ?

Le modèle mémoire Java définit la sémantique des programmes Java, c'est-à-dire l'ensemble des résultats possibles d'un programme donné.

Il s'agit en fait de découvrir pourquoi un programme Java peut sembler

① *cacher* momentanément les écritures réalisées en mémoire,

② ou bien ne pas exécuter les instructions dans *l'ordre du programme*.

En particulier, le modèle mémoire Java précise l'effet du mot-clef **`volatile`**.

Ajout du mot-clef « volatile »

```
public static void main(String[] args) throws Exception {
  A a = new A() ;              // Création d'un objet a de la classe A
  a.start() ;                  // Lancement du thread a
  a.valeur = 1 ;               // Modification de l'attribut valeur
  a.fin = true ;               // Modification de l'attribut fin
}
static class A extends Thread {
  public int valeur = 0 ;
  public volatile boolean fin = false ;

  public void run() {
    while(! fin) {} ;          // Attente active
    System.out.println(valeur);
  }
}
```

Ce programme termine-t-il ? Peut-il afficher 0 ?

Exécutions légales

Le **modèle mémoire java** (JMM, pour « Java Memory Model ») définit la sémantique des programmes Java ; c'est en fait un *modèle d'exécution* : il détermine quelles sont les **exécutions légales** (et donc les résultats légaux) d'un programme donné.

C'est à la fois

① une *garantie* pour les développeurs :

Un programme Java ne peut pas faire n'importe quoi !

② une *contrainte* pour les fabricants de machines virtuelles Java :

Les JVM ne doivent pas faire n'importe quoi !

La relation de précédence « a lieu avant »

Il est fondamental, lors de l'exécution d'une instruction, de pouvoir déterminer quelles autres instructions (sur le même thread, ou sur les autres threads) *doivent avoir eu lieu* précédemment, celles qui *peuvent avoir eu lieu*, et celles qui *ne le peuvent pas*.

Le JMM définit une *relation* « happens-before » entre les instructions d'une exécution, relation qui est un **ordre partiel**, noté $<_{\mathrm{HB}}$. Cette relation est donc **transitive** :

$\rightsquigarrow$ si x a lieu avant y et si y a lieu avant z, alors x a lieu avant z.

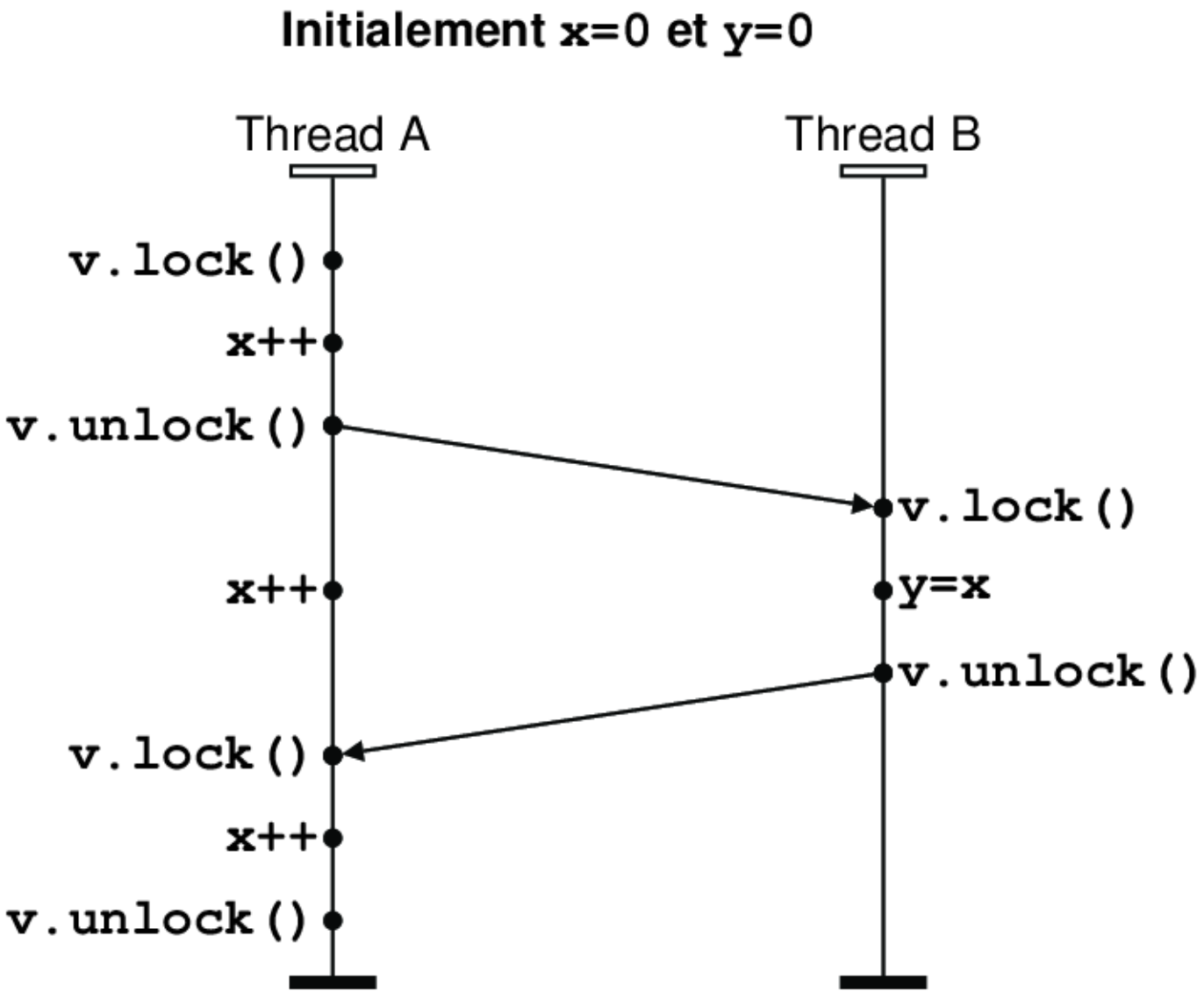
Cette relation est construite principalement par

① **l'ordre du programme** qui garantit une consistance locale à chaque thread ;

② certaines actions qui produisent des **synchronisations** entre threads ; par exemple, les opérations sur un verrou sont **totalement ordonnées** par la relation $<_{\mathrm{HB}}$.

Chaque instruction *voit* ce qui la précède selon $<_{\mathrm{HB}}$, mais potentiellement un peu plus !

Exemple d'exécution légale (avec deux synchronisations)



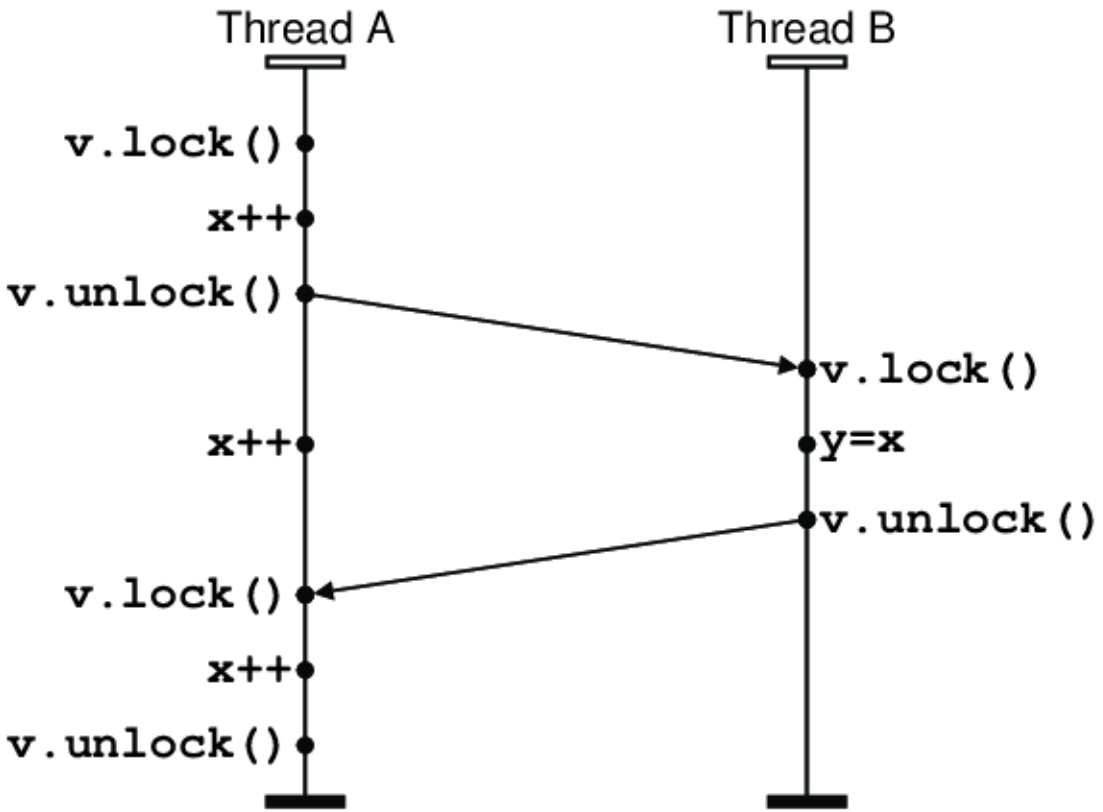
Que vaut `y` *à la fin de cette exécution ?*

✔ *Formalisme du modèle mémoire Java*

☛ *Visibilité assurée par l'emploi d'un verrou*

Il faut distinguer la relation « a lieu avant » et l'observation effective

Initialement `x=0` et `y=0`



- ⇝ **`y=x`** doit observer la première incrémentation de **`x`**, qui a lieu avant.
- ⇝ **`y=x`** ne peut pas observer la troisième incrémentation de **`x`**, qui a lieu après.
- ⇝ **`y=x`** observera ou non la seconde incrémentation de **`x`**.
- ⇝ à la fin de cette exécution, **`y=1`** ou **`y=2`** (mais jamais **`y=3`**).

Première caractéristique des exécutions légales

Une exécution légale doit, pour chaque verrou **`v`**, déterminer l'ordre des opérations sur le verrou, via une **synchronisation** de chaque appel à **`v.unlock()`** vers l'appel à **`v.lock()`** ultérieur.

Ces synchronisations déterminent l'ordre d'utilisation d'un verrou donné par les threads, conformément à leur spécification :

- au plus un seul thread possède le verrou à chaque instant ;
- les verrous sont ré-entrants ;
- seul le thread qui possède le verrou peut le relâcher : chaque appel à **`v.lock()`** ≪ a lieu avant ≫ l'appel à **`v.unlock()`** associé, *puisque l'ordre des instructions du programme est respecté.*

- Cela vaut pour les verrous intrinsèques comme pour la classe **`ReentrantLock`**.
- Cela vaut aussi pour les appels à **`v.wait()`** puisque cette méthode inclus un **`v.unlock()`** au moment de l'endormissement ainsi qu'un **`v.lock()`** au moment du réveil.

Visibilité et verrou : un exemple simple

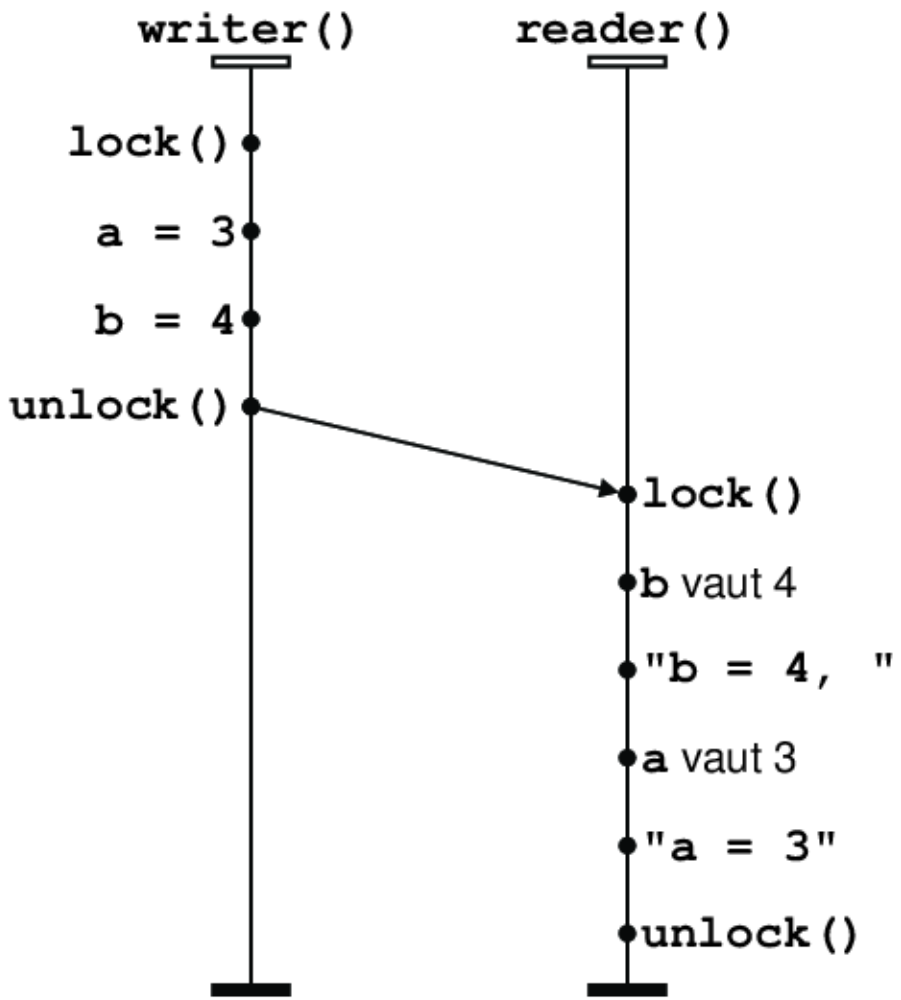
```
class SynchronizedNotSoSimple {
  private int a = 1, b = 2 ; // ne sont pas déclarées volatiles !
  void synchronized writer() {
    a = 3;
    b = 4;
  }
  void synchronized reader() {
    System.out.print("b␣=␣" + b + ",␣");
    System.out.println("a␣=␣" + a);
  }
}
```

Un thread va exécuter **`writer()`** et un autre **`reader()`**. Du fait de **`synchronized`**, les seules sorties légales sont :

- **`"b=2, a=1"`** si le lecteur prend le verrou en premier ;
- **`"b=4, a=3"`** si l'écrivain prend le verrou en premier.

Une exécution légale

Initialement a=1 et b=2



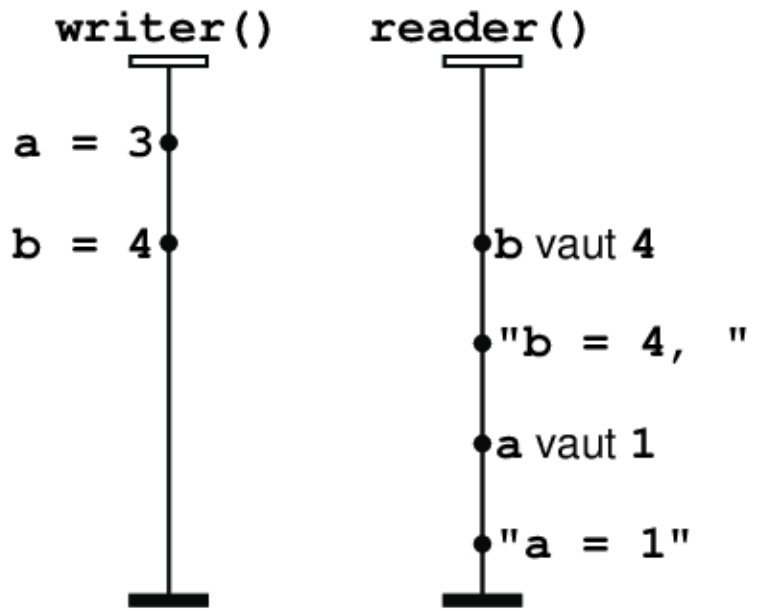
Le verrou garantit la visibilité !

En supprimant synchronized, le résultat peut être surprenant !

```
class NotSoSimple {
  int a = 1, b = 2;

  void writer() {
    a = 3;
    b = 4;
  }

  void reader() {
    System.out.print("b = " + b +", ");
    System.out.println("a = " + a);
  }
}
```



Il n'y a aucune synchronisation !

Ce programme pourra légalement afficher **"b = 4, a = 1"**...

- ✔ *Formalisme du modèle mémoire Java*
- ✔ *Visibilité assurée par l'emploi d'un verrou*
- ☛ *Le modificateur volatile*

Retour au premier exemple

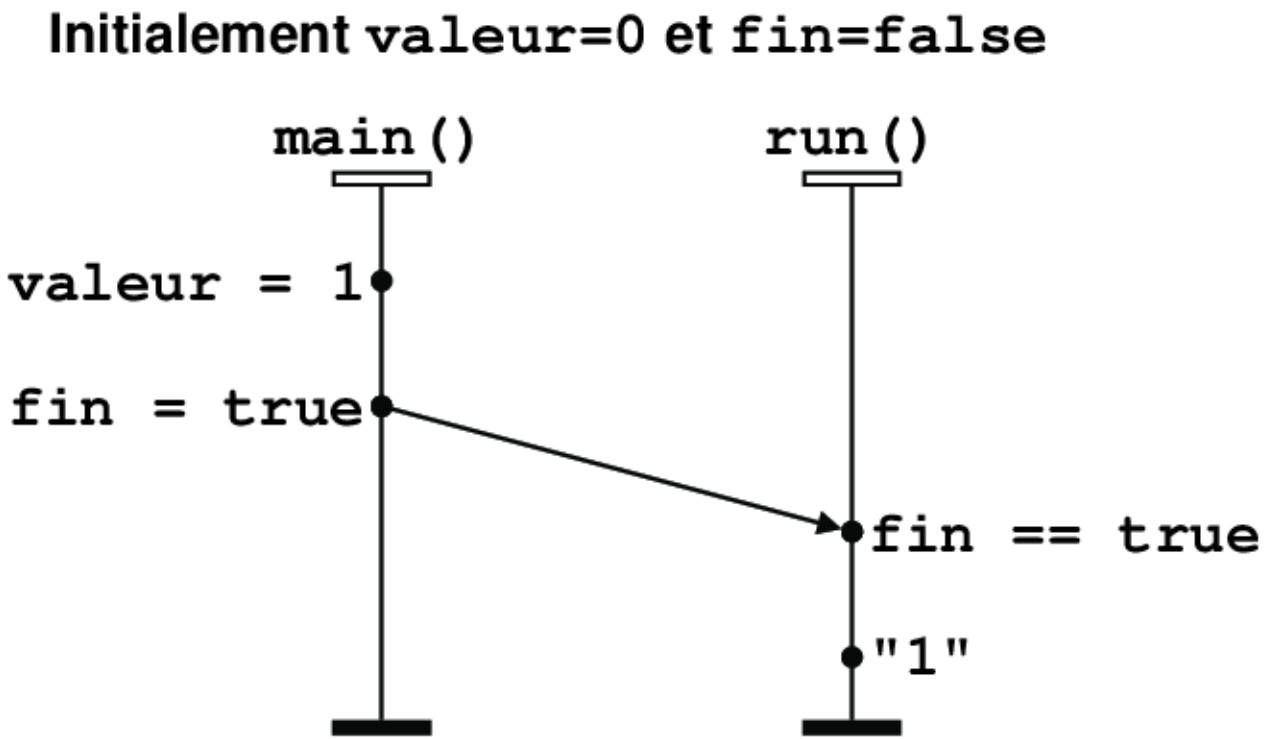
```
public static void main(String[] args) throws Exception {
  A a = new A();              // Création d'un objet a de la classe A
  a.start();                  // Lancement du thread a
  a.valeur = 1;               // Modification de l'attribut valeur
  a.fin = true;               // Modification de l'attribut fin
}
static class A extends Thread {
  public int valeur = 0 ;
  public volatile boolean fin = false ;

  public void run() {
    while(! fin) {} ;         // Attente active
    System.out.println(valeur) ;
  }
}
```

Ce programme termine-t-il ? Peut-il afficher 0 ?

`valeur` bénéficie de la volatilité de `fin`

Une exécution légale doit, pour chaque variable volatile `f`, comporter une **synchronisation** de chaque *écriture* dans `f` vers *les lectures et les écritures de `f` ≪ ultérieures. ≫*



Ce programme terminera et affichera `"1"`, car la modification du booléen volatile `fin` sera vue par le second thread, de même que celle de `valeur` *par transitivité* de la relation ≪ a lieu avant ≫ et du fait de l'ordre des instructions du programme.

Suppression du mot-clef volatile

Si tous les accès à une variable sont protégés par un *verrou*, alors il est inutile de déclarer cette variable **`volatile`**.

C'est le cas lorsque l'on adopte le modèle des moniteurs.

C'est utile à savoir car :

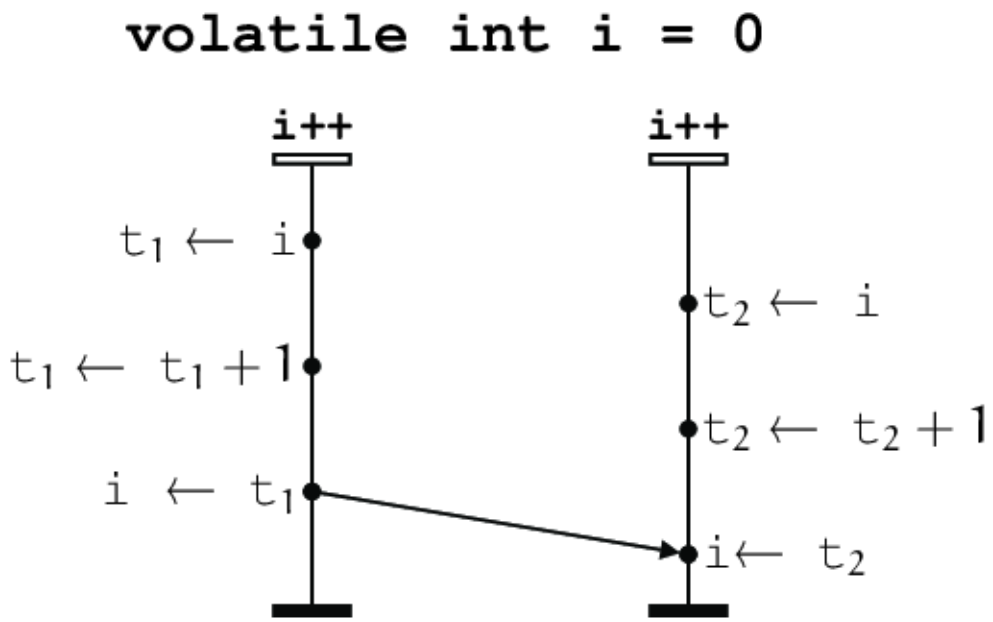
- ça évite de mettre **`volatile`** partout (ce qui peut nuire aux performances) ;
- il est parfois impossible d'assurer la visibilité à l'aide du mot-clef **`volatile`**.

En effet si un objet (ou un tableau) est déclaré **`volatile`**, les modifications de cette référence seront visibles, mais pas les modifications des attributs de cet objet (ou du contenu du tableau).

- ✔ *Formalisme du modèle mémoire Java*
- ✔ *Visibilité assurée par l'emploi d'un verrou*
- ✔ *Le modificateur volatile*
- ☛ *Visibilité des objets atomiques*

Ultime rappel

L'incrémentation d'une variable volatile n'est pas atomique.

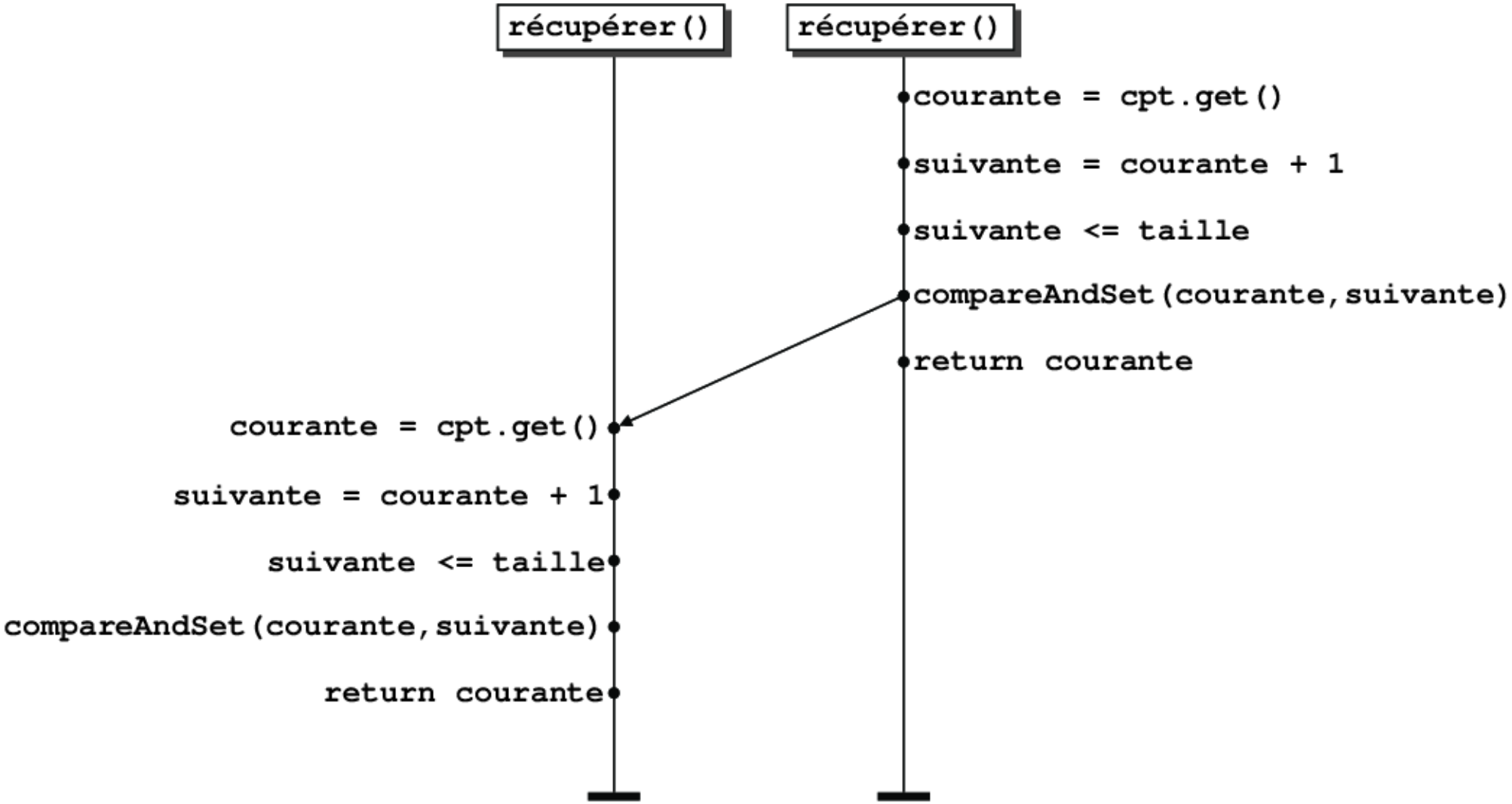


C'est l'une des raisons pour lesquelles quelques classes d'**objets atomiques** ont été introduites dans Java.

Exemple typique de codage ≪ optimiste ≫ (déjà vu)

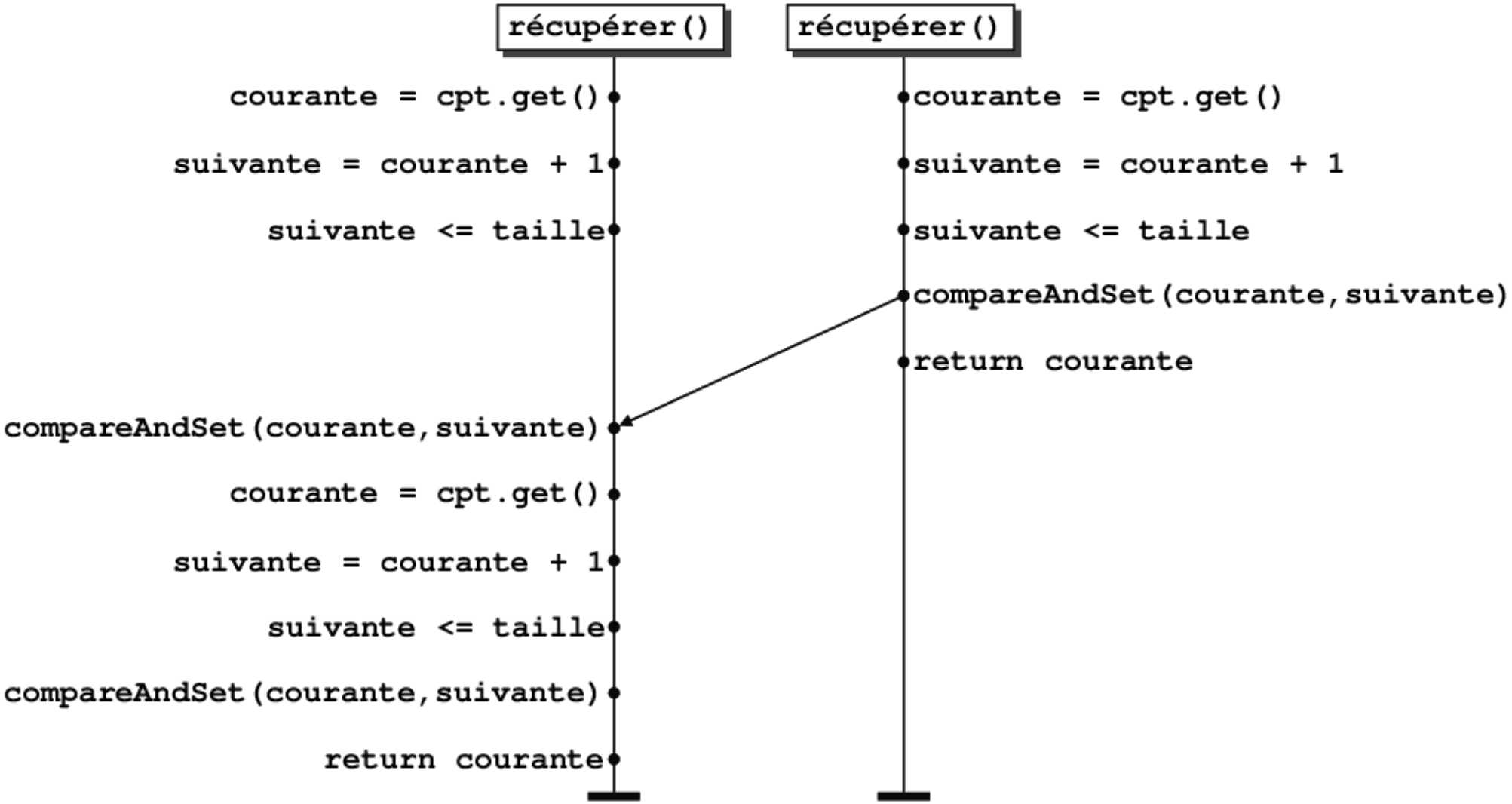
```
private static AtomicInteger cpt = new AtomicInteger(0);
public int récupérer() {
  do {
    int courante = cpt.get() ;
    int suivante = courante + 1 ;
    if ( suivante > taille ) suivante = taille ;
  } while ( ! cpt.compareAndSet(courante, suivante) ) ;
  return courante; // Cette valeur sera toujours =< taille
}
```

Exécution de la méthode `récupérer()` (1/2)



Les objets atomiques sont assurés d'être visibles, comme les volatiles !

Exécution de la méthode `récupérer()` (2/2)



Toute opération (en lecture, en écriture, etc.) sur un objet atomique est atomique !

Selon le site `docs.oracle.com`

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in section 17.4 of « The Java Language Specification. »

- **`get()`** has the memory effects of reading a volatile variable.
- **`set()`** has the memory effects of writing (assigning) a volatile variable.
- **`compareAndSet()`** and all other read-and-update operations such as **`getAndIncrement()`** have the memory effects of both reading and writing volatile variables.

Tableaux formés d'objets `atomics`

Méfiance !

Il n'y a aucun moyen de rendre les **éléments d'un tableau** « volatiles. »

En effet, si un tableau est déclaré **`volatile`**, c'est la référence du tableau qui bénéficiera de la volatilité : les modifications des éléments du tableau pourront ne pas être vues.

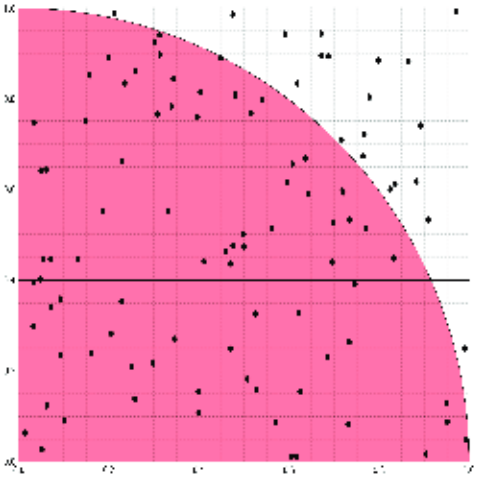
Les classes dédiées **`AtomicIntegerArray`**, **`AtomicLongArray`** ou encore **`AtomicReferenceArray`**< E > permettent de disposer d'un tableau

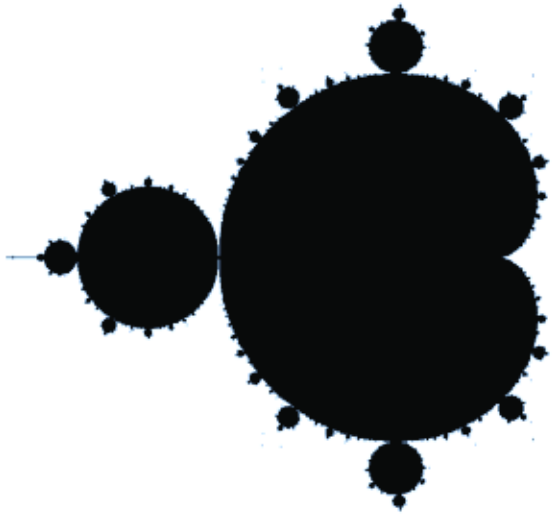
- dont les éléments peuvent être manipulés comme des objets atomiques ;
- rassemblés dans un seul objet dont la visibilité est garantie.

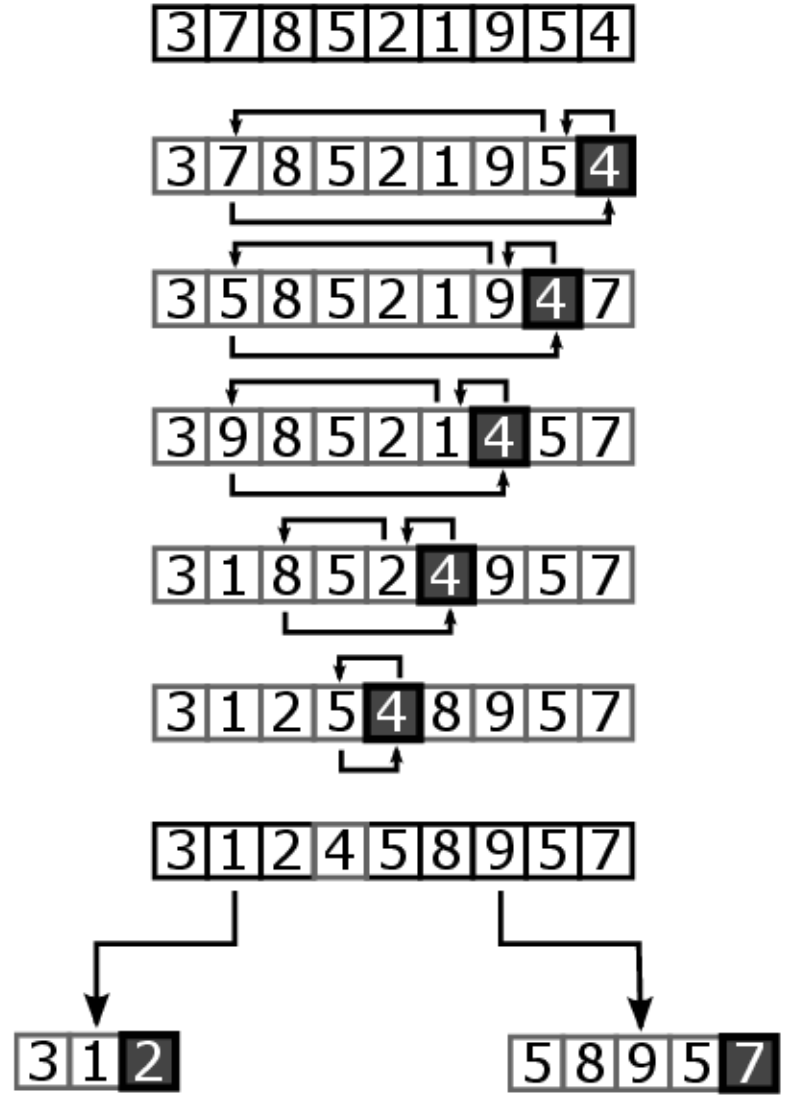
Pour un tableau d'objets quelconques, il faut utiliser un tableau de références atomiques vers les objets créés.

- ✔ *Formalisme du modèle mémoire Java*
- ✔ *Visibilité assurée par l'emploi d'un verrou*
- ✔ *Le modificateur volatile*
- ✔ *Visibilité des objets atomiques*
- ☛ *Quelques précisions utiles*

Exemples de négligences passées, mais sans conséquence







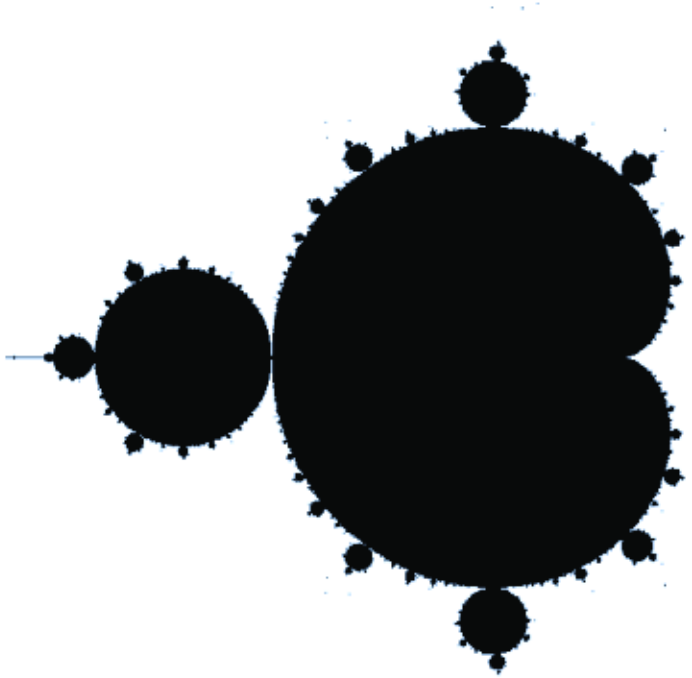
Les données partagées doivent être synchronisées pour être visibles.

Visibilité des résultats obtenus à l'issue d'un partage

Outre les synchronisations exigées par les opérations sur les verrous ou les accès aux variables volatiles, le JMM impose certaines autres synchronisations naturelles :

① La dernière action d'un thread **`t`** *a lieu avant* les actions déclarées après l'instruction **`t.join()`** sur un autre thread.

② Les actions représentées par un objet **`Future`** *ont lieu avant* l'obtention du résultat produit par la méthode **`get()`** sur cet objet.

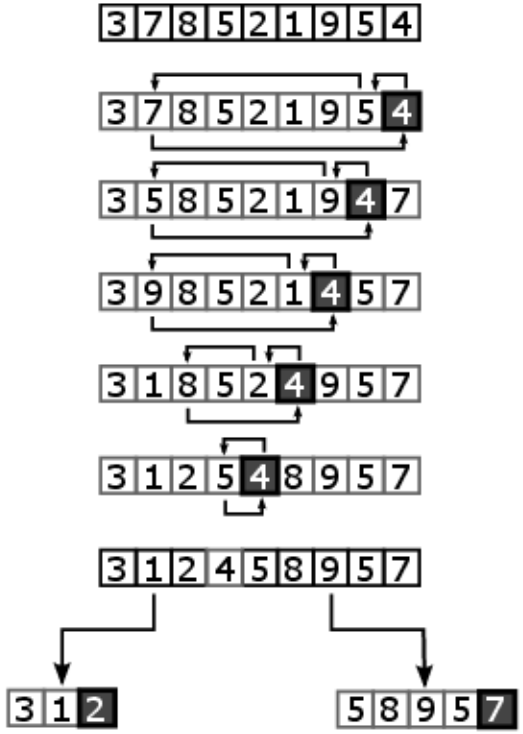


```
for (int i = 0 ; i < taille ; i++) {
  service.take().get() ;
} // Chaque ligne complétée sera nécessairement visible.
```

Visibilité des actions préalables à un partage

① Une action qui démarre un thread via la méthode **`start()`** *a lieu avant* toutes les actions exécutées par ce thread.

② La soumission d'un objet **`Runnable`** ou **`Callable`** à un exécuteur *a lieu avant* l'exécution de celui-ci.




```
public Boolean call() {
   ...
   int p = TriRapide.partitionner(tableau, début, fin) ;
   TriageRapide triAGauche = new TriageRapide( début, p-1 ) ;
   service.submit( triAGauche ) ;
   ...
} // Le thread exécutant triAGauche verra le partionnement fait.
```

Collections, barrières, loquets, sémaphores, etc.

Tout comme les objets atomiques sont garantis de se comporter comme des variables volatiles, les autres outils introduits dans Java 5 ont des propriétés naturelles vis-à-vis de la relation « a lieu avant » et donc au niveau de la visibilité :

- L'ajout d'un objet à une collection concurrente *a lieu avant* l'accès en lecture ou le retrait de cet objet dans cette collection.
- Les actions déclarées avant chaque appel à **`loquet.countDown()`** *ont lieu avant* celles déclarées après les retours de la méthode **`loquet.await()`** correspondants.
- Les actions déclarées avant l'appel à **`barrière.await()`** *ont lieu avant* celles associées éventuellement à cette barrière, et celles-ci ont elles-même lieu avant celles déclarées après le retour de la méthode **`barrière.await()`**.
- etc.

Consistance séquentielle et data-race

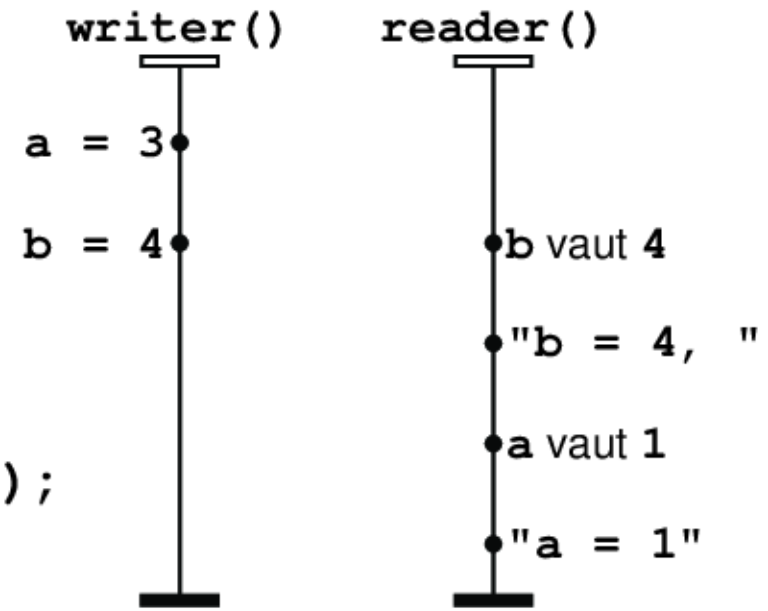
Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

Année 2017-2018

Version du 21 février 2018

Exemple d'exécution qui n'est pas séquentiellement consistante

```
class NotSoSimple {
  int a = 1, b = 2;
  void writer() {
    a = 3;
    b = 4;
  }
  void reader() {
    System.out.print("b = " + b +", ");
    System.out.println("a = " + a);
  }
}
```



Ce programme peut légalement afficher **`"b = 4, a = 1"`**.

① La mise-à-jour de la valeur de **`a`** ne semble pas avoir été réalisée, *ou bien*

② Le programme ne semble pas s'exécuter dans l'ordre des instructions de **`writer()`**.

Modèle de consistance mémoire

Principe de la consistance séquentielle (par L. Lamport, prix Turing 2013)

≪ *... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.* ≫

Leslie Lamport, IEEE Trans. Comput. C-28,9 (1979), 690-691,

How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.

Exemples :

- La trace d'exécution séquentielle **`a` ← `3; b` ← `4; "b=4, "; "a=3"`** est séquentiellement consistante.
- La trace d'exécution séquentielle **`a` ← `3; b` ← `4; "b=4, "; "a=1"`** **n'est pas** séquentiellement consistante.

Aucune trace d'exécution affichant **`"b=4, a=1"`** n'est séquentiellement consistante.

Définition de la consistance séquentielle

La **consistance séquentielle** exige que le résultat d'un programme *corresponde toujours apparemment* à un calcul *séquentiel* du programme, dans lequel chaque processus exécute ses instructions (et donc ses opérations sur la mémoire) dans *l'ordre indiqué dans son code*, les unes après les autres, et chaque instruction *voit les effets de toutes les instructions précédentes*.

Autrement dit, il existe un **ordre total** sur les instructions exécutées par le programme (appelé une *trace d'exécution*) tel que

① Les instructions du code de chaque thread ne sont pas permutées !

② Chaque modification en mémoire est visible par les actions ultérieures.

un peu comme si le programme était exécuté sur une machine monoprocesseur, sans cache, ni pipeline.

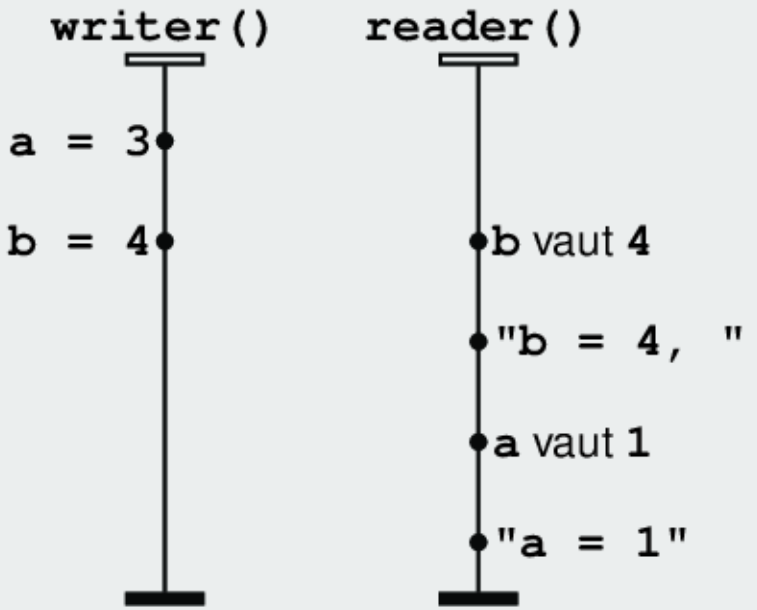
Mauvaise nouvelle (nous venons de le voir...) :

Le modèle mémoire Java *ne garantit pas* la consistance séquentielle de toutes les exécutions.

- ✔ *Qu'est-ce que la consistance séquentielle ?*
- ☛ *La notion de data-race*

Data-race et programmes bien synchronisés en Java

Une *exécution légale* d'un programme Java contient une **data-race** si elle comporte une première instruction de lecture ou d'écriture sur une variable *non volatile (ni atomique)* et une seconde instruction d'écriture *sur cette même variable* qui ne sont pas reliées par la relation ≪ happens before ≫.



Jargon :

On dit qu'un programme est ≪ **bien synchronisé** ≫ ou **DRF (pour ≪ data-race free ≫)** si aucune de ses exécutions légales ne contient de *data-race*.

Garantie des programmes bien synchronisés

Bonne nouvelle :

≪ Le modèle mémoire Java garantit que les exécutions d'un programme **sans data-race** sont toutes **séquentiellement consistantes !** ≫

C'est bien ce que l'on veut !

Garantie des programmes bien synchronisés

Bonne nouvelle :

> « Le modèle mémoire Java garantit que les exécutions d'un programme **sans data-race** sont toutes **séquentiellement consistantes !** »

C'est bien ce que l'on veut !

Ce que ça veut dire en pratique.

Si la seule explication possible d'un résultat inattendu du programme est que

- les écritures n'ont pas été correctement réalisées en mémoire, ou
- les instructions du programme n'ont pas été réalisées dans l'ordre

c'est-à-dire que l'exécution observée n'est pas *séquentiellement consistante*, alors le programme contient **nécessairement** une *data-race*.

Ça peut aider pour débugger !

Au-delà de la garantie DRF

Un programme produira uniquement des exécutions **séquentiellement consistantes** s'il satisfait l'une des conditions suivantes :

① Chaque variable partagée est manipulée systématiquement avec un verrou donné.

② Le programme ne contient pas de data-race. **Pas très pratique !**

③ Chaque variable partagée est ou bien déclarée *volatile*, ou bien dans un objet ou un tableau *atomique*, ou bien protégée avec un *verrou*. **Plus simple !**

Néanmoins,

- pour démontrer ces affirmations, il faut être un expert du modèle mémoire Java...
- pour appliquer correctement la seconde règle, il faut aussi être un expert du modèle mémoire et pouvoir affirmer que l'on a considéré toutes les exécutions potentielles...
- l'absence de data-race peut aussi résulter simplement de l'emploi des méthodes `start()`, `join()`, `submit()`, ou encore `get()` appliquée à un objet `Future`.

Que veut dire le mot `final` ?

Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

Cas particulier des champs déclarés `final`

Un champ **`final`** ne peut voir sa valeur fixée qu'une seule fois, et ne peut plus voir sa valeur changée une fois totalement exécuté le constructeur de l'objet qui le porte.

Premier cas

```
public class MaClasse {
  private final int monChamp = 3;
...
```

Second cas

```
public class MaClasse {
  private final int monChamp;
  public MaClasse() {
    ...
    monChamp = 3;
    ...
```

Cas particulier des champs déclarés `final`

≪ *Un champ* ***`final`*** *ne peut voir sa valeur fixée qu'une seule fois, et ne peut plus voir sa valeur changée une fois totalement exécuté le constructeur de l'objet qui le porte.* ≫

De plus, la javadoc dit :

≪ Fields declared final are initialized once, but never changed under normal circumstances ≫.

Néanmoins il existe des circonstances anormales !

Que veut dire le mot `final` ?

`https://docs.oracle.com/javase/specs/...`:

≪ An object is considered to be completely initialized when its constructor finishes. A thread that can only see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's final fields. ≫

Selon le modèle mémoire, quand la construction d'un objet est terminée, la référence de cet objet devient disponible pour le thread qui crée l'objet. De plus, les valeurs (définitives) des champs déclarés **`final`** sont alors visibles par tout thread qui accède à cet objet.

Ainsi, *une fois l'objet construit*, le champ **`final`** peut être accédé par différents threads *sans synchronisation*.

Une situation de data-race

```
class A {
  int f;
  public A() { f = 42 ; }
}
class B {
  A a;
  static void writer() { a = new A() ; }
  static void reader() { if ( a != null ) println(a.f) ; }
}
```

Deux threads appliquent simultanément les deux méthodes **`writer()`** et **`reader()`** sur un même objet de la classe B.

Que va afficher le second thread ?

Réponse

```
class A {
  int f;
  public A() { f = 42 ; }
}
class B {
  A a;   // n'est pas déclarée volatile! Il y a une data-race
  static void writer() { a = new A() ; }
  static void reader() { if ( a != null ) println(a.f) ; }
}
```

Le thread appliquant **`reader()`** pourra afficher :

- — **`"42"`**, si le premier thread est suffisamment rapide pour construire **`a`** ;
- — rien, si le second thread est trop rapide pour voir l'objet **`a`** construit ;
- — **`"0"`**, car rien n'assure qu'il voit l'objet **`a`** complètement, même s'il est construit ;
- — ou encore : **`NullPointerException`** car rien n'assure que la seconde lecture de la référence **`a`** ne renvoie pas **`null`**...

Code corrigé

```
class A {
  final int f;
  public A() { f = 42 ; }
}
class B {
  A a;
  static void writer() { a = new A() ; }
  static void reader() {
    A ta = a ;
    if ( ta != null ) println(ta.f) ;
  }
}
```

Le thread appliquant **`reader()`** pourra maintenant uniquement afficher :

- rien, s'il est trop rapide pour observer que l'objet **`a`** a été construit ;
- ou **`"42"`**, si le premier thread est suffisamment rapide pour construire **`a`**.

Comparaison avec volatile

```
class A {
  int e ;
  final int f ;
  public A() { e = 21 ; f = 42 ; }
}
class B {
  A a;
  static void writer() { a = new A() ; }
  static void reader() {
    A ta = a ;
    if ( ta != null ) println(ta.f + "␣" + ta.e) ;
  }
}
```

Le thread appliquant **`reader()`** pourra afficher **`"42 0"`**, car seul **`f`** est déclaré **`final`** : aucune garantie n'est accordée à **`e`** même s'il semble initialisé *avant* **`f`**.

Fausse bonne idée : remplacer final par volatile

```
class A {
  volatile int f;
  public A() { f = 42 ; }
}
class B {
  A a;
  static void writer() { a = new A() ; }
  static void reader() {
    A ta = a ;
    if ( ta != null ) println(ta.f) ;
  }
}
```

Le thread appliquant **`reader()`** pourra afficher **`"0"`**, car il n'y a aucune synchronisation entre les deux threads, le champ **`a`** n'étant pas lui-même déclaré **`volatile`**.

Mauvaise pratique : publication prématurée de `this`

```
class A {
  final int f;
  public A(B b) { f = 42 ; b.a = this ; }
}
class B {
  A a;
  static void writer() { new A(this) ; }
  static void reader() {
    A ta = a ;
    if ( ta != null ) println(ta.f) ;
  }
}
```

Le thread appliquant **`reader()`** pourra afficher **`"0"`**, car la référence **`a`** est potentiellement disponible avant la fin de la construction de l'objet ! Le mot-clef **`final`** est alors sans effet.

Les origines des inconsistances

Master Informatique — Semestre 2 — UE obligatoire de 3 crédits

Année 2017-2018

Version du 21 février 2018

À quoi sert le modèle mémoire Java ?

Les sources multiples du problème :

① Les compilateurs peuvent **réordonner** les instructions.

② Les **processeurs** peuvent aussi réordonner les instructions !

③ Parfois les nouvelles valeurs des variables sont **retenues** dans les caches.

Le modèle mémoire d'un langage détermine (plus ou moins directement) le type de manipulations autorisées sur le code et sur les mises-à-jour des variables, et donc l'ensemble de *ses résultats légaux*.

Il ne faut pas confondre !

Le *Java Language Specification* et le *Java Virtual Machine Specification* ont des objectifs bien distincts :

- le premier détermine ce que sont les **exécutions légales** d'un programme, à l'aide du *modèle mémoire* qui s'appuie sur des **ordres partiels**.
- le second guide les concepteurs de machines virtuelles en autorisant certaines **permutations d'instructions**.

En revanche, certaines *permutations* avec les instructions `lock()` et `unlock()` sont bien connues pour être formellement interdites au compilateur. C'est le cas aussi avec les accès mémoire à une variable déclarée `volatile` ou à des objets atomiques.

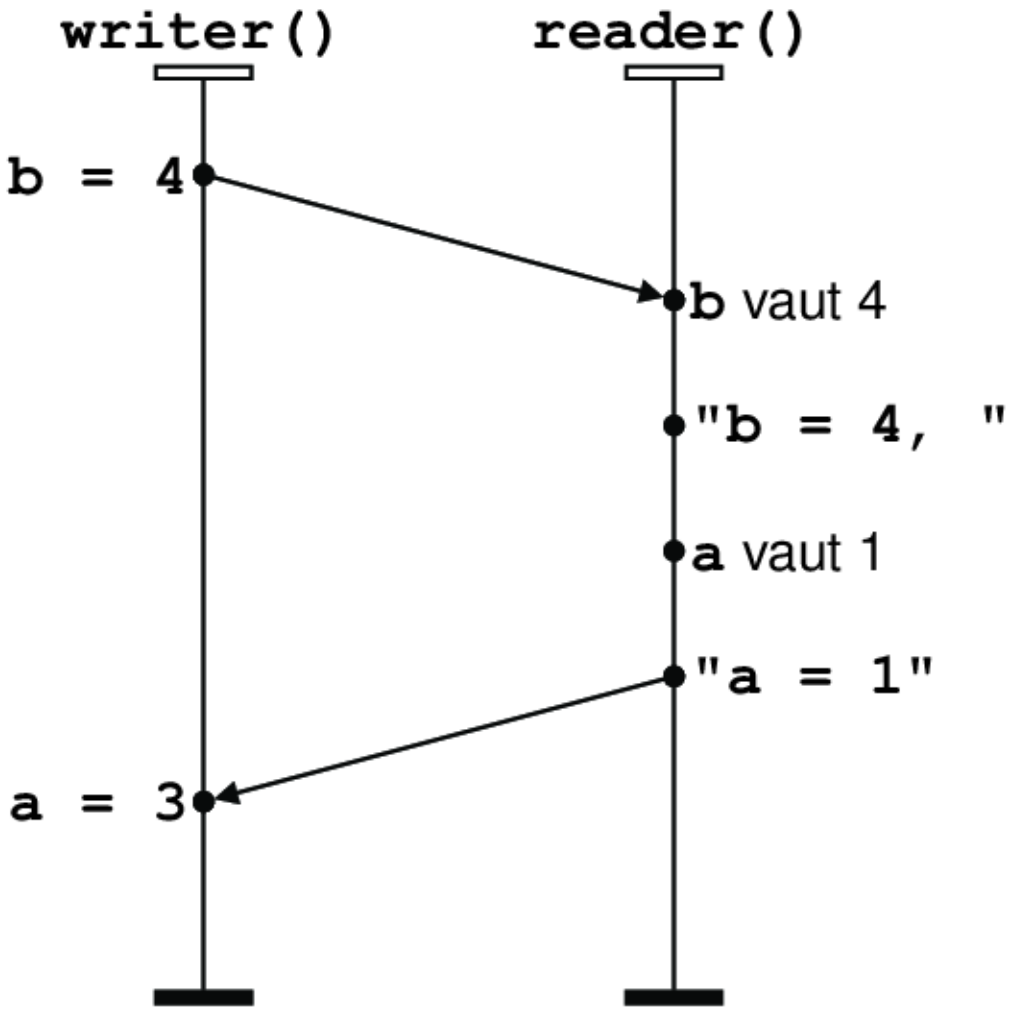
Une autre explication

```
class NotSoSimple {
  int a = 1, b = 2;
  void writer() {
    a = 3;
    b = 4;
  }
  void reader() {
    System.out.print("b = " + b +", ");
    System.out.println("a = " + a);
  }
}
```

D'une certaine manière, le modèle mémoire Java autorise la **permutation** des deux instructions `a = 3` et `b = 4`. Ceci conduit à la sortie `"b = 4, a = 1"`.

Après réordonnancement autorisé du code

Initialement a=1 et b=2



✔ *Les optimisations de code*

☛ *Le rôle des mémoires caches*

La loi de Moore

L'industrie informatique s'est développée depuis 40 ans en s'appuyant sur la loi de Moore, qui incite à changer d'ordinateur tous les deux ans.

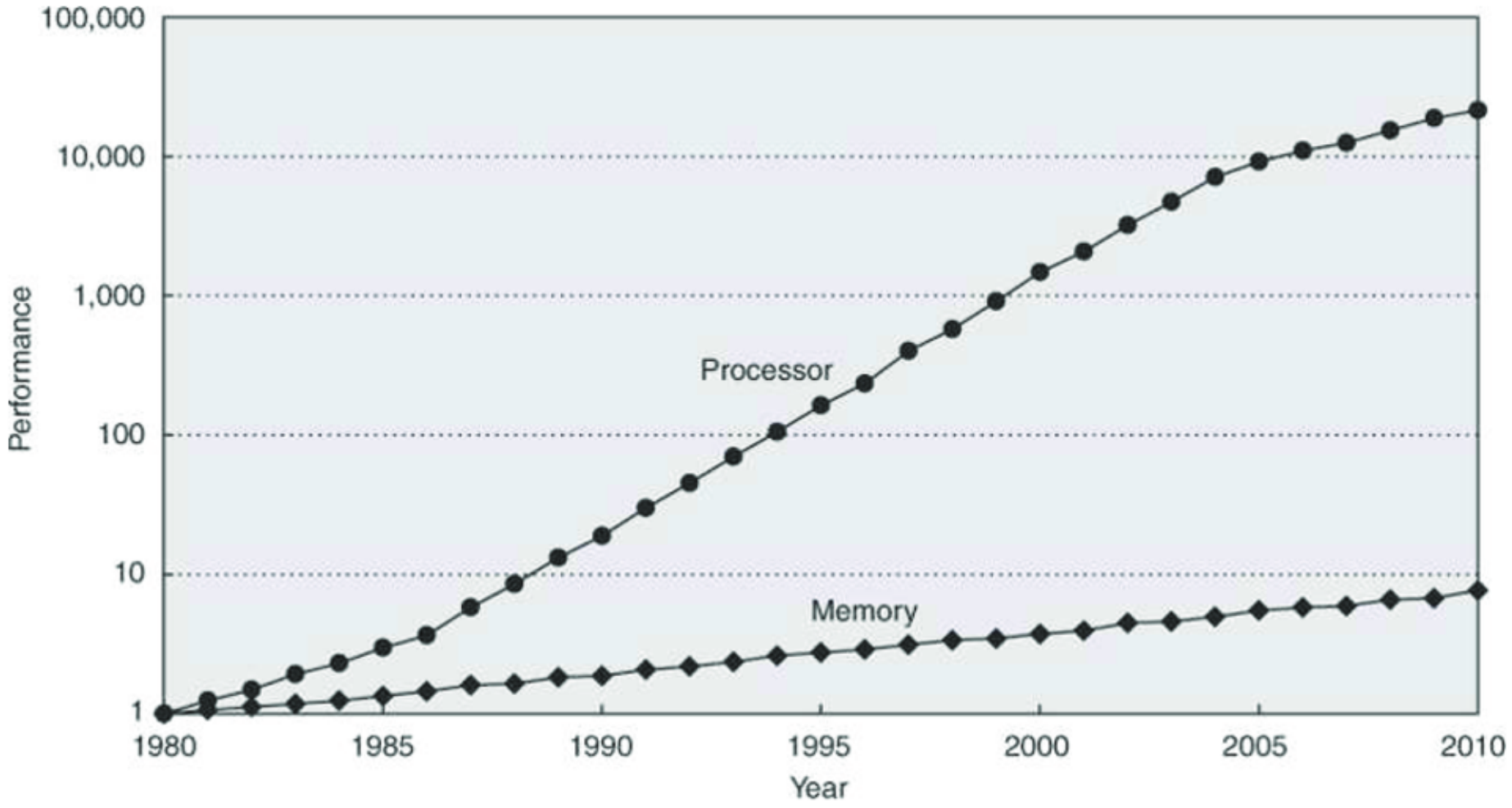
En 2004, la perspective de poursuivre la progression des performances selon le modèle classique du monoprocesseur est abandonnée par Intel.

> *≪ Intel said on Friday that it was scrapping its development of two microprocessors, a move that is a shift in the company's business strategy... ≫*
>
> SAN FRANCISCO, May 7. 2004, New York Times

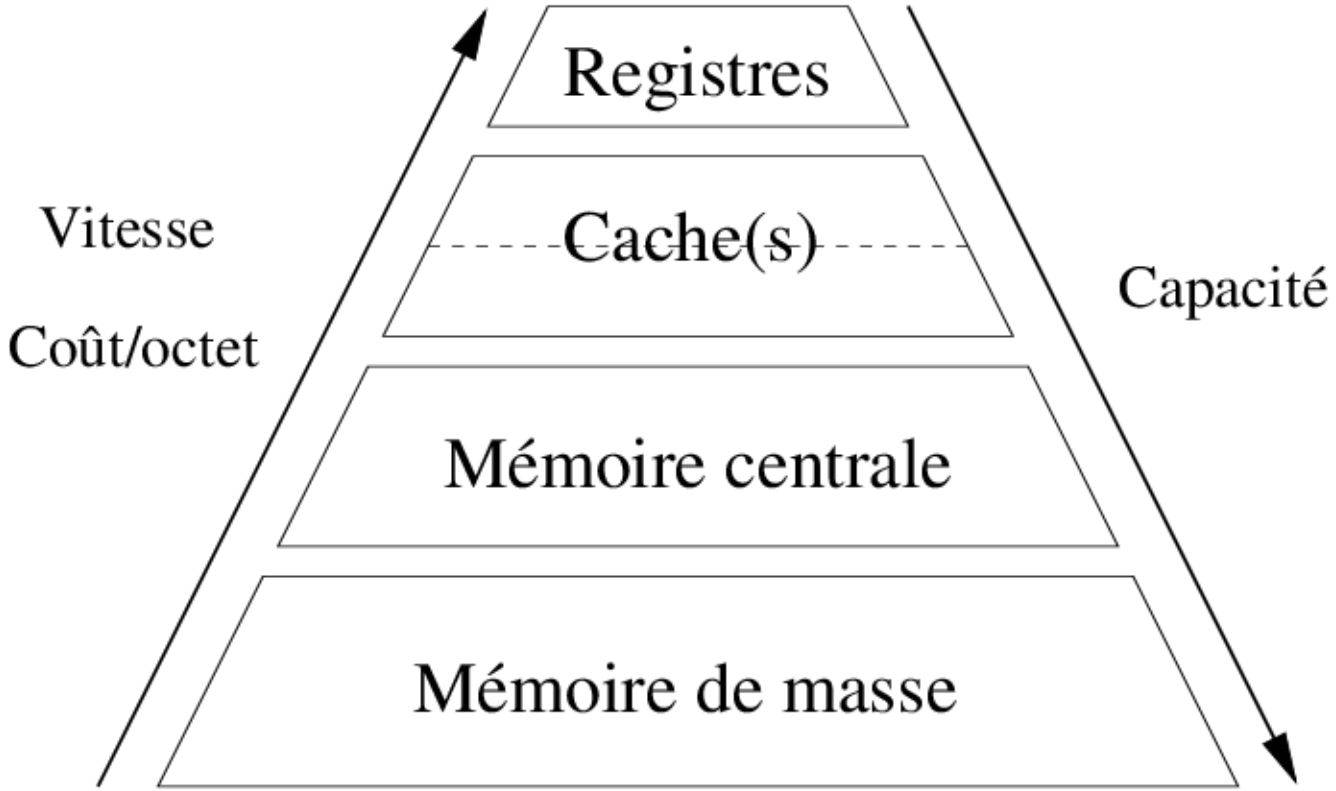
Depuis lors, la fréquence d'horloge des microprocesseurs s'est stabilisée et la loi de Moore n'est maintenue que par la multiplication du nombre de coeurs.

Le ≪ mur de la mémoire ≫

Cependant les performances de la *mémoire principale* progressent beaucoup moins vite que celles des microprocesseurs.



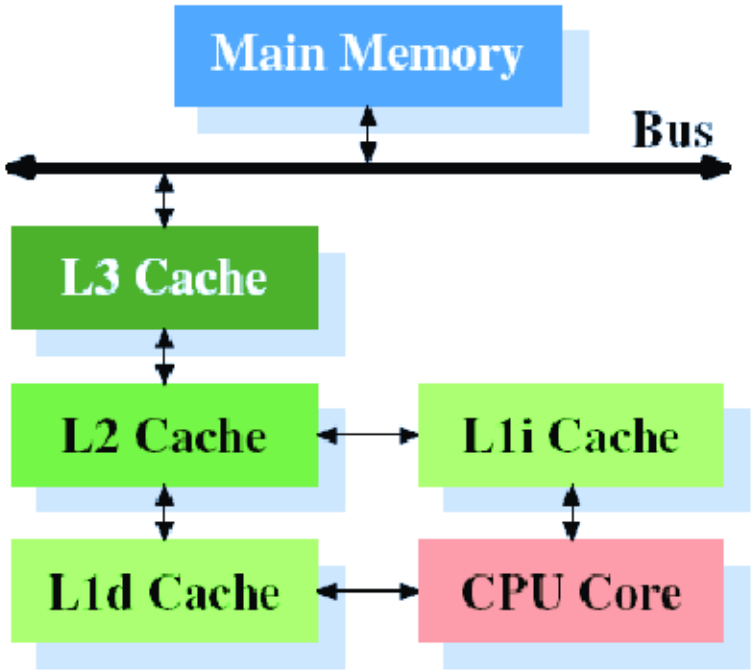
Hiérarchie de mémoire



Organisation et utilisation des caches mémoires

Temps d'accès à une donnée en cycles d'horloge :

- Registre $<$ 1 ns
- L1 $\simeq$ 3 ns
- L2 $\simeq$ 15 ns
- L3 $\simeq$ 50 ns
- RAM $\simeq$ 200 ns



Si la donnée n'est pas dans les caches, il faudra attendre 200 cycles d'horloge pour la récupérer dans un registre du coeur.