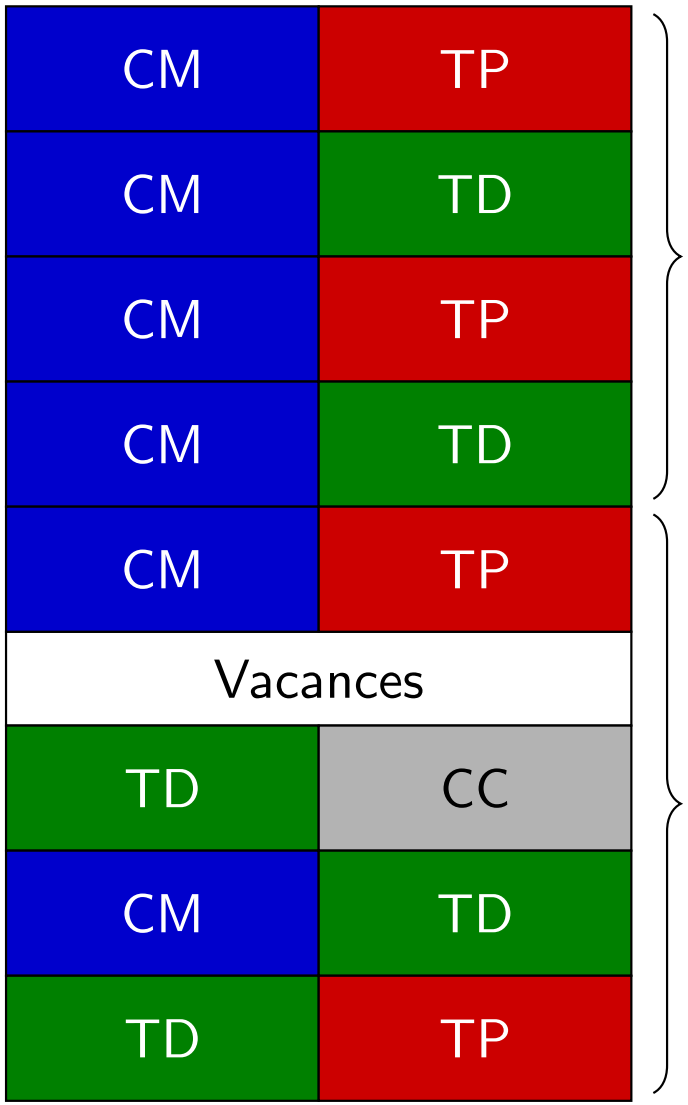
Programmation multi-cœurs

Initiation à la programmation et algorithmique concurrente multi-threads
Master en Informatique, première année

Matthieu Perrin
Laboratoire des Sciences du Numérique de Nantes, UMR CNRS 6004
Bureau 410, bâtiment 34
matthieu.perrin@univ-nantes.fr

2022-2023

Organisation (le programme peut changer)



Programmation (en Java)

- Gestion des threads
- Synchronisation bloquante
- Modèles de mémoire

Algorithmique

- Algorithmes non-bloquants
- Calculabilité du réparti
- Mémoire transactionelle logicielle (projet)

Bibliographie

Programmation

- docs.oracle.com/javase/tutorial/essential/concurrency/index.html
- https://jenkov.com/tutorials/java-concurrency/index.html
- *Concurrent Programming in Java : Design Principles and Patterns*
  Doug Lea (Addison-Wesley 1996)

Algorithmique

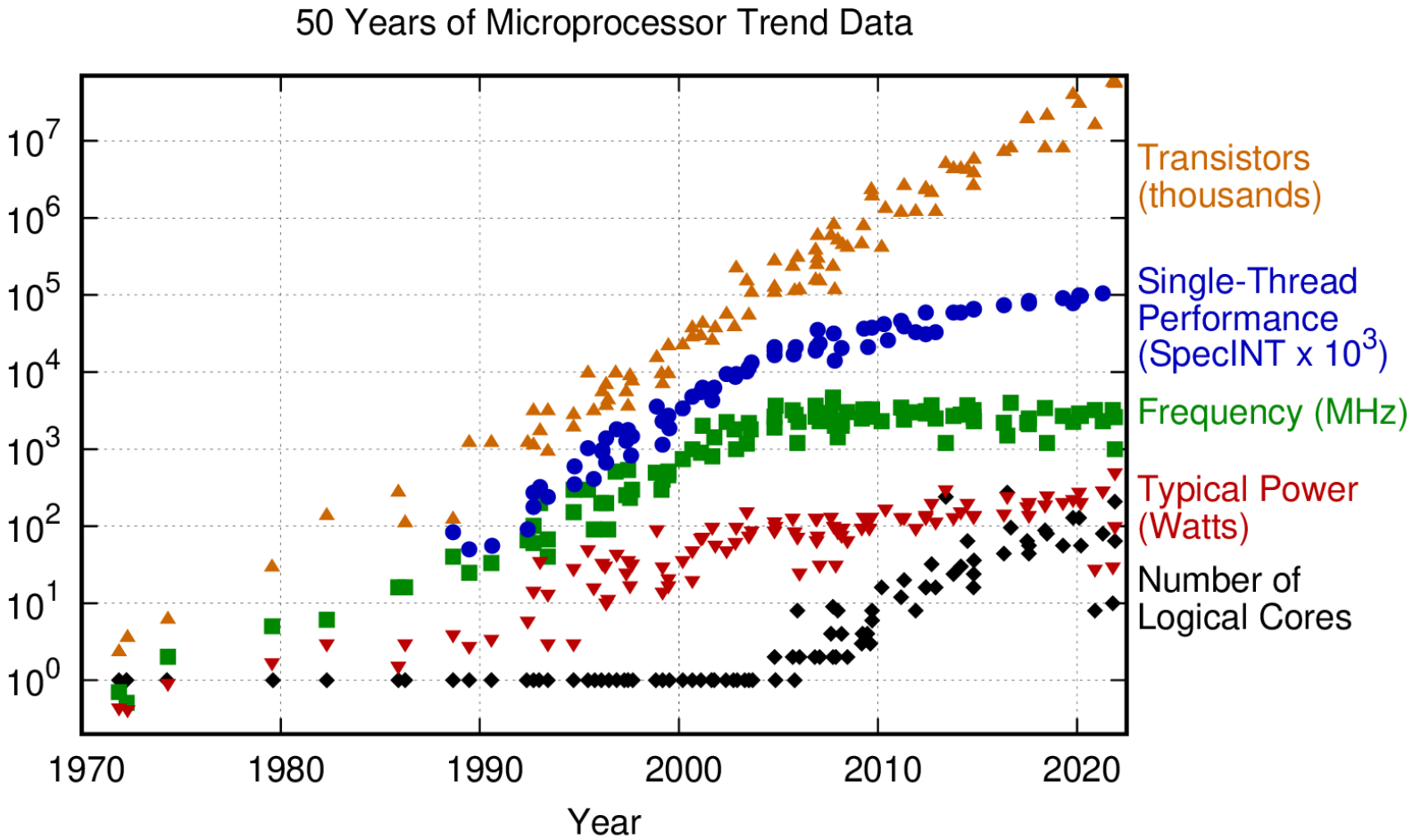
- *The Art of Multiprocessor Programming*
  Maurice Herlihy & Nir Shavit (Morgan Kaufmann 2008)
- *Concurrent Programming : Algorithms, Principles, and Foundations*
  Michel Raynal (Springer 2013)

Sources Les articles et ouvrages sont sur Madoc.

Du parallélisme à la concurrence

- Motivations
- Programmation parallèle
- Spécification d'un problème concurrent
- Introduction à la concurrence

Fin de la loi de Moore ?

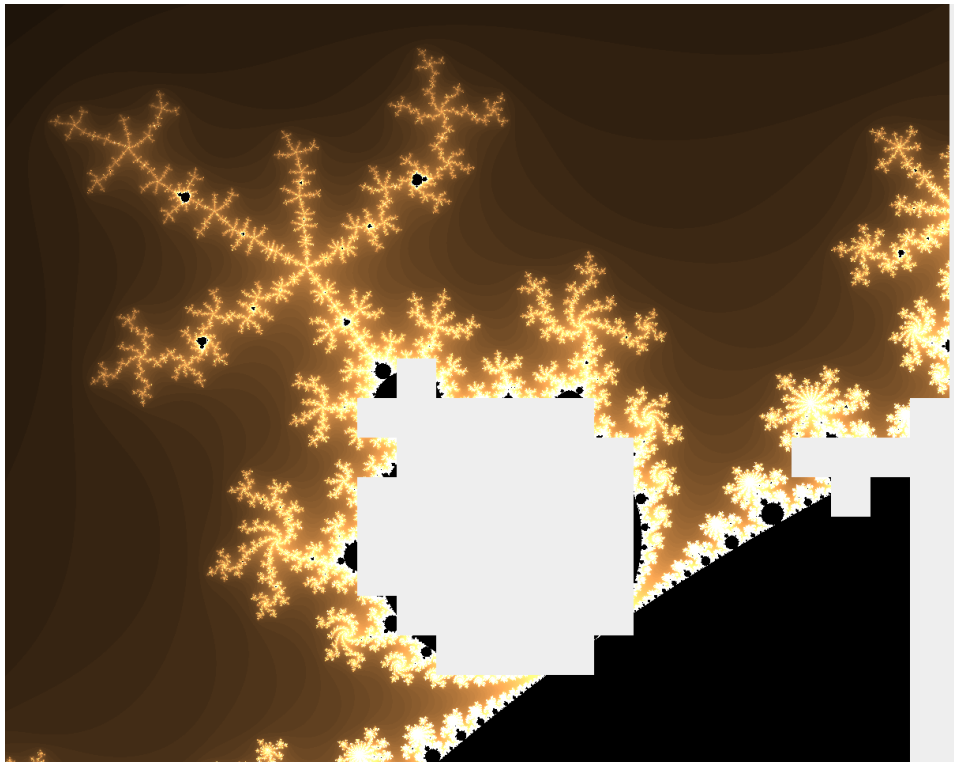


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Pourquoi s'intéresser à la programmation multi-cœurs ?

Calculs coûteux

- En temps de processeur (distanciel, simulations climatiques, . . .)



Pourquoi s'intéresser à la programmation multi-cœurs ?

Calculs coûteux

- En temps de processeur (distanciel, simulations climatiques, ...)
- En entrées/sorties (TP3, recherche dans les fichiers, ...)

```
$ java WebGrep Nantes https://fr.wikipedia.org/wiki/Nantes
https://fr.wikipedia.org/wiki/Nantes
https://fr.wikiquote.org/wiki/Nantes
https://fr.wiktionary.org/wiki/Nantes
https://fr.wikivoyage.org/wiki/Nantes
...
```

Pourquoi s'intéresser à la programmation multi-cœurs ?

Calculs coûteux

- En temps de processeur (distanciel, simulations climatiques, ...)
- En entrées/sorties (TP3, recherche dans les fichiers, ...)

Activités coopérantes

- Interface réactive (compilation à la volée, ...)
- Traitement de requêtes sur un serveur, ...

Stratégie de parallélisation

- Découper le problème en tâches indépendantes
- Associer des tâches à chaque cœur
- S'assurer que les cœurs collaborent correctement

Approches pour la programmation concurrente

Processus versus thread

- Informatique = traitement automatique de l'information
- Le système d'exploitation abstrait le processeur et la mémoire
  - Processus : contexte d'exécution d'un programme (partage de mémoire)
  - Thread : description d'une exécution séquentielle (partage du processeur)
- Chaque processus contient un ou plusieurs threads

Programmation multi-processus : plusieurs processus, un thread par processus

Robustesse : les processus sont isolés les uns des autres par l'OS

Répartition : on peut répartir les processus sur un réseau de machines

Programmation multi-threads : un processus, plusieurs threads

Efficacité : données et références partagées

Programmation : intégration presque transparente au sein des langages

Intégration aux langages de programmation

Java

La JVM est une machine multi-thread. Le `GC` est un thread.

API historique Définition de tâches, de threads et outils basiques pour la synchronisation

Depuis Java 5 La bibliothèque `java.util.concurrent` repose en grande partie sur les travaux de Doug Lea (`http://gee.cs.oswego.edu/dl/index.html`)

C++

Donne accès aux threads du système.

Avant C++11 accès direct aux threads système (POSIX pthread et WINAPI thread depuis les années 1990).

Depuis C++11 intégration d'une API à la bibliothèque standard.

Tâche

- Code exécuté par un thread.

C++

N'importe quelle fonction (pointeur de fonction)

```
void task () {
  cout << "hello world" << endl;
}
```

Java

Implémentation de l'interface `Runnable`

```
public class Task implements Runnable {
  public void run () {
    System.out.println("hello world");
  }
}
```

Tâches et contexte d'exécution

Création et lancement d'un thread `t`

Java : classe `java.lang.Thread`

```
Thread t = new Thread(new Task());
t.start();
```

C++ : classe `std::thread`

```
thread t(task);
// Started by constructor
```

Tâches et contexte d'exécution

Création et lancement d'un thread `t`

Java : classe `java.lang.Thread`

```
Thread t = new Thread(new Task());
t.start();
```

C++ : classe `std::thread`

```
thread t(task);
// Started by constructor
```

- Pousse à faire une association $1 \leftrightarrow 1$ entre tâches et unité d'exécution.

Distanciel (à faire avant le TP 3)

Thread pools

- Constructions simples d'associations $n \leftrightarrow m$ avec différentes stratégies.

Vous apprendrez à :

- utiliser des thread pools,
- implémenter vos propres thread pools.

Principales méthodes de la classe gérant les threads

Obtenir la référence sur le thread qui contrôle la tâche en cours

Java : `Thread.currentThread()`

C++ : `std::this_thread`

Principales méthodes de la classe gérant les threads

Obtenir la référence sur le thread qui contrôle la tâche en cours

Java : `Thread.currentThread()` C++ : `std::this_thread`

Endormir le thread `t` pour `ms` millisecondes

Java : `t.sleep(ms)` C++ : `t.sleep_for(std::chrono::milliseconds(ms))`

Principales méthodes de la classe gérant les threads

Obtenir la référence sur le thread qui contrôle la tâche en cours

Java : `Thread.currentThread()` C++ : `std::this_thread`

Endormir le thread `t` pour `ms` millisecondes

Java : `t.sleep(ms)` C++ : `t.sleep_for(std::chrono::milliseconds(ms))`

Attente de la terminaison d'un thread `t`

Java ou C++ : `t.join()`

attention : en C++, un thread doit être détaché avant la terminaison du thread qui l'a lancé (`t.detach()`).

Interrompre proprement un thread

Forcer un thread à s'arrêter

Impossible : `public void stop()` est dépréciée !

`public void interrupt()`

Demande la terminaison.

- Lève un drapeau booléen pour avertir le thread.
- La tâche peut (doit) vérifier régulièrement les demandes d'interruption
  - `public static boolean interrupted()` ou
  - `public boolean isInterrupted()`.
- `interrupted()` et `isInterrupted()` rebaissent le drapeau.
- Si le thread est bloqué ou en attente, `InterruptedException` est lancée.

Pas d'équivalent en C++

Utiliser une variable booléene comme drapeau.

Non déterminisme des sorties du programme

Asynchronisme

Il est impossible de prévoir la vitesse d'exécution :

- d'un thread par rapport à un autre,
- d'une portion de code exécutée par un thread par rapport à une autre,
- d'une exécution d'un thread par rapport à une autre.

Non déterminisme des sorties du programme

Asynchronisme

Il est impossible de prévoir la vitesse d'exécution :

- d'un thread par rapport à un autre,
- d'une portion de code exécutée par un thread par rapport à une autre,
- d'une exécution d'un thread par rapport à une autre.

Exécution concurrente

À chaque instant, la configuration globale d'un système est défini par :

- L'ensemble des threads s'exécutant
- L'état local de chaque thread
- L'état de la mémoire

Un ordonnanceur décide quel thread exécutera le prochain pas atomique.

- Un programme est correct si toutes ses exécutions sont correctes

Spécification d'un problème réparti

Sûreté (Safety)

- Rien de mal ne se produit

$$\forall t.P(t)$$

- Thread-safe : fonctionne correctement en multi-thread (très imprécis)

Vivacité (Liveness)

- Quelque chose de bien finira par se produire

$$\exists t.P(t)$$

Exemple : programme qui calcule π

Sûreté : À tout instant, un préfixe de l'écriture décimale de π est affiché.

Vivacité : Pour tout n, au moins n chiffres finiront par être affichés.

Sûreté et vivacité

Infirmez ou confirmez les affirmations suivante

1. Tous les empires sont des agglomérats de royaumes.
2. Tous les empires finissent par s'effondrer.

Propriété de sûreté ou de vivacité ?

1. Si deux voitures attendent à une intersection, l'une d'elle va finir par passer.
2. Il n'y aura pas d'accident à l'intersection.
3. Le feu va passer au vert.
4. Le feu va passer au vert dans les cinq prochaines minutes.
5. Seules deux choses sont certaines : la mort et les impôts.

Problématique du cours

"For many of the applications you may wish to parallelize, you will find that there are significant parts that can easily be determined as executable in parallel because they do not require any form of coordination or communication. However, (...) there is no cookbook recipe for identifying these parts. This is where the application designer needs to use his or her accumulated understanding of the algorithm being parallelized. Luckily, in many cases it is obvious how to find such parts.

The more substantial problem (...) is how to deal with the remaining parts of the program. As noted earlier, these are the parts that cannot be easily parallelized because the program must access shared data and requires interprocess coordination and communication in an essential way."

Maurice Herlihy & Nir Shavit

Parallélisme versus concurrence

Principale difficulté

Parallélisme : Identifier les tâches indépendantes

Concurrence : Gérer l'incertitude

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Leslie Lamport

R15 Michel Raynal. *Parallel Computing vs. Distributed Computing : A Great Confusion ? (Position Paper).* Euro-Par Workshops (2015).

Parallélisme versus concurrence

Principale difficulté

Parallélisme : Identifier les tâches indépendantes

Concurrence : Gérer l'incertitude

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Leslie Lamport

Contexte d'exécution

Parallélisme : Choisi en fonction du problème à résoudre

Concurrence : Imposé *a priori* par le problème

R15 Michel Raynal. *Parallel Computing vs. Distributed Computing : A Great Confusion ? (Position Paper).* Euro-Par Workshops (2015).

Parallélisme versus concurrence

Principale difficulté

Parallélisme : Identifier les tâches indépendantes

Concurrence : Gérer l'incertitude

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Leslie Lamport

Contexte d'exécution

Parallélisme : Choisi en fonction du problème à résoudre

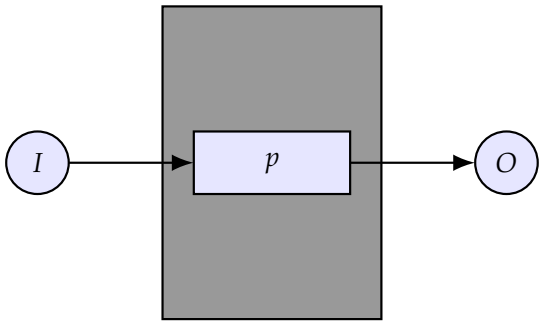
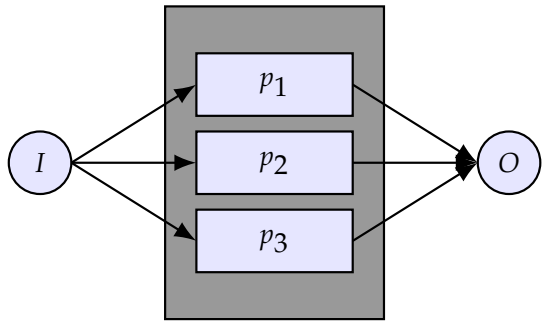
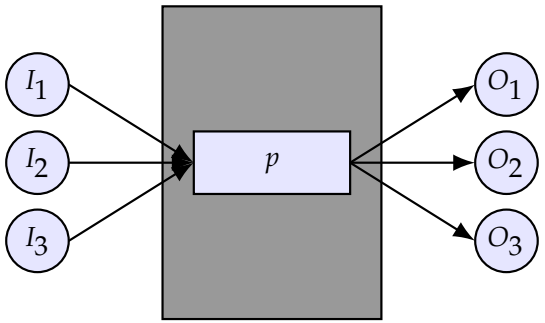
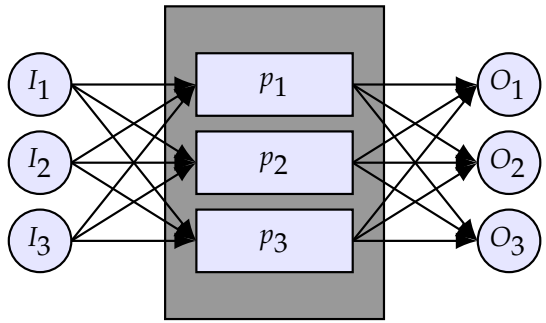
Concurrence : Imposé *a priori* par le problème

Passage à l'échelle

Parallélisme : $S_p(n) = \Omega(n)$ Concurrence : $S_c(n) = \Omega(1)$

R15 Michel Raynal. *Parallel Computing vs. Distributed Computing : A Great Confusion ? (Position Paper).* Euro-Par Workshops (2015).

Point vocabulaire

	Une seule unité d'exécution	Plusieurs unités d'exécution
Problème Séquentiel	 Séquentiel	 Parallèle
Problème de concurrence	 Centralisé	 Réparti (distributed)

Exercice

Problème de parallélisme ou de concurrence ?

1. Éviter les collisions sur la route.
2. Optimiser la vitesse de production d'une usine (Fordisme).
3. Élire le président de la république.
4. Massively multiplayer online role-playing game.
5. Prévoir la météo.
6. Les projets semestriels en groupe.
7. Un orchestre symphonique.

Synchronisation bloquante

- Section critiques et verrouillage
- Problèmes de vivacité
- Signalement

Problème de concurrence

TP 1

1. Récupérez le programme `SharedCounter.java`.
2. Exécutez le programme en faisant varier la valeur du paramètre : 10, 100, 1000, 10000, 100000. Que constatez-vous ?
3. Ajoutez le mot-clé `synchronized` et ré-exécutez le programme.

Problème de concurrence

TP 1

1. Récupérez le programme `SharedCounter.java`.
2. Exécutez le programme en faisant varier la valeur du paramètre : 10, 100, 1000, 10000, 100000. Que constatez-vous ?
3. Ajoutez le mot-clé `synchronized` et ré-exécutez le programme.

Explications ?

```
void increment();
    Code:
       0: aload_0
       1: dup
       2: getfield      #2                  // Field value:I
       5: iconst_1
       6: iadd
       7: putfield      #2                  // Field value:I
      10: return
```

Point vocabulaire

Section critique

Séquence d'instructions qui manipule des données partagées.

Point vocabulaire

Section critique

Séquence d'instructions qui manipule des données partagées.

Exclusion mutuelle

Prédicat logique « à tout instant, il y a au plus un thread en section critique ».

Point vocabulaire

Section critique

Séquence d'instructions qui manipule des données partagées.

Exclusion mutuelle

Prédicat logique « à tout instant, il y a au plus un thread en section critique ».

Verrou

Objet informatique qui ne peut être possédé que par un thread à la fois.

Point vocabulaire

Section critique

Séquence d'instructions qui manipule des données partagées.

Exclusion mutuelle

Prédicat logique « à tout instant, il y a au plus un thread en section critique ».

Verrou

Objet informatique qui ne peut être possédé que par un thread à la fois.

Attention

Un verrou (objet informatique) protège une section critique (portion de code) pour en assurer l'exclusion mutuelle (propriété de sûreté).

Assurer l'exclusion mutuelle grâce aux verrous

Java

```java
import java.util.concurrent.locks.*;

int value = 0;
Lock lock = new ReentrantLock();

void increment() {
  lock.lock();
  // Section critique
  ++value;
  ////////////////////
  lock.unlock();
}
```

C++

```cpp
#include <mutex>

int value = 0;
mutex lock;

void increment() {
  lock.lock();
  // Section critique
  ++value;
  ////////////////////
  lock.unlock();
}
```

Assurer l'exclusion mutuelle grâce aux verrous

Java

```java
import java.util.concurrent.locks.*;

int value = 0;
Lock lock = new ReentrantLock();

void increment() {
  lock.lock();
  // Section critique
  ++value;
  ////////////////////
  lock.unlock();
}
```

C++

```cpp
#include <mutex>

int value = 0;
mutex lock;

void increment() {
  lock.lock();
  // Section critique
  ++value;
  ////////////////////
  lock.unlock();
}
```

Attention au mot "Mutex"

Origine : contraction de "mutual exclusion"

En C++ : synonyme de "verrou"

Thread quittant du code en exclusion mutuelle

Quand un thread quitte une exclusion mutuelle

Deux cas de figure pour le verrou de l'objet :

- Il passé automatiquement à l'un des threads bloqués (ordre indéterminé).
- Il est libéré si aucun thread n'est bloqué.

Avant la terminaison d'un Thread

Il faut :

- libérer tous les verrous en sa possession.
- être certain qu'il laisse les données partagées dans un état cohérent.
- être certain qu'aucune autre tâche n'attend une action de sa part.

Mot-clé synchronized en Java

Moniteur d'exécution en Java

Un verrou est associé à chaque :

- objet instancié dans la JVM,
- classe.

Mot-clé synchronized

- Méthode déclarée synchronized :
  le verrou est pris pour la durée de l'exécution de la méthode.
- Bloc synchronized{} :
  le verrou est pris pour la durée de l'exécution du bloc.

Mot-clé synchronized en Java

Moniteur d'exécution en Java

Un verrou est associé à chaque :

- objet instancié dans la JVM,
- classe.

Mot-clé synchronized

- Méthode déclarée synchronized :
  le verrou est pris pour la durée de l'exécution de la méthode.
- Bloc synchronized{} :
  le verrou est pris pour la durée de l'exécution du bloc.

Avantages

- Simple d'utilisation.
- Mise en œuvre efficace.

Inconvénients

- Peu flexible dans l'utilisation.
- Une seule sorte de verrou

Notion de réentrance

TP 1

1. Étudier le programme `Reentrance.java`.
2. Exécuter le programme. Que remarquez-vous ?
3. À la ligne 5, remplacer `NonReentrantLock` par `ReentrantLock`. Que constatez-vous ?

Bonne nouvelle

Les blocks `synchronized` et les verrous C++ sont réentrants.

Verrous à lectures et écritures

En java (`java.util.concurrent`)

```
ReadWriteLock lock = new ReentrantReadWriteLock();
```

- `Lock writeLock = lock.writeLock();`
  1. Récupère le verrou exclusif pour protéger les écritures
- `Lock readLock = lock.readLock();`
  1. Récupère le verrou non exclusif pour protéger les lectures

En C++ (`boost::thread` avant C++ 17, `std` depuis C++ 17)

```
shared_mutex lock;
```

- `unique_lock<shared_mutex> writeLock(lock);`
  1. Crée un verrou exclusif pour protéger les écritures
- `shared_lock<shared_mutex> readLock(lock);`
  1. Crée un verrou non exclusif pour protéger les lectures

Sémaphore

TP 1

1. Récupérer le programme `Semaphores.java`.
2. Étudier ce programme.
3. Exécuter le programme en faisant varier la valeur du paramètre : 6, puis 4, puis 2. Que constatez-vous ?

Edsger W. Dijkstra

D62 Edsger W. Dijkstra. *Over de sequentialiteit van procesbeschrijvingen (About the sequentiality of process descriptions)* (1962-1963)

Grain de verrouillage

Verrouillage à gros grains

- Un seul verrou pour une structure de données
- Chaque méthode est entièrement protégée par le verrou
- Avantage : très simple à mettre en place
- Inconvénient : très inefficace

Verrouillage à grains fins

- Plusieurs verrous pour une structure de données
- Chaque verrou protège une partie des données
- Attention à quels verrous doivent être pris

Un nouveau problème

Le mécanisme de verrou utilisé pour résoudre notre problème peut être lui même la source d'autres problèmes, plus difficiles encore à résoudre

TP 1

1. Récupérer le programme `Friend.java`
2. Exécuter le programme. Que constatez-vous ?

Interblocage circulaire

Deadlock

Situation d'interblocage circulaire où des threads attendent entre eux la libération de ressources avant de continuer leur exécution.

Une solution pour éviter les deadlocks

Composition

Des deadlocks peuvent se produire quand plusieurs verrous sont utilisés.

Théorème

- Supposons qu'il y a un ordre sur les verrous $v_1 \leq v_2 \leq \cdots \leq v_n$.
- Supposons qu'aucun thread ne cherche à obtenir v alors qu'il possède déjà v' avec $v \leq v'$.
- Alors l'exécution ne comporte pas de deadlock.

TP 2

Utilisez le théorème ci-dessus pour résoudre le problème du dîner des philosophes

Situation de famine

Starvation

Situation où un ou plusieurs threads n'ont jamais accès à la ressource demandée.

TP 2

Observez les problèmes de famine dans le problème des toilettes unisexes.

Condition de vivacité

Progrès global

Au moins un thread progresse dans son exécution.

- propriété violée lors d'un interblocage circulaire.

Progrès local

Chaque thread progresse dans son exécution.

- propriété violée lors d'une famine.

Condition de vivacité

Progrès global

Au moins un thread progresse dans son exécution.

- propriété violée lors d'un interblocage circulaire.

Progrès local

Chaque thread progresse dans son exécution.

- propriété violée lors d'une famine.

Équité (fairness) des verrous

- Les blocks `synchronized`, les verrous `ReentrantLock` et les `mutex` ne garantissent que le progrès global.
- En Java, le constructeur de `ReentrantLock` a un argument optionnel :

```
public ReentrantLock(boolean fair)
```

Moniteur

Construction de synchronisation

- Un moniteur est un objet encapsulant des données partagées.
- Accès aux données partagées en exclusion mutuelle.
- Méthodes gardées par des conditions bloquantes (gardes).

Per Brinch Hansen

Tony Hoare

Per Brinch Hansen. *Operating System Principles*. Prentice Hall (1973)

H74 C. A. R. Hoare. *Monitors : an operating system structuring concept*. Communication of the ACM (1974)

File bloquante

Interface `java.util.concurrent.BlockingQueue`

	Exception	Valeur spéciale	Bloquante	Timeout
Insérer	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Supprimer	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Regarder	`element()`	`peek()`		

Un exemple d'utilisation

Problème des producteurs et des consommateurs

1. Étudiez le programme dans le dossier `ProducerConsumer/` sur Madoc.

Barrières de synchronisation

Classe `java.util.concurrent.CountDownLatch`

- `public void await() throws InterruptedException` :
  - Se met en attente jusqu'à ce que le compteur atteigne 0
- `public void countDown()` :
  - Décrémente le compteur

Classe `java.util.concurrent.CyclicBarrier`

- `public int await() throws InterruptedException, BrokenBarrierException` :
  - Se met en attente jusqu'à ce que tous les threads aient rejoint la barrière ;
  - Quand tous les threads ont rejoint la barrière, les threads sont réveillés et la barrière est réinitialisée.

Implémentation des moniteurs

```
public final void java.lang.Object.wait() throws InterruptedException
```

- Met le thread en attente sur l'objet verrouillé.
- Il doit posséder le verrou de l'objet sur lequel il souhaite être mis en attente.
- Lorsqu'il entre en attente, il relâche le verrou.

```
public final void java.lang.Object.notify()
```

- Réveille un des threads en attente sur l'objet verrouillé.
- Le thread réveillé doit acquérir le verrou pour poursuivre son exécution.

```
public final void java.lang.Object.notifyAll()
```

- Pareil mais réveille tous les threads en attente sur l'objet verrouillé.

Implémentation des moniteurs

`public final void java.lang.Object.wait() throws InterruptedException`

- Met le thread en attente sur l'objet verrouillé.
- Il doit posséder le verrou de l'objet sur lequel il souhaite être mis en attente.
- Lorsqu'il entre en attente, il relâche le verrou.

`public final void java.lang.Object.notify()`

- Réveille un des threads en attente sur l'objet verrouillé.
- Le thread réveillé doit acquérir le verrou pour poursuivre son exécution.

`public final void java.lang.Object.notifyAll()`

- Pareil mais réveille tous les threads en attente sur l'objet verrouillé.

Point vocabulaire

- Un thread demandant l'accès à un code déjà verrouillé est bloqué.
- Un thread qui est mis en pause par l'opération wait est en attente.

Syntaxes alternatives

En Java : java.util.concurrent.locks.Condition

```
Lock lock = new ReentrantLock();
Condition condition = lock.newCondition();

lock.lock();
condition.signal();
condition.await();
lock.unlock();
```

En C++ : std::condition_variable

```
mutex mtx;
condition_variable condition;

unique_lock<mutex> lock(mtx);
condition.notify_one();
condition.wait(lock);
// unlock on ~unique_lock
```

Bonne utilisation du wait

Imaginez 2 threads concurrents où l'un attend qu'un booléen `condition` soit mis à vrai par l'autre pour continuer son exécution.

```
public void guardedCondition() {

  while( !condition );
  System.out.println( "Condition true!" );

}
```

Bonne utilisation du wait

Imaginez 2 threads concurrents où l'un attend qu'un booléen `condition` soit mis à vrai par l'autre pour continuer son exécution.

```
public synchronized void guardedCondition()
        throws InterruptedException {

  if( !condition ) { wait(); }
  System.out.println( "Condition true!" );

}
```

Bonne utilisation du wait

Imaginez 2 threads concurrents où l'un attend qu'un booléen `condition` soit mis à vrai par l'autre pour continuer son exécution.

```
public synchronized void guardedCondition()
        throws InterruptedException {

  while( !condition ) { wait(); }
  System.out.println( "Condition true!" );

}
```

Bonne utilisation du wait

Imaginez 2 threads concurrents où l'un attend qu'un booléen `condition` soit mis à vrai par l'autre pour continuer son exécution.

```
public synchronized void guardedCondition()
        throws InterruptedException {

  while( !condition ) { wait(); }
  System.out.println( "Condition true!" );

}
```

TP 2

Résolvez le problème des toilettes unisexes comme un moniteur.

Modèles de mémoire

- Le mot-clé `Volatile`
- Linéarisabilité
- Cohérence causale
- Instructions spéciales

Modèles de mémoire

TP 1

1. Récupérer et étudier le programme `Volatile.java`.
2. Exécuter le programme.
3. Ajouter le mot-clé `volatile`.
4. Exécuter le programme.

Modèles de mémoire

TP 1

1. Récupérer et étudier le programme `Volatile.java`.
2. Exécuter le programme.
3. Ajouter le mot-clé `volatile`.
4. Exécuter le programme.

Problèmes

- À quel niveau se passe l'atomicité ?
- Que signifie lire et écrire dans une variable partagée ?
- Comment spécifier un objet partagé ?

Cohérence de cache

Quels affichages sont possibles ? (initialement, `condition = false;`)

Thread 1

```
// Do something
condition = true;
```

Thread 2

```
while(!condition);
system.out.println("Done");
```

Cohérence de cache

Quels affichages sont possibles ? (initialement, `condition = false;`)

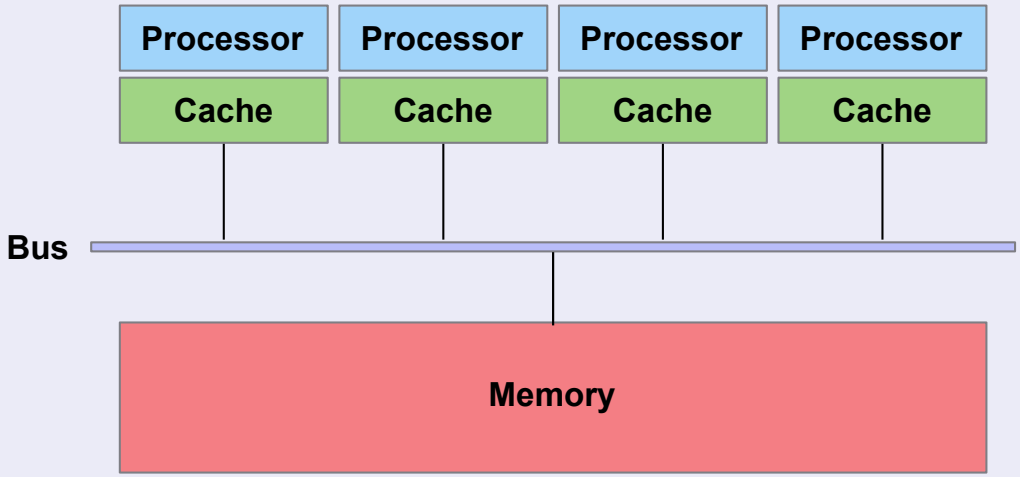
Thread 1

```
// Do something
condition = true;
```

Thread 2

```
while(!condition);
system.out.println("Done");
```

Architectures UMA (Uniform Memory Access)



Barrières de mémoire

C++

- `void std::atomic_thread_fence( std::memory_order order )`
- Choix de synchronisation read/write, write/read...

Barrières de mémoire

C++

- `void std::atomic_thread_fence( std::memory_order order )`
- Choix de synchronisation read/write, write/read...

`volatile int x;` (depuis Java 5)

- Tout le cache est invalidé avant chaque lecture de x
- Tout le cache est invalidé après chaque écriture de x
- Le cache n'est pas invalidé pendant une lecture ou écriture de x

Attention

- Une invalidation de cache est très inefficace !

Barrières de mémoire

C++

- `void std::atomic_thread_fence( std::memory_order order )`
- Choix de synchronisation read/write, write/read...

`volatile int x;` (depuis Java 5)

- Tout le cache est invalidé avant chaque lecture de x
- Tout le cache est invalidé après chaque écriture de x
- Le cache n'est pas invalidé pendant une lecture ou écriture de x

Attention

- Une invalidation de cache est très inefficace !

Lien avec les verrous

Java : cache invalidé à l'acquisition et au relâchement d'un verrou

C++ : on choisit l'option (voir `memory_order`)

Optimisations du code

Quels affichages sont possibles ?

Initialement

```
int sum = 0;
int i = 0;
```

Thread 1

```
for(i=1; i<2; i++)
  sum=sum+i;
```

Thread 2

```
if(sum != 0)
  system.out.println(i);
```

Optimisations du code

Quels affichages sont possibles ?

Initialement

```
int sum = 0;
int i = 0;
```

Thread 1

```
for(i=1; i<2; i++)
  sum=sum+i;
```

Thread 2

```
if(sum != 0)
  system.out.println(i);
```

Le compilateur et l'interpréteur respectent la sémantique séquentielle

```
for(i=1;i<2;i++){
  sum=sum+i;
}
```

Optimisations du code

Quels affichages sont possibles ?

Initialement

```
int sum = 0;
int i = 0;
```

Thread 1

```
for(i=1; i<2; i++)
  sum=sum+i;
```

Thread 2

```
if(sum != 0)
  system.out.println(i);
```

Le compilateur et l'interpréteur respectent la sémantique séquentielle

```
for(i=1;i<2;i++){
  sum=sum+i;
}
```

```
i=1;
sum=1;
i=2;
```

Optimisations du code

Quels affichages sont possibles ?

Initialement

```
int sum = 0;
int i = 0;
```

Thread 1

```
for(i=1; i<2; i++)
  sum=sum+i;
```

Thread 2

```
if(sum != 0)
  system.out.println(i);
```

Le compilateur et l'interpréteur respectent la sémantique séquentielle

```
for(i=1;i<2;i++){
  sum=sum+i;
}
```

```
i=1;
sum=1;
i=2;
```

```
sum=1;
i=2;
```

Optimisations du code

Quels affichages sont possibles ?

Initialement

```
int sum = 0;
int i = 0;
```

Thread 1

```
for(i=1; i<2; i++)
  sum=sum+i;
```

Thread 2

```
if(sum != 0)
  system.out.println(i);
```

Le compilateur et l'interpréteur respectent la sémantique séquentielle

```
for(i=1;i<2;i++){
  sum=sum+i;
}
```

```
i=1;
sum=1;
i=2;
```

```
sum=1;
i=2;
```

```
volatile int x;
```

- Pas d'optimisation qui change la position d'une opération sur x
- Transmet la causalité sur x

Histoire concurrente

Problème

- Que signifie lire et écrire dans une variable partagée ?

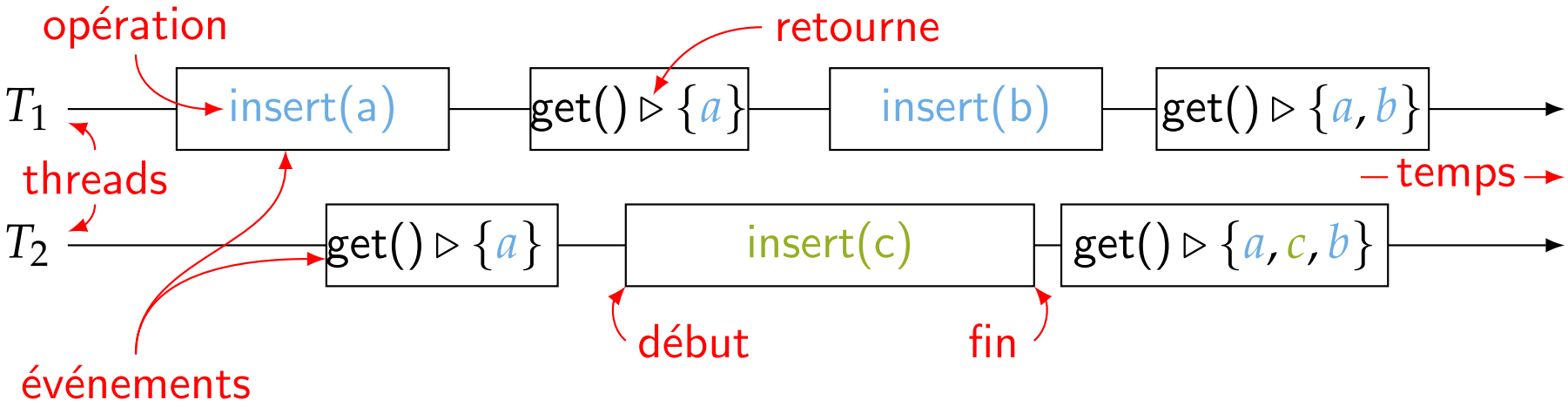
Histoire concurrente

Problème

- Que signifie lire et écrire dans une variable partagée ?

Spécifier un objet partagé

- Définir quelles histoires concurrentes sont admissibles pour un objet donné



Spécification d'un objet partagé

Spécification séquentielle : comportement fonctionnel séquentiel

- Dans ce cours (non standard) : remplacement de `class` par `specification`

```
specification RWRegister<T> {
  T value;
  T get()            { return value; }
  void set(T x)      { value = x;    }
}
```

Spécification d'un objet partagé

Spécification séquentielle : comportement fonctionnel séquentiel

- Dans ce cours (non standard) : remplacement de `class` par `specification`

```
specification RWRegister<T> {
  T value;
  T get()            { return value; }
  void set(T x)      { value = x;    }
}
```

Critère de cohérence : effet de la concurrence sur le comportement

- Linéarisabilité, cohérence causale, cohérence locale...

Spécification d'un objet partagé

Spécification séquentielle : comportement fonctionnel séquentiel

- Dans ce cours (non standard) : remplacement de `class` par `specification`

```
specification RWRegister<T> {
  T value;
  T get()            { return value; }
  void set(T x)      { value = x;    }
}
```

Critère de cohérence : effet de la concurrence sur le comportement

- Linéarisabilité, cohérence causale, cohérence locale...

Condition de progression : propriété de vivacité

- Deadlock-freedom, starvation-freedom, lock-freedom, wait-freedom...

Cohérence locale

Cohérence locale

- Pour chaque thread t, la séquence des opérations de t respecte la spécification séquentielle.

Cohérence locale

Cohérence locale

- Pour chaque thread t, la séquence des opérations de t respecte la spécification séquentielle.

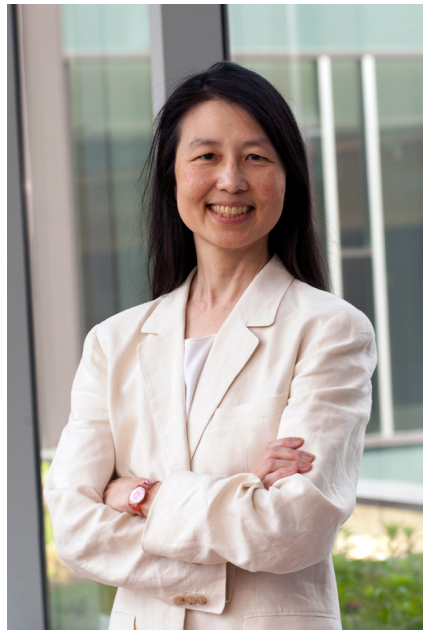
Utilisation de `ThreadLocal`

- En Java, utiliser la classe `ThreadLocal`
- En C++, utiliser le mot-clé `thread_local`
- On peut aussi déclarer une variable membre dans `Thread` / `Runnable`

```
class IdAssignor {
  ThreadLocal<Integer> id = ThreadLocal.withInitial(() -> -1);
  int next = 0;

  synchronized int getId() {
    if (id.get() == -1) id.set(next++);
    return id.get();
  }
}
```

Linéarisabilité

Maurice P. Herlihy



Jeanette M. Wing

L86 Leslie Lamport. *On interprocess communication.* Distributed Computing (1986)

HW90 Maurice P. Herlihy, Jeanette M. Wing. *Linearizability : A Correctness Condition for Concurrent Objects.* ToPLaS (1990)

Linéarisabilité

Définition par points de linéarisation

Tout se passe comme si chaque événement avait lieu en un point atomique situé entre le début et la fin de cet événement.

Définition par ordre total

Il existe un ordre total $<$ sur tous les événements tel que :

- l'exécution des opérations dans l'ordre $<$ respecte la spécification séquentielle
- si e se termine avant que e' ne commence, alors $e < e'$

Linéarisabilité

Définition par points de linéarisation

Tout se passe comme si chaque événement avait lieu en un point atomique situé entre le début et la fin de cet événement.

Définition par ordre total

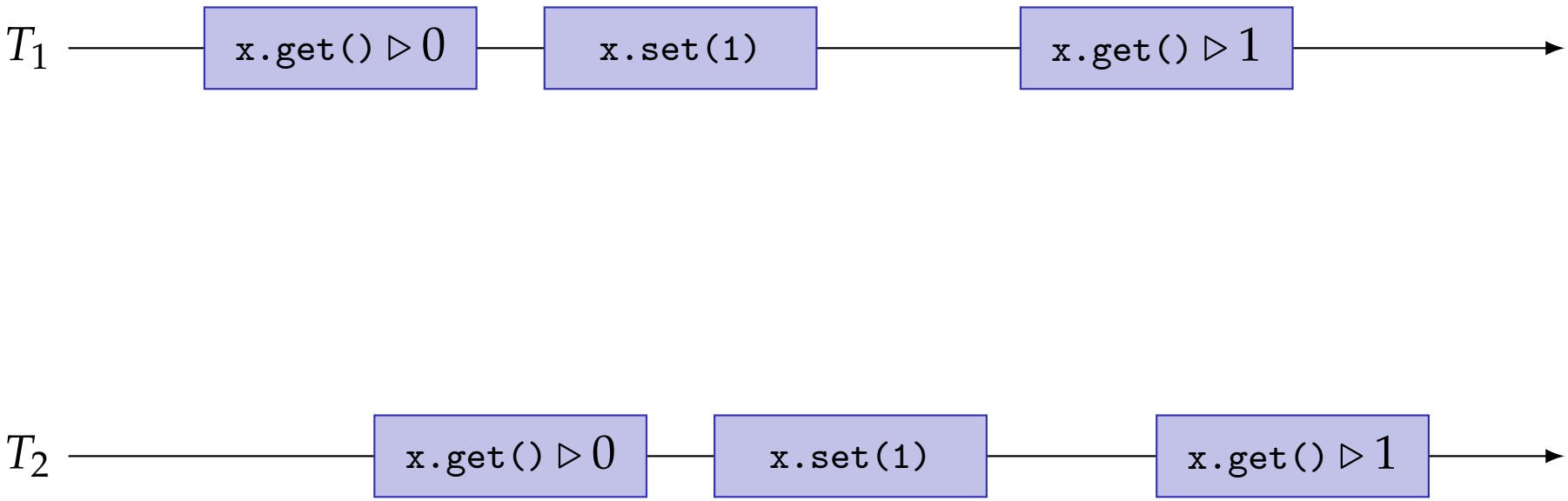
Il existe un ordre total $<$ sur tous les événements tel que :

- l'exécution des opérations dans l'ordre $<$ respecte la spécification séquentielle
- si e se termine avant que e' ne commence, alors $e < e'$

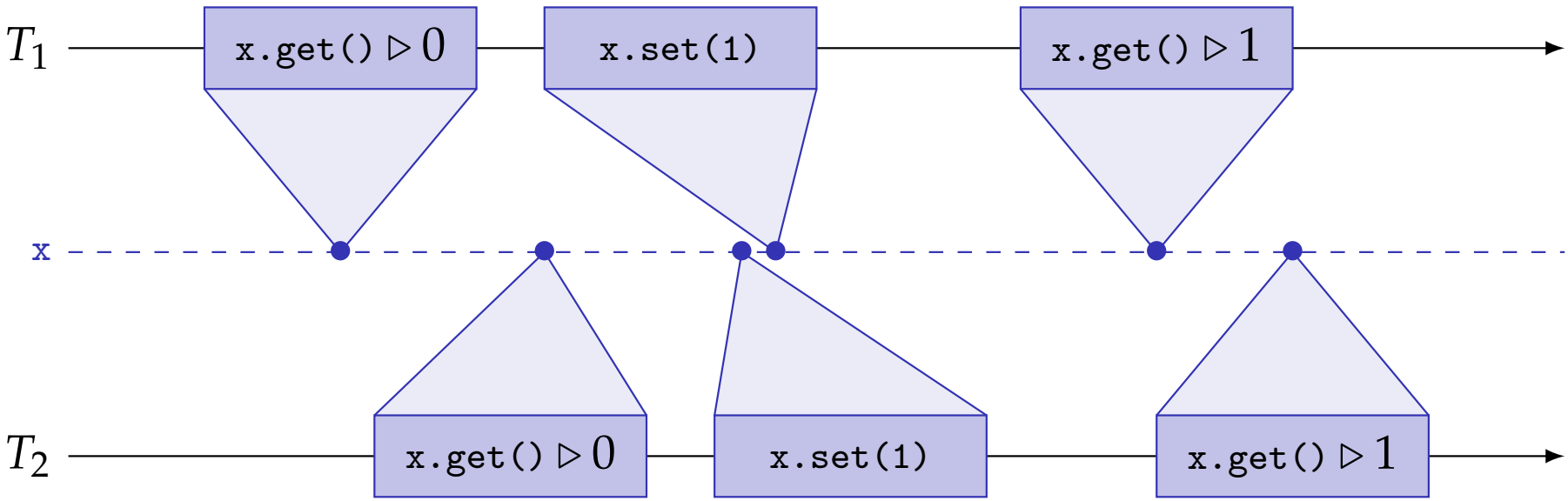
Observational refinement

- Pas de différence observable entre linéarisabilité et atomicité !
- Si toutes les variables partagées sont `volatile`,
  pas de calcul pour l'ordonnanceur = opération sur une variable

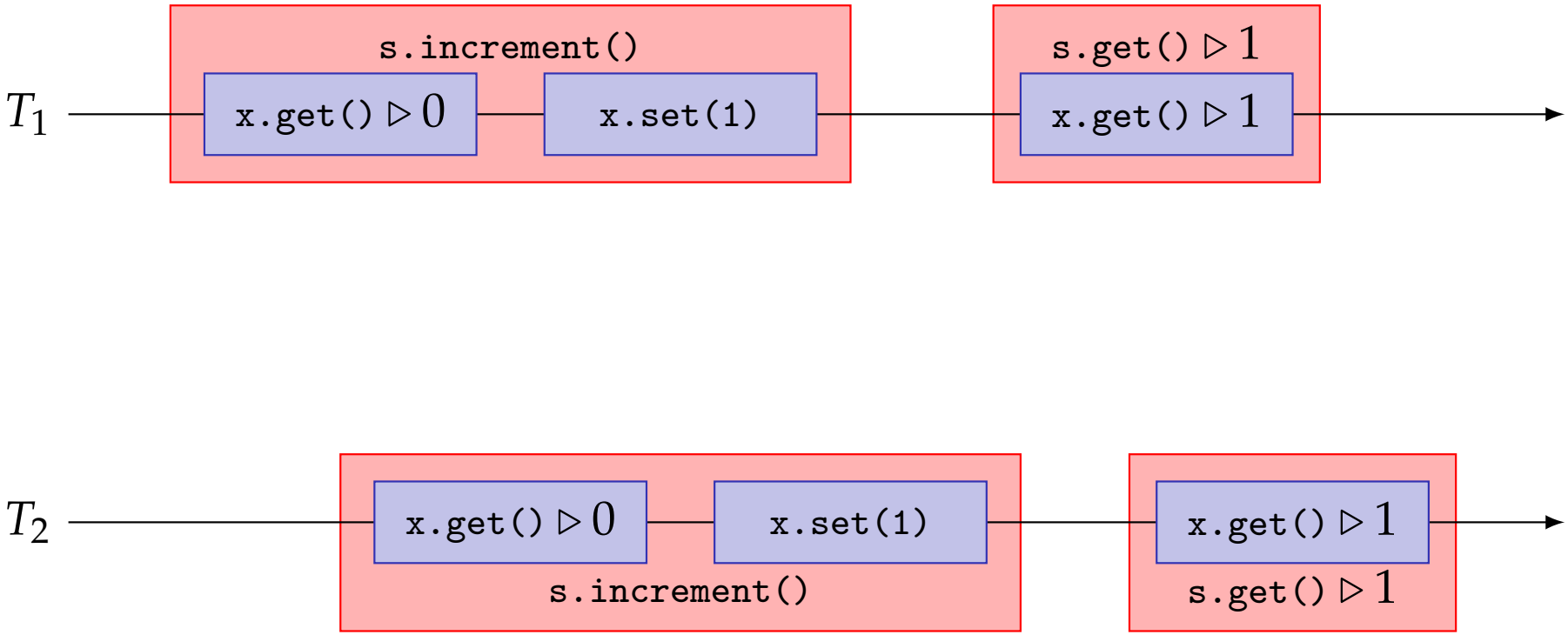
Exemple : compteur partagé



Exemple : compteur partagé



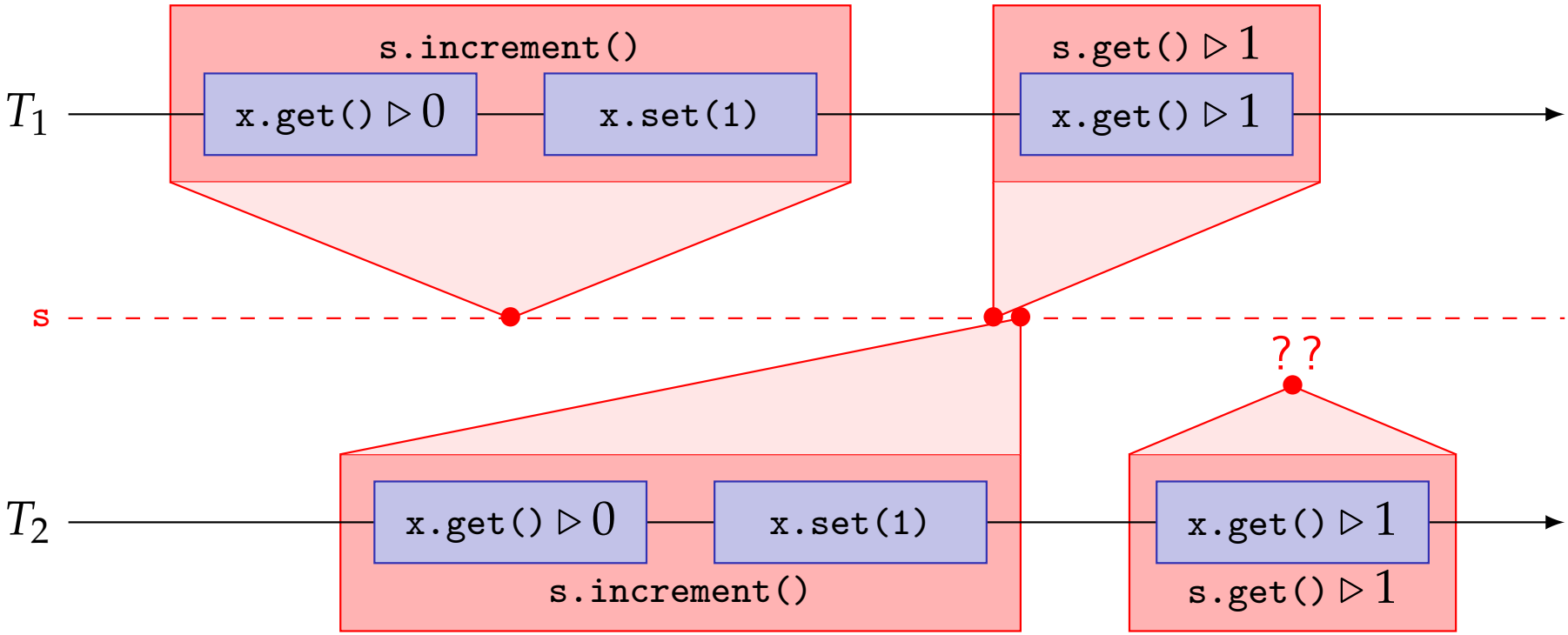
Exemple : compteur partagé




```
class Shared {
  int x = 0;
  void increment() {x = x+1;}
  int get() {return x;}
}
```

```
Shared s = new Shared();
// Thread T1 :
s.increment(); a = s.get();
// Thread T2 :
s.increment(); b = s.get();
```

Exemple : compteur partagé




```
class Shared {
  int x = 0;
  void increment() {x = x+1;}
  int get() {return x;}
}
```

```
Shared s = new Shared();
// Thread T1 :
s.increment(); a = s.get();
// Thread T2 :
s.increment(); b = s.get();
```

Cohérence causale

Mustaque Ahamad

ANBKH98 Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, Phillip W. Hutto. *Causal memory : Definitions, implementation, and programming.* ICDCS. (1998)

PMJ16 Matthieu Perrin, Achour Mostéfaoui, Claude Jard. *Causal consistency : Beyond memory.* PPoPP. (2016)

Mémoire causale

La relation "happens-before" (adaptée à Java)

$e \overset{HB_v}{\longmapsto} e'$ si l'une des trois s'applique :

- Ordre de programme :
  - e ou e' est un accès à une variable `volatile` ou un bloc `synchronized` ;
  - ET e et e' produites par le même thread dans cet ordre
- Précédence lire/écrire : $e = x.set(a)$ et $e' = x.get() \triangleright a$;
- Transitivité : $\exists e'', e \overset{HB_v}{\longmapsto} e'' \overset{HB_v}{\longmapsto} e'$.

L78 Leslie Lamport. *Time, clocks, and the ordering of events in a distributed system.* CACM (1978)

Mémoire causale

La relation "happens-before" (adaptée à Java)

$e \overset{HB_v}{\longmapsto} e'$ si l'une des trois s'applique :

- Ordre de programme :
  - e ou e' est un accès à une variable `volatile` ou un bloc `synchronized` ;
  - ET e et e' produites par le même thread dans cet ordre
- Précédence lire/écrire : $e = x.set(a)$ et $e' = x.get() \triangleright a$;
- Transitivité : $\exists e'', e \overset{HB_v}{\longmapsto} e'' \overset{HB_v}{\longmapsto} e'$.

Cohérence causale (faible)

Il est impossible que :

$$x.set(a) \overset{HB_v}{\longmapsto} x.set(b) \overset{HB_v}{\longmapsto} x.get() \triangleright a$$

L78 Leslie Lamport. *Time, clocks, and the ordering of events in a distributed system.* CACM (1978)

Exemple (schématique)

```
class Future<T> {


  final Callable<T> task;
  volatile boolean isDone = false;
  T value = null;


  void init() { // par thread du thread pool
    value = task.call();
    isDone = true;
  }


  T get() { // par autre thread
    while(!isDone);
    return value;
  }


}
```

Exemple (schématique)

```
class Future<T> {

  final Callable<T> task;
  volatile boolean isDone = false;
  T value = null;

  void init() { // par thread du thread pool
    value = task.call();
    isDone = true;
  }

  T get() { // par autre thread
    while(!isDone);
    return value;
  }

}
```

Exemple (schématique)

```
class Future<T> {

  final Callable<T> task;
  volatile boolean isDone = false;
  T value = null;

  void init() { // par thread du thread pool
    value = task.call();
    isDone = true;
  }

  T get() { // par autre thread
    while(!isDone);
    return value;
  }

}
```

Ordre de programme

Ordre de programme

Exemple (schématique)

```
class Future<T> {

  final Callable<T> task;
  volatile boolean isDone = false;
  T value = null;

  void init() { // par thread du thread pool
    value = task.call();
    isDone = true;
  }

  T get() { // par autre thread
    while(!isDone);
    return value;
  }

}
```

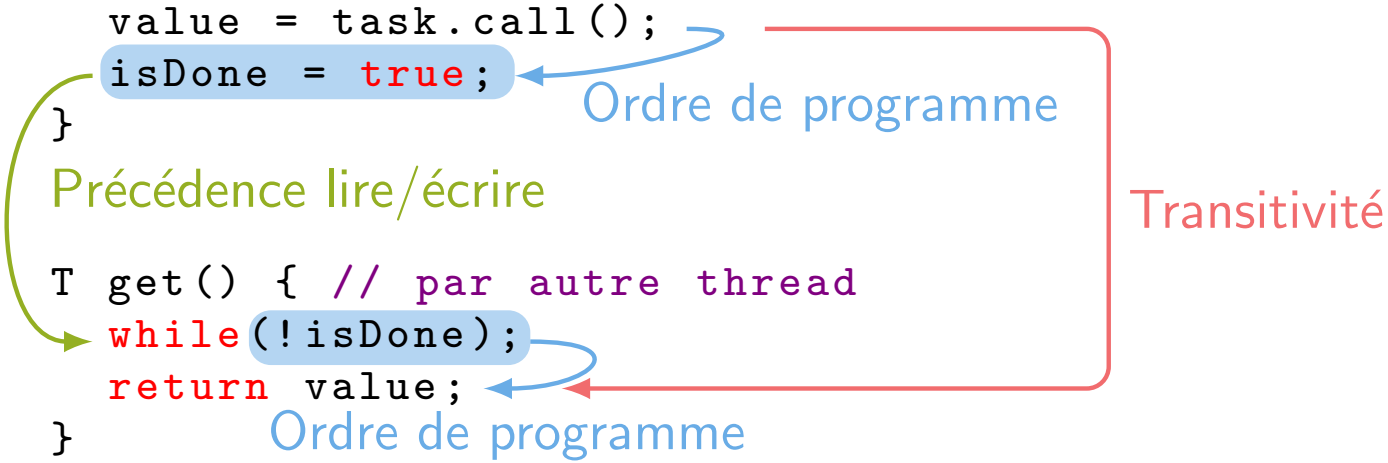
Ordre de programme

Précédence lire/écrire

Ordre de programme

Exemple (schématique)

```
class Future<T> {

  final Callable<T> task;
  volatile boolean isDone = false;
  T value = null;

  void init() { // par thread du thread pool
    value = task.call();
    isDone = true;
  }

  T get() { // par autre thread
    while(!isDone);
    return value;
  }

}
```



Modèle de mémoire en Java

Le mot-clé `volatile` en Java

- Garanties apportées
  - Les variables `volatile` sont linéarisables
  - Les variables `volatile` transmettent la causalité
- Gestion par le système
  - Limite l'utilisation des caches
  - Limite les optimisations du compilateur

Toute variable partagée doit être :

- soit déclarée `final`
- soit déclarée `volatile`
- soit uniquement accédée dans des blocs `synchronized`
- tout autre cas doit être parfaitement commenté et justifié

Limites de `volatile` : les tableaux

Tableaux volatiles

```
volatile int array[];
array = new int[10];              // volatile
array[5] = 3;                     // pas volatile
```

Limites de `volatile` : les tableaux

Tableaux volatiles

```
volatile int array[];
array = new int[10];              // volatile
array[5] = 3;                     // pas volatile
```

Première solution

```
class VolatileInt{public volatile int x=0;}
VolatileInt array[] = new VolatileInt[10];
array[5] = new VolatileInt(); // pas volatile
array[5].x = 3;               // volatile
```

Limites de `volatile` : les tableaux

Tableaux volatiles

```
volatile int array[];
array = new int[10];              // volatile
array[5] = 3;                     // pas volatile
```

Première solution

```
class VolatileInt{public volatile int x=0;}
VolatileInt array[] = new VolatileInt[10];
array[5] = new VolatileInt(); // pas volatile
array[5].x = 3;               // volatile
```

Meilleure solution : `java.util.concurrent.atomic.AtomicIntegerArray`

```
AtomicIntegerArray array = new AtomicIntegerArray(10);
array.set(5, 3);                  // volatile
```

Limites de `volatile` : instructions complexes

Exemple : incrémentation

```
volatile int counter;
counter++;                    // pas atomique
```

Limites de `volatile` : instructions complexes

Exemple : incrémentation

```
volatile int counter;
counter++;                    // pas atomique
```

Première solution : utiliser des verrous

```
Integer counter;
synchronized(counter){
  counter++;                  // section critique
}
```

Limites de `volatile` : instructions complexes

Exemple : incrémentation

```
volatile int counter;
counter++;                    // pas atomique
```

Première solution : utiliser des verrous

```
Integer counter;
synchronized(counter){
  counter++;                  // section critique
}
```

Meilleure solution : `java.util.concurrent.atomic.AtomicInteger`

```
AtomicInteger counter;
counter.getAndIncrement(); // linearisable
```

Registres atomiques

Java

- Dans le package `java.util.concurrent.atomic`
  - `AtomicInteger`, `AtomicLong`, `AtomicReference<T>`...

C++

- Structure `template <class T> struct std::atomic`
- Fonctions de `<atomic>`

Méthodes `get()` et `set()` (Java) ou `load()` et `store()` (C++)

Lisent et écrivent dans une variable `volatile` en interne

Méthodes RMW (read-modify-write)

- Opérations complexes linéarisables
- Certaines sont des primitives du processeur (opérations spéciales)
- Nécessaire (et suffisantes) pour implémenter des objets non bloquants.

L'instruction test_and_set

Spécification

```
T old = get();
set(val);
return old;
```

Java : `T getAndSet(T val)`

C++ : `T exchange(T val)`

L'instruction test_and_set

Spécification

```
T old = get();
set(val);
return old;
```

Java : `T getAndSet(T val)`

C++ : `T exchange(T val)`

Exemple : le spin lock

```
class SpinLock {
  private AtomicBoolean taken = new AtomicBoolean(false);

  public void lock() {
    while(taken.getAndSet(true));
  }

  public void unlock() {
    taken.set(false);
  }
}
```

D62 Edsger W. Dijkstra. *Over de sequentialiteit van procesbeschrijvingen (About the sequentiality of process descriptions)* (1962-1963)

L'instruction fetch_and_add

Spécification

```
int old = get();
set(old + delta);
return old;
```

Java : `int getAndAdd(int delta)`

C++ : `int fetch_add(int delta)`

L'instruction fetch_and_add

Spécification

```
int old = get();
set(old + delta);
return old;
```

Java : `int getAndAdd(int delta)`

C++ : `int fetch_add(int delta)`

Exemple : un autre verrou

```
class FairLock() {
  volatile int suivant = 0;
  AtomicInteger distributeur = new AtomicInteger(0);

  void lock() {
    int ticket = distributeur.getAndAdd(1);
    while(ticket != suivant);
  }

  void unlock() {
    suivant++;
  }
}
```

L'instruction compare_and_swap

Spécification

```
if (get() == expect) {
  set(update);
  return true;
}
return false;
```

Java :

```
boolean compareAndSet(T expect, T update)
```

C++ :

```
bool compare_exchange_strong (T& expected, T val)
```

L'instruction compare_and_swap

Spécification

```
if (get() == expect) {
  set(update);
  return true;
}
return false;
```

Java :

```
boolean compareAndSet(T expect, T update)
```

C++ :

```
bool compare_exchange_strong (T& expected, T val)
```

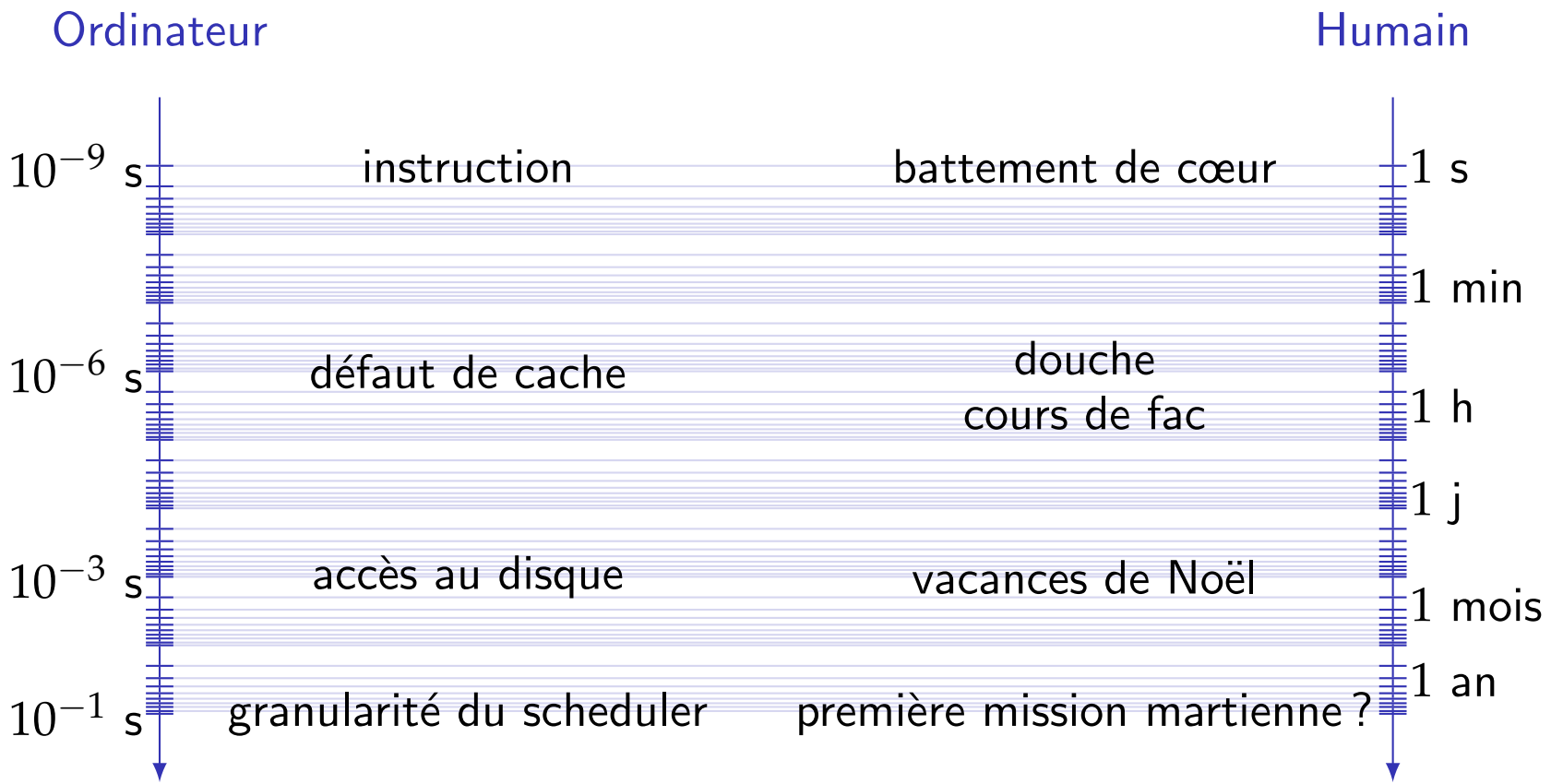
Exemple : implémentation de `AtomicInteger.getAndIncrement` dans l'OpenJDK

```
public final int getAndIncrement() {

  for (;;) {
    int current = get();
    int next = current + 1;

    if (compareAndSet(current, next))
      return current;
  }
}
```

Algorithmes non-bloquants

- Vivacité non-bloquante
- La file lock-free

Échelles de temps



Système asynchrone

- Un système asynchrone n'est pas régi par une horloge commune.
- Quid d'un thread qui attend un verrou détenu par un thread non schedulé ?

Problèmes avec les verrous

Notion de panne franche (crash)

Un thead en panne n'execute plus jamais de pas

- En mémoire partagée, approximation d'un thread infiniment lent

Problèmes

- Si un thread crashe alors qu'il possède un verrou, aucun autre ne peut l'avoir
- Si un thread est bloqué, on ne peut pas savoir combien de temps il va l'être

Bloquant/Non-bloquant

- Un algorithme est non-bloquant s'il peut tolérer les pannes franches.
- Tous les algorithmes utilisant des verrous sont bloquants

Conditions de progression

Progrès global : les rapides peuvent passer devant les lents en permanence

Progrès local : équité sur la terminaison

Progrès bloquant : les threads lents peuvent gêner les rapides

Progrès non-bloquant : terminaison malgré les pannes

Conditions de progression

Progrès global : les rapides peuvent passer devant les lents en permanence

Progrès local : équité sur la terminaison

Progrès bloquant : les threads lents peuvent gêner les rapides

Progrès non-bloquant : terminaison malgré les pannes

	Progrès global	Progrès local
Progrès bloquant	Deadlock-free	Starvation-free
Progrès non-bloquant	Lock-free	Wait-free

Définitions

Toutes les opérations (des non-crashés) se terminent sous l'hypothèse :

Deadlock-freedom : il n'y a pas de crash et il y a un nombre fini d'opérations

Starvation-freedom : il y a un nombre fini d'opérations

Lock-freedom : il n'y a pas de crash

Wait-freedom : vrai

HS11 Maurice Herlihy, Nir Shavit. *On the nature of progress.* PODS (2011)

Structures de données lock-free en Java

File `java.util.concurrent.ConcurrentLinkedQueue<E>` :

- `void add(E e)` : ajoute en queue
- `E peek()` : lit la tête
- `E poll()` : lit et supprime la tête

Structures de données lock-free en Java

File `java.util.concurrent.ConcurrentLinkedQueue<E>` :

- `void add(E e)` : ajoute en queue
- `E peek()` : lit la tête
- `E poll()` : lit et supprime la tête

Liste doublement chaînée `java.util.concurrent.ConcurrentLinkedDeque<E>`

- `void addFirst(E e)`, `E getFirst()`, `E pollFirst()` : accède la tête
- `void addLast(E e)`, `E getLast()`, `E pollLast()` : accède la queue

Structures de données lock-free en Java

File `java.util.concurrent.ConcurrentLinkedQueue<E>` :

- `void add(E e)` : ajoute en queue
- `E peek()` : lit la tête
- `E poll()` : lit et supprime la tête

Liste doublement chaînée `java.util.concurrent.ConcurrentLinkedDeque<E>`

- `void addFirst(E e)`, `E getFirst()`, `E pollFirst()` : accède la tête
- `void addLast(E e)`, `E getLast()`, `E pollLast()` : accède la queue

Table de hashage `java.util.concurrent.ConcurrentHashMap<K,V>`

- `put(K key, V value)` : ajoute
- `get(K key)` : lit
- `remove(K key)` : supprime une clé
- `putIfAbsent(K key, V value)` : ajoute ou retourne la valeur

Structures de données lock-free en Java

File `java.util.concurrent.ConcurrentLinkedQueue<E>` :

- `void add(E e)` : ajoute en queue
- `E peek()` : lit la tête
- `E poll()` : lit et supprime la tête

Liste doublement chaînée `java.util.concurrent.ConcurrentLinkedDeque<E>`

- `void addFirst(E e)`, `E getFirst()`, `E pollFirst()` : accède la tête
- `void addLast(E e)`, `E getLast()`, `E pollLast()` : accède la queue

Table de hashage `java.util.concurrent.ConcurrentHashMap<K,V>`

- `put(K key, V value)` : ajoute
- `get(K key)` : lit
- `remove(K key)` : supprime une clé
- `putIfAbsent(K key, V value)` : ajoute ou retourne la valeur

Compteur `java.util.concurrent.atomic.LongAdder`

- `void add(long x)` : incrémente
- `long sum()` : lit

La file lock-free `java.util.concurrent.ConcurrentLinkedQueue`

Maged M. Michael

Michael L. Scott

Implémentations

Java `java.util.concurrent.ConcurrentLinkedQueue<E>`,

C++ https://github.com/schani/michael-alloc/blob/master/lock-free-queue.c,

Madoc Récupérez le programme `MichaelScottQueue.java`.

MS96 Maged M. Michael, Michael L. Scott. *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.* PODC (1996)

Problème : la file linéarisable lock-free

Spécification séquentielle

```
specification Queue<T> {
  private List<T> list = new LinkedList<E>();

  public void enqueue (T value) {
    list.addLast(value);
  }

  public T dequeue () {
    if (list.length() == 0) return null ;
    T value = list.getFirst();
    list.removeFirst();
    return value;
  }
}
```

Critère de cohérence

Linéarisabilité

Condition de progression

Lock-freedom

Calculabilité en mémoire partagée

- Constructions universelles
- Impossibilité du consensus
- Hiérarchie de Herlihy

Synchronisation wait-free

Maurice P. Herlihy

Pourquoi a-t-on ajouté le compareAndSet aux processeurs ?

- Read/Write n'est pas universel
- CompareAndSet est universel

H88 Maurice P. Herlihy. *Wait-Free Synchronization*. PODC (1988)

Exemple d'algorithme avec compareAndSet

Compteur lock-free (Voir Slide 57)

```
class Compteur {

  AtomicInteger value = new AtomicInteger(0);

  int getNext() {
    int current;
    do current = value.get();
    while (!value.compareAndSet(current, current+1));
    return current;
  }
}
```

Généralisation

Une spécification séquentielle peut être décrite par un automate :

- `State` : ensemble d'états
- `initialState` : état initial
- `Operation` : ensemble d'opérations (méthodes + arguments)
- `State Effect(State, Operation)` : fonction qui décrit les effets de bord
- `R Result(State, Operation)` : fonction qui décrit les retours

```
class UniversalConstruction {

  AtomicReference<State> s = new AtomicReference(initialState);

  R execute(Operation o) {
    State current;
    do current = s.get();
    while (!s.compareAndSet(current, Effect(current, o)));
    return Result(current, o);
  }
}
```

Universalité de compareAndSet

Notion d'universalité

Soit P une condition de progression.

Construction universelle Algorithme générique linéarisable paramétré par une spécification séquentielle.

Objet universel X est P-universel si on peut implémenter n'importe quel objet respectant P et la linéarisabilité, en n'utilisant que des registres et des instances de X.

Exemples

- CompareAndSet est lock-free universel
- Exemples de constructions universelles sur Madoc

Universalité de compareAndSet

Notion d'universalité

Soit P une condition de progression.

Construction universelle Algorithme générique linéarisable paramétré par une spécification séquentielle.

Objet universel X est P-universel si on peut implémenter n'importe quel objet respectant P et la linéarisabilité, en n'utilisant que des registres et des instances de X.

Exemples

- CompareAndSet est lock-free universel
- Exemples de constructions universelles sur Madoc

Et les registres lire/écrire ?

Universalité bloquante

Verrouillage à gros grains

```
class BlockingUniversal {
  State s = initialState;
  synchronized R execute(Operation o) {
    State current = s;
    s = Effect(current, o)));
    return Result(current, o);
  }
```

- Les verrous deadlock-free sont deadlock-free-universels
- Les verrous starvation-free sont starvation-free-universels

Théorème

- Lire/écrire est starvation-free universel (admis).
- Lire/écrire n'est pas lock-free universel (à démontrer).

D62 Edsger W. Dijkstra. *Over de sequentialiteit van procesbeschrijvingen* (1962-1963)

L74 Leslie Lamport, *A New Solution of Dijkstra's Concurrent Programming Problem*, Communications of the ACM (1974)

Rappel : montrer qu'un problème est indécidable

Preuve par réduction

Supposons qu'on a déjà démontré que P_0 est impossible.
Peut-on résoudre le problème P ?

1. Supposons que l'on peut résoudre P
2. On prouve qu'alors on peut utiliser la solution pour résoudre P_0
3. Or on sait qu'on ne peut pas résoudre P_0 (voir ci-dessus)
4. Donc l'hypothèse est fausse (on ne peut pas résoudre P)

Rappel : montrer qu'un problème est indécidable

Preuve par réduction

Supposons qu'on a déjà démontré que P_0 est impossible.
Peut-on résoudre le problème P ?

1. Supposons que l'on peut résoudre P
2. On prouve qu'alors on peut utiliser la solution pour résoudre P_0
3. Or on sait qu'on ne peut pas résoudre P_0 (voir ci-dessus)
4. Donc l'hypothèse est fausse (on ne peut pas résoudre P)

Preuve directe

Il faut toujours montrer qu'un premier problème P_0 est indécidable

Géométrie plane : la quadrature du cercle

Calcul séquentiel : l'arrêt de la machine de Turing

Calcul réparti : le consensus

Le consensus

Définition fonctionelle

Chaque thread propose une valeur et peut en décider une.

- **Accord :** Au plus une valeur décidée
- **Validité :** Toute valeur décidée a été proposée
- **Wait-free :** Tous les threads corrects (pas en panne) se terminent

Le consensus

Définition fonctionelle

Chaque thread propose une valeur et peut en décider une.

- **Accord :** Au plus une valeur décidée
- **Validité :** Toute valeur décidée a été proposée
- **Wait-free :** Tous les threads corrects (pas en panne) se terminent

Définition par un objet linéarisable et wait-free

```
specification Consensus<T> {
  private T value = null;
  public T propose(T val) {
    if(value == null) value = val;
    return value;
  }
}
```

Le consensus

Définition fonctionelle

Chaque thread propose une valeur et peut en décider une.

- **Accord :** Au plus une valeur décidée
- **Validité :** Toute valeur décidée a été proposée
- **Wait-free :** Tous les threads corrects (pas en panne) se terminent

Définition par un objet linéarisable et wait-free

```
specification Consensus<T> {
  private T value = null;
  public T propose(T val) {
    if(value == null) value = val;
    return value;
  }
}
```

Le consensus est wait-free universel

- On doit se mettre d'accord sur un journal des opérations exécutées

Impossibilité du consensus

Michael J. Fischer

Nancy A. Lynch

Michael S. Paterson

FLP85 Michael J. Fischer, Nancy A. Lynch, Michael S. Paterson. *Impossibility of Distributed Consensus with One Faulty Process.* JACM (1985)

H88 Maurice P. Herlihy. *Wait-Free Synchronization.* PODC (1988)

`RWRegister` versus consensus

Théorème

Il n'existe pas d'implémentation wait-free du consensus qui n'utilise que des instances linéarisables wait-free de `RWRegister`.

Plan de la démonstration

- **Illustration** `getAndSet` et arbre des possibles ;
- **Définitions** états v-valents, monovalents, bivalents et critiques ;
- **Introduction** supposons, par l'absurde, qu'il existe un algorithme A ;
  - **Lemme 1** il existe un état critique ;
  - **Lemme 2** il n'existe pas d'état critique ;
- **Conclusion** absurde : A n'existe pas !

Calculabilité de `getAndSet`

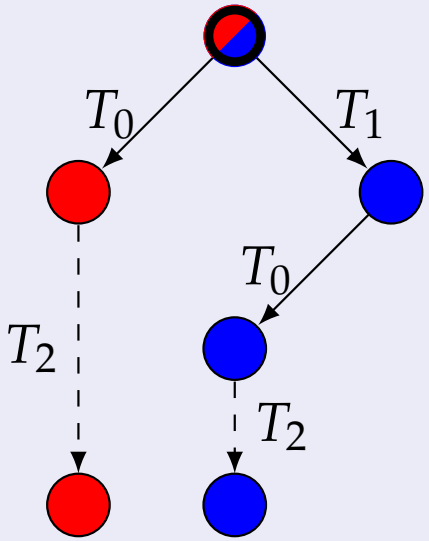
Théorème

- `getAndSet` résout le consensus entre 2 threads
- `getAndSet` ne résout pas le consensus entre ≥ 3 threads

Démonstration

Supposons qu'il existe un protocole pour 3 threads. Dans un état critique :

- si l'un d'eux lit, écrit ou accède à un registre différent : comme avant
- si les trois appellent `getAndSet`



Consensus number

Consensus number d'une structure de données T

Le consensus number de T vaut :

- si ça existe, le nombre n tel que :
  - T est lock-free-universel quand il y a au plus n threads
  - T n'est pas lock-free-universel quand il y a au moins $n+1$ threads
- sinon, ∞ ($\simeq$ universel)

Consensus number d'une opération spéciale

Le consensus number d'une opération spéciale est celui d'un registre avec une opération de lecture, une opération d'écriture, et cette opération

Exemples

- `RWRegister`, Increment, Insert/Delete : $CN = 1$
- `getAndSet`, `getAndAdd`, Pile, File : $CN \geq 2$
- `compareAndSet` Consensus : $CN = \infty$

Raisonnement avec le consensus number

Hiérarchie des consensus

Consensus à 1 thread (trivial) $<$ Consensus à 2 threads $< \cdots <$ Consensus

Théorème

Soit deux structures de données S et T de consensus number respectifs $x < y$. Il est impossible d'implémenter T en utilisant seulement des instances de S et des `RWRegister`.

Attention : la réciproque est fausse !

Il existe deux structures de données S et T de consensus number respectifs $x \geq y$ telles qu'il est impossible d'implémenter T en utilisant seulement des instances de S et des `RWRegister`.

Conclusion : pourquoi la classe `AtomicInteger` ?

`int`	2	**`addAndGet`**`(int delta)` Atomically adds the given value to the current value.
`boolean`	∞	**`compareAndSet`**`(int expect, int update)` Atomically sets the value to the given updated value if the current value == the expected value.
`int`	2	**`decrementAndGet`**`()` Atomically decrements by one the current value.
`double`	1	**`doubleValue`**`()` Returns the value of the specified number as a `double`.
`float`	1	**`floatValue`**`()` Returns the value of the specified number as a `float`.
`int`	1	**`get`**`()` Gets the current value.
`int`	2	**`getAndAdd`**`(int delta)` Atomically adds the given value to the current value.
`int`	2	**`getAndDecrement`**`()` Atomically decrements by one the current value.
`int`	2	**`getAndIncrement`**`()` Atomically increments by one the current value.
`int`	2	**`getAndSet`**`(int newValue)` Atomically sets to the given value and returns the old value.
`int`	2	**`incrementAndGet`**`()` Atomically increments by one the current value.
`int`	1	**`intValue`**`()` Returns the value of the specified number as an `int`.
`void`	1	**`lazySet`**`(int newValue)` Eventually sets to the given value.
`long`	1	**`longValue`**`()` Returns the value of the specified number as a `long`.
`void`	1	**`set`**`(int newValue)` Sets to the given value.
`String`	1	**`toString`**`()` Returns the String representation of the current value.
`boolean`	∞	**`weakCompareAndSet`**`(int expect, int update)` Atomically sets the value to the given updated value if the current value == the expected value.

Conclusion

- Bilan
- Perspectives

Bilan

Attention aux contre-intuitions !

- `x++` ne signifie pas ≪ incrémenter `x` ≫
- `A ; B` ne signifie pas ≪ exécuter `A` puis `B` ≫
- ≪ Écrire dans une variable ≫ est une notion très floue
- Lire/écrire ne permet pas de tout faire
- `getAndIncrement()` et `get()`/`Increment()` n'ont rien à voir
- ...

Calcul réparti en mémoire partagée

- La concurrence crée de nouveaux problèmes de sûreté
- La synchronisation pour les résoudre limite la vivacité

Construction d'objets partagés

Beaucoup d'approches

Du moins au plus performant (en général)

- Constructions universelles non bloquantes
- Verrouillage à gros grains
- Algorithme wait-free dédié
- Verrouillage à grains fins
- Algorithme lock-free dédié

Perspective

Mémoire transactionelle logicielle

```
void increment (Register<Integer> X) {
  Transaction t = new STMTransaction();
  while (!t.isCommited()) {
    try {
      t.begin();
      X.write(t, X.read(t) + 1);
      t.try_to_commit();
    } catch (AbortException e) {}
  }
}
```

Perspective

Mémoire transactionelle logicielle

```
void increment (Register<Integer> X) {
  Transaction t = new STMTransaction();
  while (!t.isCommited()) {
    try {
      t.begin();
      X.write(t, X.read(t) + 1);
      t.try_to_commit();
    } catch (AbortException e) {}
  }
}
```

Projet

1. Lire l'article sur Madoc
2. Implémenter l'algorithme TL2

IR09 Damien Imbs et Michel Raynal. *Software transactional memories : an approach for multicore programming*. PaCT 2009