Programmation concurrente

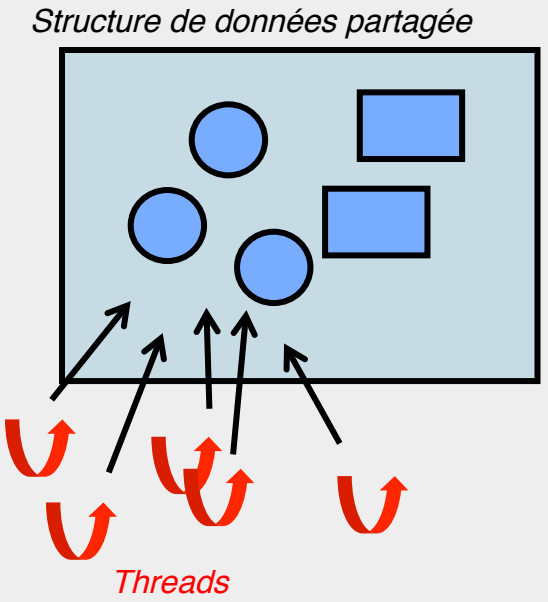
Bilan / Wrap-Up

Polytech/INFO 4, 2022-2023

Fabienne Boyer
UFR IM2AG, LIG, Université Grenoble Alpes
Fabienne.Boyer@imag.fr

Où en est t'on

- **Concepts**
  - Processus et threads
  - Notion d 'exclusion mutuelle
- **Outils**
  - Verrous
  - Moniteurs
  - Sémaphores
- **Méthodologies de programmation**
  - Tableau de gardes/actions
  - Solution directe
  - Solutions FIFO
  - Solutions à base de priorités
  - Gestion des interblocages
- **Problèmes étudiés**
  - Allocation de ressource
  - Producteurs-Consommateurs
  - Lecteurs-Rédacteurs
  - Philosophes
- **Wrap-up**



Processus et threads

- **Très utilisés dans le domaine des applications interactives, des applications de calcul, des serveurs applicatifs (serveurs Web et autres)**

  Exemple: le serveur Apache (46% des serveurs Web dans le monde) utilise les processus pour résister aux fautes et les threads pour la parallélisation du traitement des requêtes

  https://httpd.apache.org/docs/2.4/fr/mod/worker.html

Outils pour synchroniser les threads

- **Verrous: pour mettre en œuvre des sections critiques - blocs de code exécuté de manière atomique** → ***au maximum 1 thread en exécution dans le bloc***

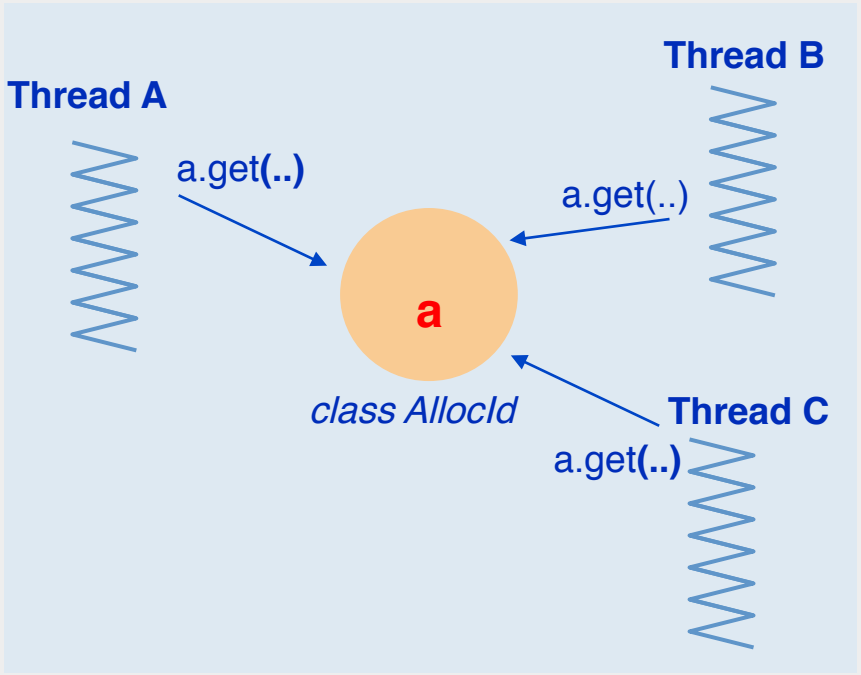
- **Moniteurs / Sémaphores : pour gérer de l'allocation de ressources ou coordonner des threads**

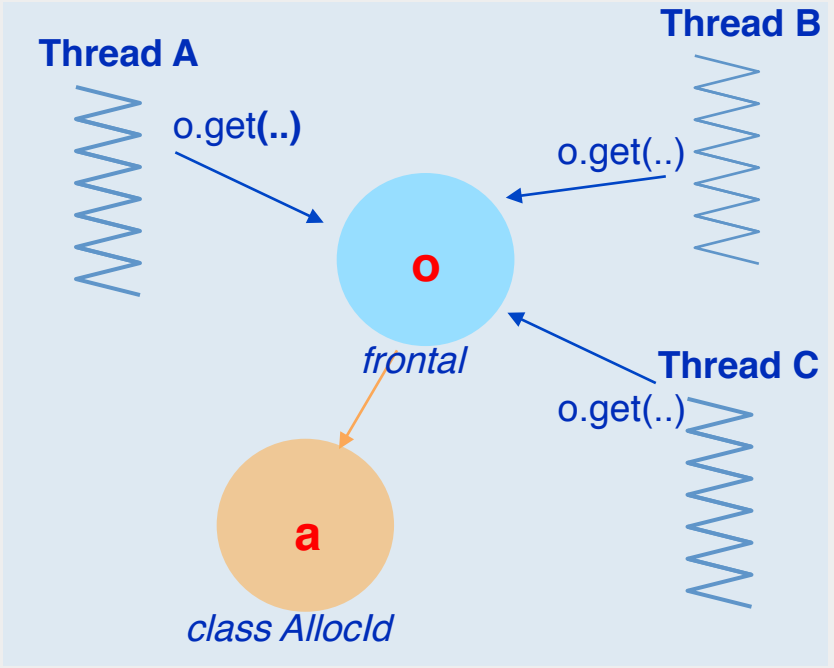
- **Ces outils permettent d'implémenter des classes *thread-safe***

  - A class is thread safe if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

  ref: Java Concurrency in Practice

Classes thread-safe vs non thread-safe





Si la classe AllocId est thread-safe :

→ T1, T2, T3 peuvent invoquer *a.get()* concurremment

Si AllocId n'est pas thread-safe :

→ T1, T2, T3 doivent se synchroniser pour invoquer *a.get()*, par exemple en appelant une méthode synchronisée sur un objet frontal

→ Attention aux alias sur l'objet *a*….

Classes thread-safe vs non thread-safe

- **Si l'on a le contrôle d'une classe, on peut la rendre thread-safe**
- **Si l'on n'a pas le contrôle (ex: librairies) alors il faut gérer la synchronisation depuis l'extérieur**

*Example taken from **ArrayList / Javadoc**:*

Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.

(A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.)

Quelques classes Java « thread-safe »

- **Collections** Interface Collection<E>
  - Classe synchronizedCollection<E> (*wrapper class*)
- **Listes** Interface List<E>
  - Classe Vector<E>
  - Classe SynchronizedList<E>
  - Classe CopyOnWriteArrayList<E>
- **Ensembles** Interface Set<E>
  - Classe SynchronizedSet<E>
- **Associations** Interface Map<K,V>
  - Classe HashTable<K,V>
  - Classe SynchronizedMap<K,V>
  - Classe ConcurrentHashMap<K,V>
- **Queues** Interface Queue<E>
  - Classe ArrayBlockingQueue<E>
  - Classe ConcurrentLinkedQueue<E>
  - Classe PriorityBlockingQueue<E>
  - Classe SynchronousQueue<E>
  - Classe LinkedBlockingQueue<E>

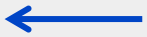
Attention, ces classes fournissent différents niveaux de cohérence et de performances

Classes Java thread-safe : différents niveaux de cohérence et de performances

Synchronized Collections

- Classes Vector, Hashtable, Stack, ..
- Wrapper classes (SynchronizedList, SynchronizedSet, SynchronizedMap, SynchronousQueue)
- Chaque méthode est *synchronized*
- Attention, cela ne fournit pas une atomicité multi-invocations

```
Vector v = ..
if (!v.contains(o))
   v.add(o);
```



unicité de o dans v non garantie, sauf si on synchronise tout le bloc

- Itérateurs fail-fast

Concurrent Collections (new)

- Classes CopyOnWriteArrayList, ConcurrentHashMap
- Classes implementing BlockingQueue
- Classes optimisées pour certains *patterns de concurrence* (bcp d'écritures / peu de lectures, collections de petite taille, …)
- Ne fournit pas une atomicité multi-invocations
- Itérateurs fail-safe

Types d'itérateurs

- **Un itérateur *fail-fast* lève l'exception *ConcurrentModificationException* si la collection est modifiée durant l'itération**
  - Tout objet contient un champ *modification-count*
  - Un itérateur teste si ce champ change durant son itération
  - En général exception non levée si un élement est retiré de la collection via l'itérateur

- **Un itérateur *fail-safe* itère sur une copie de la collection (il ne lève donc jamais l'exception *ConcurrentModificationException)***
  - Appelés aussi itérateurs de type *snapshots*
  - Différentes mises en œuvre
    - Création d'une copie (snapshot) *au* moment de la création de l'itérateur
    - Création d'une copie lors de la modification de la collection (copy-on-write)

SynchronizedList vs CopyOnWriteArrayList

SynchronizedList

- Toute méthode vérrouille la liste entière
- Fail-fast itération

Usage

- Listes volumineuses (car liste non recopiée en mémoire sauf pour du resizing)
- Beaucoup de manipulations en écriture par rapport aux lectures

CopyOnWriteArrayList

- Vérrouille la liste seulement pour les méthodes en écriture
- Pas de verrouillage pour les méthodes en lecture
- Fail-safe itération

Implémentation

- Toute méthode en écriture est exécutée sur une copie fraîche de la liste (copy-on-write)

Usage

- Petites listes
- Beaucoup de manipulations en lecture par rapport aux écritures
- Beaucoup d'itérations (peu coûteuses car non synchronisées et ne génèrent pas de copies)

SynchronizedHashMap vs ConcurrentHashMap

- **SynchronizedHashMap**
  - Toute méthode (put, get, ..) vérrouille la liste entière
  - Fail-fast itération

Usage

- Besoin d'itérer sur la version courante de la map

- **ConcurrentHashMap**
  - Lectures // écritures (lectures sales)
  - Ecritures // (segmentation de la Map)
  - Fail-safe itération

Implémentation

- Verrouille seulement le bucket courant pour une méthode en écriture (put, remove, ..)
- Ne verrouille rien pour une opération en lecture
- Itérateur de type snapshot

Usage

- Besoin de performances (facteur 3 / SynchronizedMap)

LinkedBlockingQueue vs ConcurrentLinkedQueue

- **LinkedBlockingQueue**
  - File de taille bornée ou pas
  - Blocking (si file vide ou pleine)
  - Lock-based
  - Adapté au pattern général producteurs/consommateurs (MPMC)
- **ConcurrentLinkedQueue**
  - File de taille non bornée
  - Non blocking
  - Lock-free
  - Marche pour un producteur / un consommateurs (SPSC)

→ **File de taille non bornée : attention, si la production est plus rapide que la consommation, la file peut grandir sans limite**

Quelques classes Java non « thread-safe »

- **Collections**
  - Interface Collection<E>
    - Classe AbstractCollection<E>
- **Listes**
  - Interface List<E>
    - Classe ArrayList<E>
    - Classe LinkedList<E>
- **Ensembles**
  - Interface Set<E>
    - Classe HashSet<E>
    - Classe TreeSet<E>
- **Associations**
  - Interface Map<K,V>
    - Classe HashMap<K,V>
    - Classe TreeMap<K,V>)

Les méthodes ne sont pas synchronisées

Si concurrence, il faut synchroniser les accès depuis l'extérieur

A savoir, nombre de ces classes ont des itérateurs fail-fast

Usage de collections non thread-safe

- Si l'on doit utiliser une collection non thread-safe dans un contexte concurrent, un moyen de faire est de wrapper (*encapsuler*) la collection dans une collection synchronisée

  Par exemple, pour une liste:

```
List slist = Collections.synchronizedList(new ArrayList(...));

// wrapper la liste au moment de sa création pour éviter
// tout accès non synchronisé à la liste
```

Dans tous les cas

- En programmation concurrente, il y a souvent plusieurs options possibles et c'est au programmeur de faire le choix de la classe la plus adaptée

- Pour une atomicité multi-invocations, il faut souvent définir des blocs synchronisés

```
List slist = Collections.synchronizedList(new ArrayList(...));
…
  synchronized(slist) {
    for (int i=0; i < list.size();) {
                if (toRemove(list.get(i)))
                   list.remove(i);
                i++;
    }
  }
…
```

Notion de cohérence mémoire

Visibilité d'une variable : la dernière valeur de la variable est visible par tout thread qui accède à la variable.

→ Pas toujours le cas lorsque les variables sont mises en cache localement à un thread ou à un CPU, ou sont gérées de manière optimisée par le compilateur

```
class X () {
        static boolean ready = false;
        static void waitReady() { while (!ready) ;}
        static void setReady() { ready = true; }
}
```

cache local / T1
ready = false

```
Thread T1:
    …
    X.waitReady();
    …
```

cache local / T2
ready = true

```
Thread T2:
    …
    X.setReady();
    …
```

➔ T1 peut rester bloqué indéfiniment sur *waitReady()*

Cohérence mémoire

- **Visibilité provoquée sur l'entrée et la sortie d'un bloc synchronized**
  - After we exit a synchronized block, we release the monitor, which has the effect of flushing the cache to main memory, so that writes made by this thread can be visible to other threads.
  - Before we can enter a synchronized block, we acquire the monitor, which has the effect of invalidating the local processor cache so that variables will be reloaded from main memory. We will then be able to see all of the writes made visible by the previous release. *from the javadoc..*
- **Visibilité provoquée sur l'usage d'un objet thread-safe**
  - Ex: Semaphore - Memory consistency effects: actions in a thread prior to calling a "release" method such as release() happen-before actions following a successful "acquire" method such as acquire() in another thread. *from the javadoc..*
  -
- **Visibilité basée sur l'usage de variables *volatile***
  - *Variables jamais mises en cache donc toujours visibles*

Cohérence mémoire

Solution à base de moniteur

```
class X () {
        static boolean ready = false;
        static synchronized void waitReady() { while (!ready) wait();}
        static synchronized void setReady() { ready = true; notify(); }
}
```

Solution à base de volatile

```
class X () {
        static volatile boolean ready = false;
        static void waitReady() { while (!ready) ; [yield();]}
        static void setReady() { ready = true; }
}
```

Thread T1:

```
…
X.waitReady();
…
```

Thread T2:

```
…
X.setReady();
…
```

Variables volatiles & variables atomiques

- **Variable volatile :**
  - jamais mise en cache donc toujours visible, *mais attention : lire puis écrire une variable volatile n'est pas globalement atomique*

- **Variables atomiques (package java.util.concurrent)**
  - AtomicBoolean, AtomicInteger, AtomicLong, AtomicReference, AtomicLongArray, AtomicReferenceArray

  - Fournissent des opérations composées (lecture+écriture) atomiques
  - Les variables atomiques sont mises en œuvre avec des variables volatile

```
AtomicInteger i = new AtomicInteger(0);
 …
 int current = i.incrementAndGet();
…
```

Programmation concurrente

- **Garantir cohérence et visibilité pour les données partagées**
- **Pour cela, sont à notre disposition**
  - Les classes thread-safe fournies par la librairie java.concurrent (collections concurrentes, variables atomiques, ..)
  - La directive volatile et les variables atomiques
  - Les outils de synchronisation (moniteurs, sémaphores, verrous)
- **Additionnellement**
  - JCTools (Java Concurrency Tools) : offre des classes additionnelles pour la concurrence
    - SPSC/MPSC/SPMC/MPMC Bounded lock free queues
    - SPSC/MPSC Unbounded lock free queues
    - Single Writer Map/Set implementations
    - …