Paris JUG

www.parisjug.org

09/06/2009

09/06/2009

Java Concurrent

Denis Ah-Kang

Atos Origin

Open Source Center

www.parisjug.org

Programmation concurrente

- **Loi de Moore commence à atteindre ses limites**
  - Multiplication du nombre de processeurs sur les puces
- **Meilleure utilisation des ressources**
- **Meilleur temps de réponse des programmes**

Notions

- **Atomicité**
  - Ensemble d'actions exécuté sans interférence
- **Visibilité**
  - Effet d'un thread visible par un autre
- **Ordonnancement**
  - Ordre d'exécution des actions

synchronized

- **Pose de locks**

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

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

volatile

- Synchronized-like sans pose de locks et sans le caractère atomic
- Simplicité
- Scalabilité

volatile

```
public class StoppableTask extends Thread {
    private volatile boolean pleaseStop;

    public void run() {
        while (!pleaseStop) {
            // do some stuff...
        }
    }

    public void tellMeToStop() {
        pleaseStop = true;
    }
}
```

Problèmes classiques

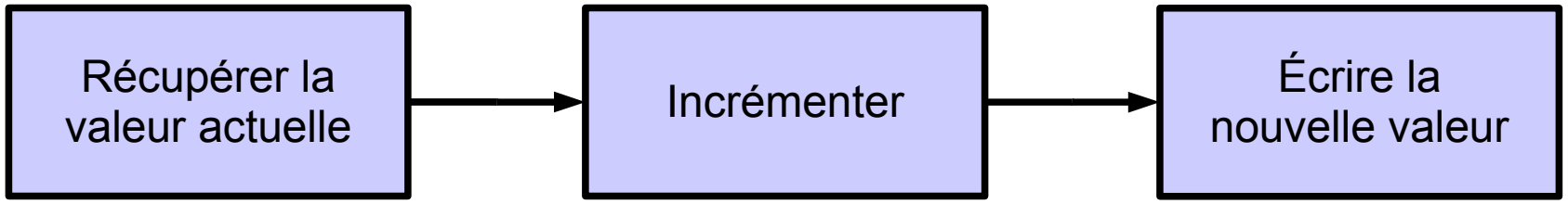
- **Synchronized: lock**
  - Impossible d'arrêter un thread lorsque celui-ci attend un lock
  - Possibilité de deadlock

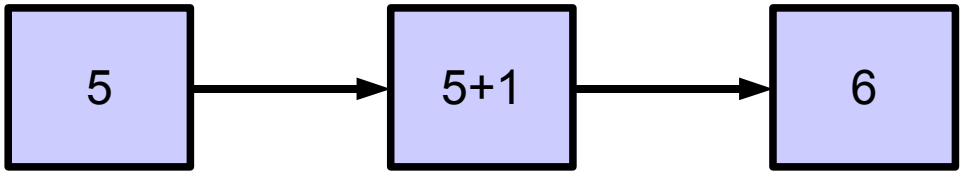
Thread 1	Thread 2
Get A	Get B
Get B	Get A
Release B	Release A
Release A	Release B

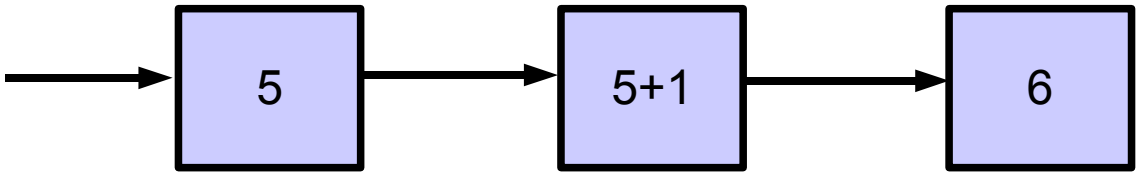
- **Baisse des performances**
- **Possibilité d'inversion des priorités (Mars Pathfinder)**
- **Méconnaissance des problèmes de concurrence**

i++

- **Atomique?**
- **3 étapes:**







- **Lecture-modification-écriture atomique**

L'API Java Concurrent

- JSR-166
- Depuis Java 5
- Introduite par Doug Lea
- Objets avec un haut niveau d'abstraction
- Simple d'emploi

Variables atomiques

- Algorithmes non-bloquants très efficaces
- Variables atomiques = variables volatiles + atomicité
- 12 classes de variables atomiques (AtomicInteger, AtomicBoolean, AtomicLong, …)
- AtomicInteger: getAndIncrement()

Future<T> 1/2

- **Abstraction du résultat d'un calcul asynchrone**
- **Méthodes:**
  - boolean cancel (boolean): tentative d'arrêt de la tâche
  - boolean isCancelled()
  - boolean isDone()
  - **T** get(): attente bloquant de la fin du traitement et retourne le résultat

Future<T> 2/2

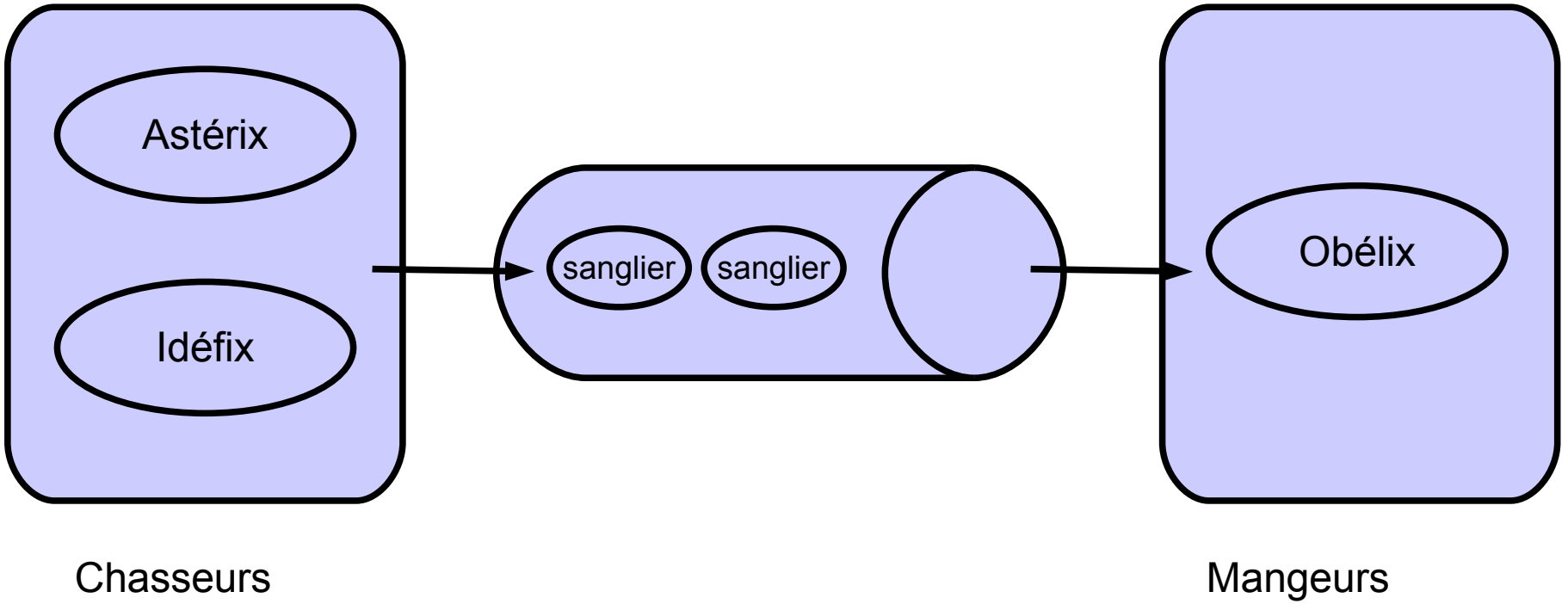
- **ExecutorService**
  - execute(...) - exécution à un moment donné
  - submit(...) - exécution à un moment donné et retourne « Future »

```
ExecutorService executorService =
    Executors.newCachedThreadPool();
Callable<String> check = new Callable<String>()(
        public String call() throws Exception {
            //some actions
        }
);
Future<String> future =
    executor.submit(check);
String value = future.get();
```

BlockingQueue

- **Producteur/Consommateur:**
  - Thread(s) producteur:
    - queue.put(obj) : bloque si la queue est pleine
  - Thread(s) consommateur:
    - queue.take(): bloque jusqu'à ce qu'un objet soit disponible

BlockingQueue Demo 1/4



BlockingQueue Demo 2/4

- **Producteur**

```
public class ChasseurSanglier implements Callable {
    private final String nomProducteur;
    private final BlockingQueue<Sanglier> queue;
    private final TimeUnit timeUnit;
    private final long intervalle;
    …
    public Void call() {
        try {
            while (true) {
                timeUnit.sleep(intervalle);
                Sanglier sanglier = new Sanglier(nomProducteur);
                queue.put(sanglier);
            }
        } catch (InterruptedException e) {
            System.out.println(nomProducteur + " a fini.");
        }
        …
    }
}
```

BlockingQueue Demo 3/4

- **Consommateur**

```
public class MangeurSanglier implements Callable {
    private final String nom;
    private final BlockingQueue<Sanglier> queue;
    private final TimeUnit timeUnit;
    private final long intervalle;
    …
    public Void call() {
        try {
            while (true) {
                Sanglier sanglier = queue.take();
                timeUnit.sleep(intervalle);
            }
        } catch (InterruptedException e) {
            System.out.println(nom + " a terminé.");
        }
        …
    }
}
```

BlockingQueue Demo 4/4

```
ExecutorService executorService =
    Executors.newCachedThreadPool();
//Max 10 sangliers dans la queue
BlockingQueue<Sanglier> sanglierQueue = new
    LinkedBlockingQueue<Sanglier>(10);

// Asterix attrape un sanglier toutes les secondes
executorService.submit(
    new ChasseurSanglier("Asterix",
                         sanglierQueue,
                         TimeUnit.SECONDS,
                         1));
// Obelix mange un sanglier toutes les deux secondes
MangeurSanglier obelix =
    New ConsommateurSanglier("Obelix",
                          sanglierQueue,
                          TimeUnit.SECONDS,
                          2);
```

java.util.concurrent

- Lock
- CountDownLatch
- CyclicBarrier
- ConcurrentHashmap<K,V>

Concurrent dans Java 7

- Fork/Join
- Phasers
- LinkedTransferQueue
- ConcurrentReferenceHashMap
- Fences

Conclusion

- identifier les problèmes de concurrence
- java.util.concurrent

Bibliographie / liens

- Java Concurrency in Practice – Brian Goetz
- Doug Lea's Home Page - http://g.oswego.edu/

Questions / Réponses

www.parisjug.org

Sponsors

Merci de votre attention!

www.parisjug.org

Licence

- http://creativecommons.org/licenses/by-nc-sa/2.0/fr/