Programmation concurrente

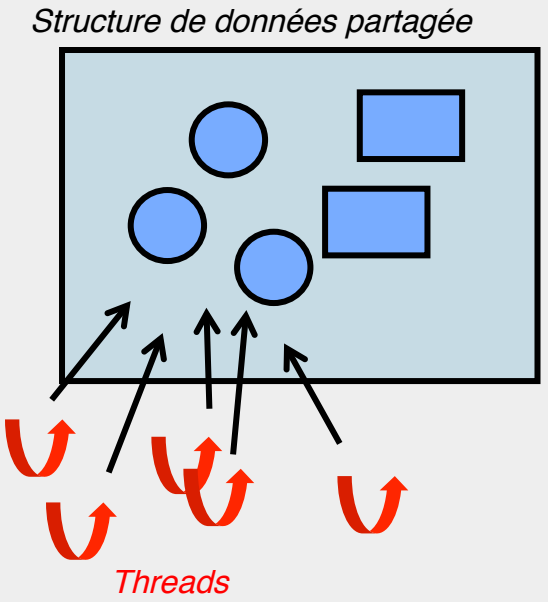
Serveurs multi-threadés et Exécuteurs

Polytech/INFO 4, 2022-2023

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

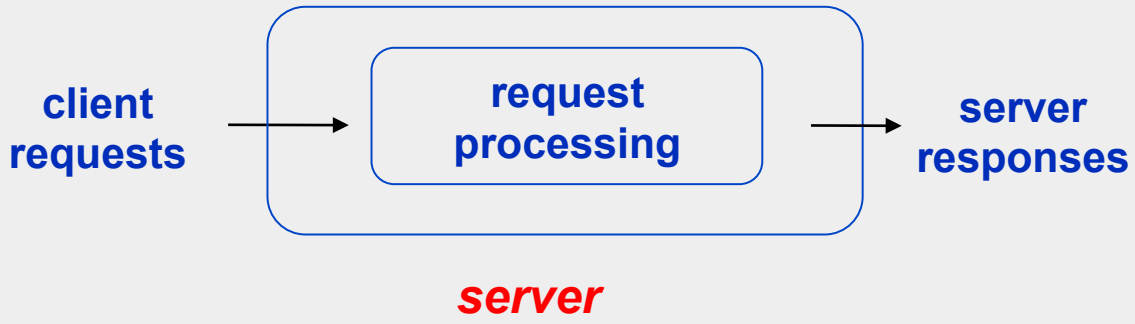
Ou en est t'on

- **Processus**
- **Threads**
- **Notion d 'exclusion mutuelle**
- **Outils de synchronisation**
  - Verrous
  - Moniteurs
  - Sémaphores
- **Méthodologie 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
- **Outil de synchronisation additionnel: les exécuteurs**
  - Usage pour la mise en œuvre de serveurs applicatifs
  - Usage pour la parallélisation de traitements
- **Wrap-up**



Serveur applicatif

- **Une application concurrente telle que**
  - Serveurs Web
  - Serveurs de fichiers
  - Serveurs de streaming
  - …
- **Qui reçoit des requêtes, les exécute et renvoie le résultat aux clients**



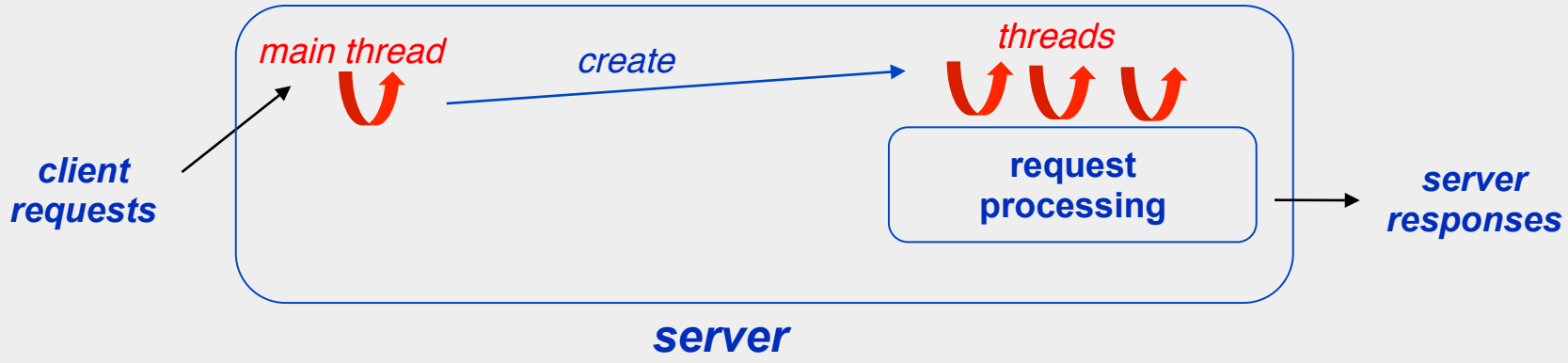
Structuration des serveurs (applicatifs)

- **Serveur mono-threadé**
- **Serveur multi-threadé**
  - Diverses stratégies
    - Un thread par requête
    - Un pool de threads pour l'ensemble des requêtes
    - …
- **Serveur à base d'exécuteurs**
  - Diverses stratégies
    - Mono-threadé
    - Multi-threadé
    - …

Serveur mono-threadé


```
class SingleThreadServer {
 public static void main(String[] args) throws IOException {
   initServer();
   while (true) {
         Request request = receiveRequest();
         handleRequest(request);
   }
 }
}
```

Serveur multi-threadé avec création de threads

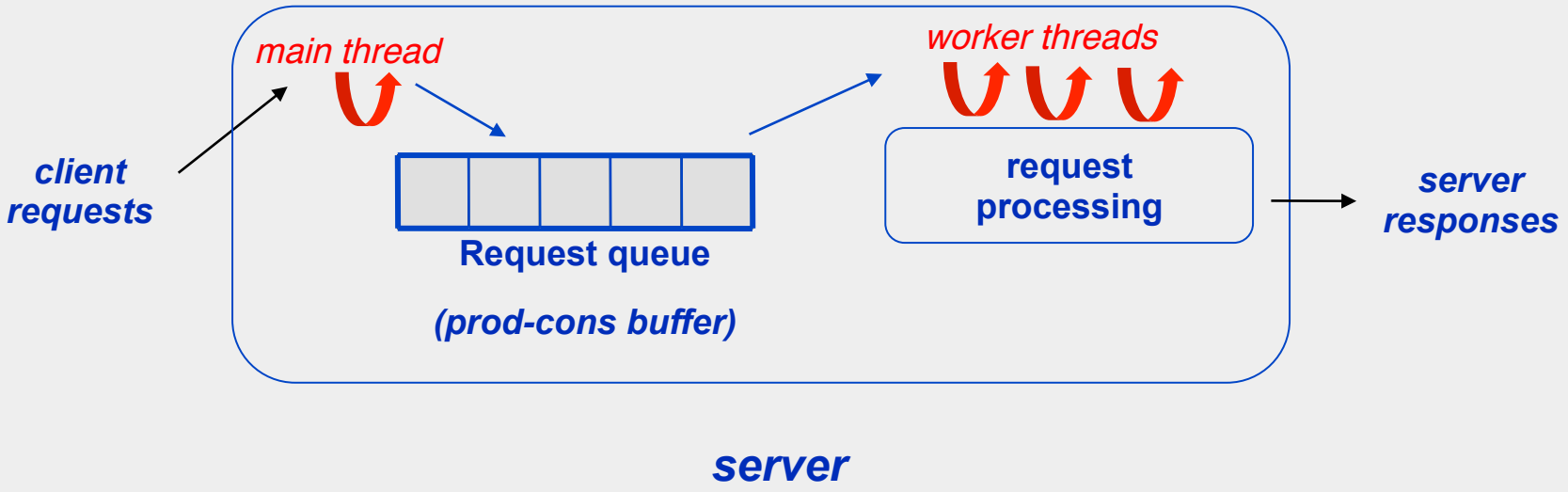



```
class MultiThreadServer {
    public static void main(String[] args) throws IOException {
        initServer();
        while (true) {
                Request request = receiveRequest();
                Runnable worker = new Runnable() {
                        public void run() { handleRequest(request); }
                 };
                new Thread(worker).start();
        } ..
```

Serveur multi-threadé avec création de threads

- **Requêtes traitées en parallèle**
  - Meilleurs temps de réponse
- **Mais..**
  - Coût CPU lié aux créations / destructions de threads
  - Coût mémoire si grand nombre de threads en parallèle
  - Nombre de threads parallèles limité par la capacité VM

Serveur multi-threadé avec pool de threads



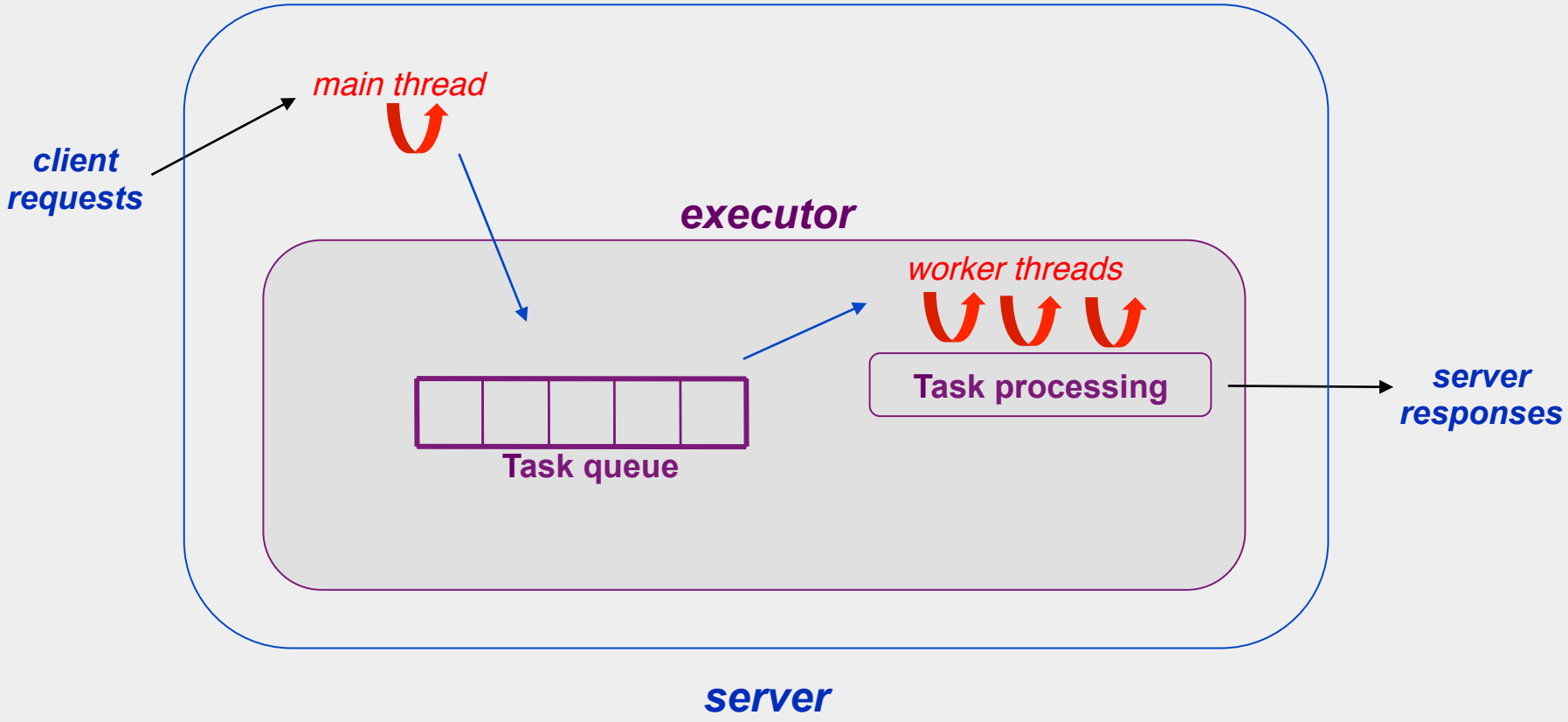
Serveur multi-threadé avec de pool de threads

```
Class ThreadPoolServer {
 ProdCons requests = new ProdCons(100);

 public static void main(String[] args) throws IOException {
    initServer();
    launchWorkers(requests);
    while (true) {
           Request request = receiveRequest();
           requests.put(request);
    }
 }
}

class Worker extends Thread{
   Worker(ProdCons prodcons){this.setDeamon(true); this.prodcons = prodcons;}
   public void run(){
          while (true) {
             Request request = requests.get();
             handleRequest(request);
          };
   }
 }
```

Serveur multi-threadé avec exécuteur



Serveur multi-threadé avec un exécuteur

```
class MultiTaskServer {
private static final int NTHREADS = 30;
private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);

 public static void main(String[] args) throws IOException {
    initServer();
    while (true) {
         Request request = receiveRequest();
         Runnable task = new Runnable() {
                   public void run() {handleRequest(request); }
         };
         exec.execute(task);
    }
   }
 }
```

Notion d'exécuteur

- **Généralisation du pattern producteurs-consommateurs (orienté tâches)**
  - Les producteurs produisent des tâches à exécuter
  - Les consommateurs exécutent les tâches
- **Un exécuteur reçoit et exécute des tâches**
  - Maintient une liste de tâches à exécuter
  - Gère des *worker* threads pour exécuter les tâches
- **Nombreuses politiques de gestion du pool de worker threads**
  - SingleThreadExecutor
  - FixedThreadPool *fixed number of threads*
  - CachedThreadPool

    create as many threads it needs to execute the task in parallel

    If a thread is not used during 60 seconds, it will be terminated and removed from the pool
  - ScheduledThreadPool

    pool to schedule tasks to execute after a given delay has expired

Interface élémentaire *Executor*

- **L'interface Executor permet de soumettre des tâches à exécuter**

```
public interface Executor {
    void execute(Runnable task);
}
```

- **Exemple**

```
Executor executor = Executors.newSingleThreadExecutor();
executor.execute(new Runnable()
          { public void run() { System.out.println("task 1"); .. } });
executor.execute(new Runnable()
         { public void run() { System.out.println(" task 2"); .. } });
```

Interface *ExecutorService*

- **Permet de contrôler le cycle de vie d'un exécuteur**
  - A l'instant courant, des tâches peuvent être en cours d'exécution et d'autres en attente
  - Shutdown: pas de tâche nouvelle acceptée, mais les tâches en attente seront traitées
  - ShutdownNow: arrêt brutal immédiat
  - isTerminated: vrai quand toutes les tâches ont terminé

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit) throws
    InterruptedException;
    // ... Additional methods for task submission
}
```

Exemple

```
Executor exec = Executors.newSingleThreadExecutor();
executor.execute(new Runnable() {public void run() {..} });
executor.execute(new Runnable() {public void run() {..} });
exec.awaitTermination(..);
System.out.println("All done");
```

Création d'un exécuteur

- **Instancier une classe implémentant ExecutorService**
  - ScheduledThreadPoolExecutor
  - ThreadPoolExecutor (corePoolSz, MaxPoolSz, KeepAliveTime)
  - ForkJoinPool: exécuteur pour les tâches de type *fork and join (tâches qui se décomposent en sous-tâches avec aggrégation des résultats)*

- **Ou bien utiliser la classe *factory* Executors**
  - newFixedThreadPool **(fabrique)**
  - newCachedThreadPool
  - newSingleThreadExecutor
  - …

Serveur multi-threadé avec un exécuteur

```
class MultiTaskServer {
private static final int NTHREADS = 30;
private static final Executor exec = Executors.newFixedThreadPool(NTHREADS);

 public static void main(String[] args) throws IOException {
    initServer();
    while (true) {
      Request request = receiveRequest();
      Runnable task = new Runnable(){public void run(){handleRequest(request);}};
      exec.execute(task);
    }
   }
 }
```

Types de tâches

- **Tâches qui implémentent Runnable**
  - Cela impose que les tâches ne retournent pas de résultat à l'exécuteur

```
public interface Runnable {
        void run();
}
```

  - Cela n'est pas une limitation pour la mise en œuvre de serveurs, car les tâches renvoient le résultat au client directement
  - Mais cela peut être une limitation dans d'autres cas, par exemple si on utilise un exécuteur pour paralléliser des traitements dans le but de gagner en efficacité

- **Tâches qui implémentent Callable**
  - Ces tâches peuvent retourner un résultat

```
public interface Callable<V> {
        V call() throws Exception;
}
```

Notions de Callable et Futures

- **Deux notions supplémentaires pour gérer des tâches avec résultats**
  - **Callable** : tâche pouvant retourner un résultat et émettre une exception
  - **Future**: objet qui permet de contrôler la vie d'une tâchede type Callable (l'annuler, savoir si elle est terminée, récupérer son résultat, ..)

- **Exemple d'usage**
  - Requête complexe, dont le traitement met en jeu des calculs lourds qui peuvent être parallélisés
  - La requête est décomposée en tâches exécutées en parallèle par un exécuteur
  - Chaque tâche délivre un résultat partiel
  - Le résultat final est calculé à partir des résultats partiels

Interfaces Callable et Future

```
public interface Callable<V> {
   V call() throws Exception;
}

public interface Future<V> {
   boolean cancel(boolean mayInterruptIfRunning);
   boolean isCancelled();
   boolean isDone();
   V get() throws InterruptedException, ExecutionException, ..
   V get(long timeout, TimeUnit unit) throws InterruptedException, ..
}

public interface ExecutorService extends Executor {
<T> Future<T> submit(Callable<T> task)
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
..
```

Exemple

```
public class CallableExample {

  public static class SumIntValues implements Callable {
    private int values[];
    public SumIntValues (int[] values) {this.values = values;}
    public Integer call() {
        int sum=0;
        for (int i=0; i<values.length; i++) sum += values[i];
        return new Integer(sum);}
  }

  public static void main(String args[]) throws Exception {
    ExecutorService pool = Executors.newFixedThreadPool(3);
    Set<Future<Integer>> futures= new HashSet<Future<Integer>>();
    for (String fname: args) {
          int[] values = readFileContent(fname);
          Callable<Integer> callable = new SumIntValues (values);
          Future<Integer> future = pool.submit(callable);
          futures.add(future);
    }
    int sum = 0;
    for (Future<Integer> future : futures) {
       sum += future.get();
    }
   System.out.printf("The total sum is %s%n", sum.intValue());
 }
}
```

Oracle docs