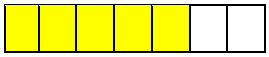
Java

Version 1.0.0
Niveau requis : 5/7



MultiThreading

Sommaire

I. Préambule

I.I. *Objet*

L'objet de ce document est de présenter et de décrire par l'exemple l'usage de la programmation Multi Thread.

I.II. *Pré-requis*

Nous allons exploiter dans ce document un projet Java maven 17 sous IntelliJ-Community sous Windows.

Il est nécessaire préalablement d'avoir donc :

- Java 17 : https://bell-sw.com/pages/downloads/
- Maven Apache : https://maven.apache.org/download.cgi
- IntelliJ-Community : https://www.jetbrains.com/idea/download/?section=windows

Des connaissances sur la programmation Java est nécessaire avec de comprendre les exemples.

I.III. *Versions du document*

Version	Date	Auteur	Description
1.0	13/08/2023	Péquignat.eu	Création du document

I.IV. *Documents de référence*

#	Document	Version	Auteur(s)
[R1]	https://stacklima.com/solution-producteur-consommateur-utilisant-des-threads-en-java/	05/07/2022	StackLima

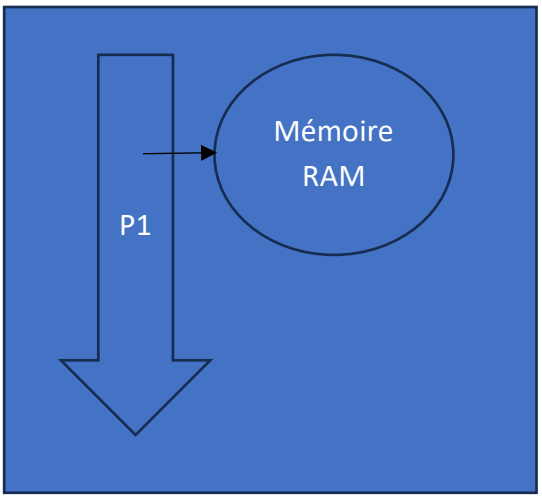
II. Principe de la programmation concurrente (MultiThreading)

II.I. *Rappel de la programmation synchrone*

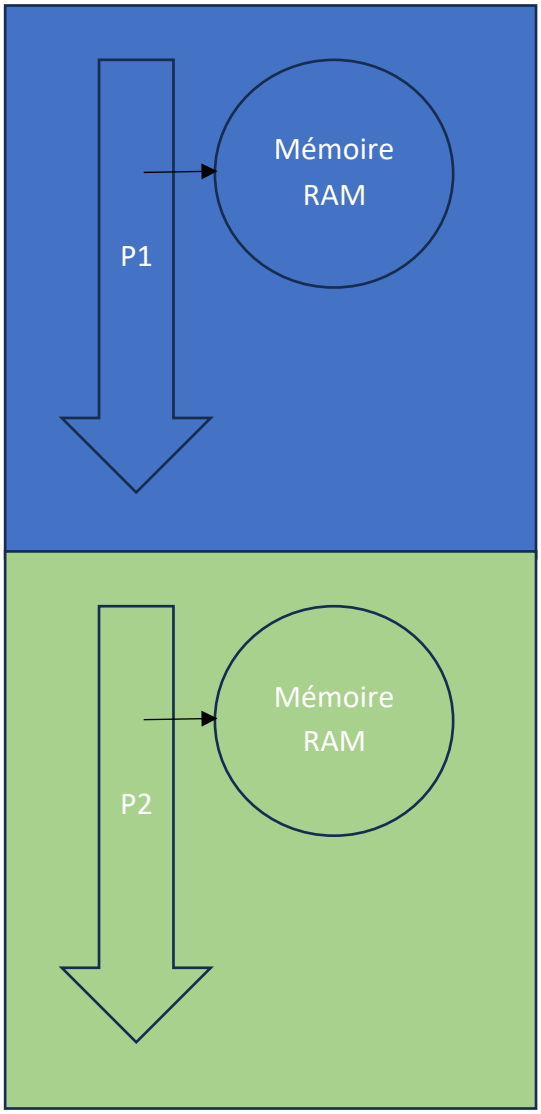
Un programme Java de base est synchrone, c'est-à-dire que chaque instruction de code rend le code bloqué tant que l'opération n'est pas terminée par le processeur.

On peut représenter donc par une ligne verticale le fonctionnement d'un programme qui s'exécute dans le temps de haut en bas.

On peut prendre l'exemple d'un programme simple qui manipule des données en mémoire de manière isolée.



Aussi, dans ce cas, si on a besoin de réaliser deux opérations indépendantes par la programmation synchrone, on doit attendre la fin du programme P1, avant de lancer le programme P2, P3…
Donc si chaque programme P1, P2 … PN met 1 secondes. Le temps global sera de N secondes.



II.II. *Principe de la programmation concurrente*

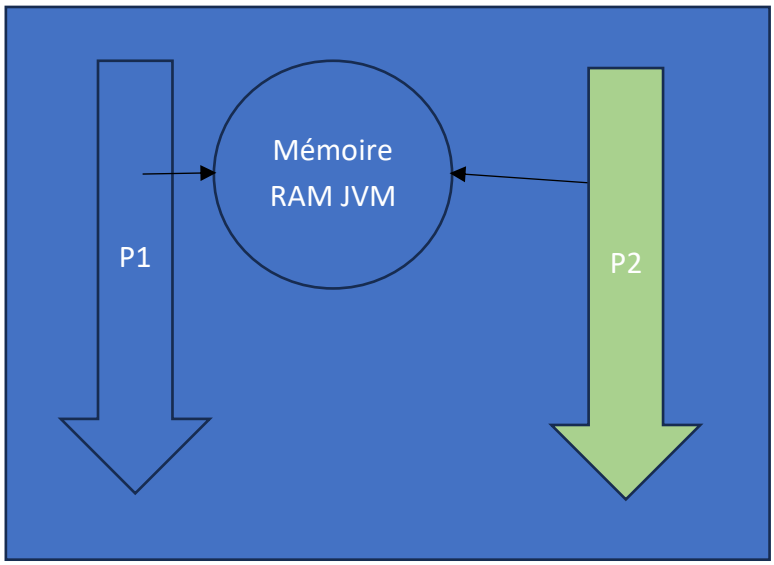
Le principe de la programmation concurrente consiste à rendre la programmation dite synchrone en asynchrone. En java, il est utilisé les Thread pour lancer un code en « arrière-plan » ou asynchrone, ce qui a pour effet de continuer le code dans le programme principale (Main).

La mémoire, utilise l'espace de la JVM et n'a pas plus de RAM dédié copiée et n'est pas isolé contrairement à la programmation dite multi processus. En java, les Threads partage la mémoire de la JVM contrairement à d'autres langage en mode processus.

Attention : La programmation concurrente nécessite de s'assurer qui n'y a pas chevauchement entre la lecture et l'écriture en mémoire, sur le réseau (Base de données) ou disque.

II.III. *Revenons à l'exemple*

Dans l'exemple, avec la programmation concurrente, si on a N codes qui durent chacun en synchrone 1 secondes, donc soit N secondes en programmation synchrone, alors il est possible d'avoir si N < Nombre de cœurs virtuelle de la machine et que la quantité de Ram manipulée totale par l'addition des N programme ne dépasse pas la RAM maximum de la JVM (Défaut 4Go), alors le temps globale de l'exécution du code sera 1 seconde (temps le plus long entre P1 et PN) + quelque mini secondes pour la création des Thread et Classe Runnable.



Dans le cas où les N programme ne consomme que peu de CPU, alors il est possible d'observer le même gain de temps avec N pouvant être bien plus important : des centaines voir plus, cela dépend des caractéristiques de votre machine.

II.IV. *Fin de l'exécution d'un ou des Threads*

Maintenant que l'on a vu les bénéfices de la programmation concurrente, il convient dans un programme plus riche, d'attendre la fin d'exécution de tous les Thread avant de pouvoir réaliser l'opération suivante.

Pour cela, il est nécessaire pour un Thread de lancer la fonction « join() » qui attend la fin du Thread.

Un Thread se termine dans la fonction **run()** du Runnable qui est contenu dans le Thread est terminée.

III. Mise en pratique : quelques exemples

III.I. *Exécution d'un Simple Thread*

Ci-dessous, l'exemple de manipulation d'un simple Thread contenant une classe MyRunnable.

Ce Thread est actif dans le Code avec comme constantes :

- NB_LOOP = 5
- SLEEP_MS = 100

Soit un peu de plus de 500 ms de durée de la fonction run().

III.I.1 Codes

```
MyRunnable runnable = new MyRunnable();
Thread thread= new Thread(runnable);
thread.start();
message("The code continue");
//To wait the end of the thread
try {
    message("Waiting");
    thread.join();
    message("Finished");
} catch (InterruptedException e) {
    LOG.info(e.getLocalizedMessage(),e);
}
```

```
package eu.pequignat.runnable;

import eu.pequignat.MyFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import static eu.pequignat.constant.Constant.NB_LOOP;
import static eu.pequignat.constant.Constant.SLEEP_MS;

public class MyRunnable implements Runnable, MyFunction {

    protected static final Logger LOG =
LoggerFactory.getLogger(MyRunnable.class);

    @Override
    public void run() {
        final String refMemory= this.toString();
        message(refMemory.concat(".run()"));
        message(refMemory.concat("> loop ").concat(String.valueOf(NB_LOOP)));
        for (int i = 0; i< NB_LOOP; i++){
            message(String.format("%s> loop:%d", refMemory, i));
            try {
                Thread.sleep(SLEEP_MS);
            } catch (InterruptedException e) {
                LOG.error(e.getLocalizedMessage(),e);
            }
        }
    }
}
```

```java
package eu.pequignat;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public interface MyFunction {

    Logger LOG = LoggerFactory.getLogger(MyFunction.class);

    default void message(String message){
        LOG.debug(String.format("%s> %s", getName() ,message));
    }

    default String getName(){
        return this.getClass().getName();
    }

    void run() ;
}
```

III.I.2 Exemple à l'exécution

```
15:15:21.267 [main] INFO  eu.pequignat.App - case1SimpleThreadRunnable> Just One Thread

15:15:21.268 [main] INFO  eu.pequignat.App - eu.pequignat.function.SimpleThreadRunnable> BEGIN

15:15:21.269 [main] DEBUG eu.pequignat.MyFunction - eu.pequignat.function.SimpleThreadRunnable> The code continue

15:15:21.269 [Thread-0] DEBUG eu.pequignat.MyFunction - eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@19622763.run()

15:15:21.269 [main] DEBUG eu.pequignat.MyFunction - eu.pequignat.function.SimpleThreadRunnable> Waiting

15:15:21.270 [Thread-0] DEBUG eu.pequignat.MyFunction - eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@19622763> loop size5

15:15:21.270 [Thread-0] DEBUG eu.pequignat.MyFunction - eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@19622763> loop:0

15:15:21.378 [Thread-0] DEBUG eu.pequignat.MyFunction - eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@19622763> loop:1

15:15:21.488 [Thread-0] DEBUG eu.pequignat.MyFunction - eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@19622763> loop:2

15:15:21.599 [Thread-0] DEBUG eu.pequignat.MyFunction - eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@19622763> loop:3
```

```
15:15:21.707 [Thread-0] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@19622763>
loop:4

15:15:21.819 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.SimpleThreadRunnable> Finished

15:15:21.819 [main] INFO  eu.pequignat.App -
eu.pequignat.function.SimpleThreadRunnable> END in ms: 551

15:15:21.819 [main] INFO  eu.pequignat.App -
eu.pequignat.function.SimpleThreadRunnable> ----------------------------------
-----------
```

III.II. *Lancement de plusieurs Thread dont la détection de la fin est gérée manuellement*

III.II.1 Principe

Nous allons produire l'exemple du lancement de 100 Threads en parallèle dont la détection de la fin dans le programme principale est gérée manuellement. Nous verrons dans l'exemple d'après comment faire mieux.

III.II.2 Codes

```
Queue<Thread> allTreads = new ConcurrentLinkedQueue<>();
message("Prepare the List of all Thread to start in parallel");
for (int i = 0; i< NB_THREAD_IN_PARALLEL; i++){
    allTreads.add(new Thread(new MyRunnable()));
}
message("start all Thread");
for (Thread thread : allTreads){
    thread.start();
}
boolean isAllFinished = false;
message("How to know when finished. Bad usage because take resources");
while(! isAllFinished){
    Queue<Thread> tmpQueue = new ConcurrentLinkedQueue<>(allTreads);
    isAllFinished=true;
    Iterator<Thread> it = tmpQueue.iterator();
    while (it.hasNext()){
        Thread t = it.next();
        if (t.isAlive()){
            isAllFinished=false;
        }else{
            allTreads.remove(t);
        }
    }
}
```

III.II.3 Exemple à l'exécution

```
15:15:21.820 [main] INFO  eu.pequignat.App - Main> All tests below have the
number of functions: 100

15:15:21.821 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiThreadingManuallyManaged> BEGIN

15:15:21.822 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingManuallyManaged> Prepare the List of all
Thread to start in parallel

15:15:21.823 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingManuallyManaged> start all Thread

15:15:21.824 [Thread-1] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@df5d2d.run()

15:15:21.824 [Thread-3] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable>
eu.pequignat.runnable.MyRunnable@36cf7f06.run()

15:15:21.824 [Thread-2] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable>
eu.pequignat.runnable.MyRunnable@5a023565.run()

15:15:21.824 [Thread-1] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@df5d2d> loop
size5

[…]

15:15:21.845 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingManuallyManaged> How to know when finished.
Bad usage because take resources

[…]

15:15:22.302 [Thread-87] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@3c805de4>
loop:4

15:15:22.410 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiThreadingManuallyManaged> END in ms: 588

15:15:22.411 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiThreadingManuallyManaged> ----------------------------
--------------------
```

Nous notons que le temps global est de 588 ms alors que pour un unique Thread c'était de 551 ms. Nous avons donc lancé 100 fois plus de codes pour sensiblement le même temps machine.

III.III. *Lancement de plusieurs Threads orchestrée par un ExecutorService*

III.III.1 Principe

Depuis Java 1.5, il est possible de gérer le Pool de Thread par un ExecutorService. C'est lui qui se charge de gérer la création des Threads qui peut être couteuse ainsi que les lancers et d'attendre la fin de leur exécution.

Toutefois, cette ExecutorService a besoin d'être arrêté manuellement pour que notre programme se termine, sinon nous restons sur un programme permanent.

L'instanciation d'un ExecutorService passe par :
Executors.newFixedThreadPool(NB_THREAD_IN_PARALLEL);

Ici nous devons implémenter non plus un Runnable mais un Callable qui permet de renvoyer un Objet T qui est accessible en fin de traitement dans le Future<T>.

III.III.2 Codes

```
ExecutorService executorService =
Executors.newFixedThreadPool(NB_THREAD_IN_PARALLEL);
Queue<Callable<String>> allCallableQueue = new ConcurrentLinkedQueue<>();

message("Prepare the List of all Thread to start in parallel");
for (int i = 0; i< NB_THREAD_IN_PARALLEL; i++){
    allCallableQueue.add(new MyCallable());
}
message("start all Thread");
try {
    List<Future<String>> futures =  executorService.invokeAll(allCallableQueue);
    message("All are terminated");
    for (Future<String> f : futures){
        try {
            message(this.toString().concat(": ").concat(f.get()));
        } catch (ExecutionException e) {
            LOG.error(e.getLocalizedMessage(),e);
        }
    }
    message("We need to shutdown the ExecutorService when all Futures are
Done");
    executorService.shutdown();

} catch (InterruptedException e) {
    LOG.error(e.getLocalizedMessage(),e);
}
```

```
package eu.pequignat.runnable;

import eu.pequignat.MyFunction;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

```java
import java.util.concurrent.Callable;

import static eu.pequignat.constant.Constant.NB_LOOP;
import static eu.pequignat.constant.Constant.SLEEP_MS;

public class MyCallable implements Callable<String> {

    protected static Logger LOG = LoggerFactory.getLogger(MyFunction.class);

    public String getName(){
        return this.getClass().getName();
    }

    private void message(String message){
        LOG.debug(String.format("%s> %s", getName() ,message));
    }

    @Override
    public String call()  {
        final String refMemory= this.toString();
        message(refMemory.concat(".run()"));
        message(refMemory.concat("> loop ").concat(String.valueOf(NB_LOOP)));
        for (int i = 0; i< NB_LOOP; i++){
            message(String.format("%s> loop:%d", refMemory, i));
            try {
                Thread.sleep(SLEEP_MS);
            } catch (InterruptedException e) {
                LOG.error(e.getLocalizedMessage(),e);
            }
        }
        return "OK";
    }
}
```

III.III.3 Exemple à l'exécution

```
15:15:22.412 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiThreadingFixedPoolExecutor> BEGIN

15:15:22.413 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingFixedPoolExecutor> Prepare the List of all
Thread to start in parallel

15:15:22.414 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingFixedPoolExecutor> start all Thread

15:15:22.416 [pool-1-thread-1] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyCallable>
eu.pequignat.runnable.MyCallable@44bc7990.run()

15:15:22.416 [pool-1-thread-2] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyCallable> eu.pequignat.runnable.MyCallable@49f4bef.run()

[…]
```

```
15:15:22.864 [pool-1-thread-25] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyCallable> eu.pequignat.runnable.MyCallable@66f1c840>
loop:4

15:15:22.972 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingFixedPoolExecutor> All are terminated

15:15:22.973 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingFixedPoolExecutor>
eu.pequignat.function.MultiThreadingFixedPoolExecutor@1198b989: OK

15:15:22.973 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingFixedPoolExecutor>
eu.pequignat.function.MultiThreadingFixedPoolExecutor@1198b989: OK

[…]

15:15:22.977 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingFixedPoolExecutor>
eu.pequignat.function.MultiThreadingFixedPoolExecutor@1198b989: OK

15:15:22.977 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiThreadingFixedPoolExecutor> We need to shutdown the
ExecutorService when all Futures are Done

15:15:22.979 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiThreadingFixedPoolExecutor> END in ms: 567

15:15:22.980 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiThreadingFixedPoolExecutor> ------------------------
----------------------
```

Encore ici nous avons maintenant un temps global pour le même périmètre de code exécuté de 567 ms ce qui est légèrement plus rapide que la fois précédente de 588 ms. Avec du code plus simple et la possibilité d'avoir un retour sur chacun des Threads.

III.IV. *MutiThreading en passant par les Stream parallel*

III.IV.1 **Idée**

L'idée avec la mise en place de Stream en Java 8, est d'exploité leur mécanisme de Parallélisme afin d'effectuer les opérations en parallèle.

Avertissement : nous observons que nous n'avons pas le choix du nombre de parallelism dans la fonction Stream.parallel(). Aussi nous observerons que le nombre de parallélisme est Max(1, NB CPU -1).

III.IV.2 Codes

```
Queue<Runnable> allFunctions = new ConcurrentLinkedQueue<>();
message("Prepare the List of all Thread to start in parallel");
for (int i = 0; i< NB_THREAD_IN_PARALLEL; i++){
    allFunctions.add(new MyRunnable());
}
message("start all in parallel");
allFunctions.parallelStream().forEach(funct->{
    funct.run();
});
```

III.IV.3 Exemple à l'exécution

```
15:15:22.980 [main] INFO  eu.pequignat.App - case4MultiThreadingByStream> Use
the default number of parallel:7

15:15:22.982 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiTreadingByStream> BEGIN

15:15:22.982 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiTreadingByStream> Prepare the List of all Thread to
start in parallel

15:15:22.982 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiTreadingByStream> start all in parallel

15:15:22.985 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable>
eu.pequignat.runnable.MyRunnable@10db82ae.run()

15:15:22.985 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@10db82ae>
loop size5

15:15:22.985 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@10db82ae>
loop:0

15:15:22.985 [ForkJoinPool.commonPool-worker-1] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable>
eu.pequignat.runnable.MyRunnable@109cfa64.run()

15:15:22.985 [ForkJoinPool.commonPool-worker-1] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@109cfa64>
loop size5

[…]

15:15:33.254 [ForkJoinPool.commonPool-worker-4] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyRunnable> eu.pequignat.runnable.MyRunnable@3e124e74>
loop:4

15:15:33.365 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiTreadingByStream> END in ms: 10383
```

```
15:15:33.365 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiTreadingByStream> ----------------------------------
------------
```

On se rend bien compte que le temps est bien plus long que les 567 ms d'avant avec ici un temps global de 10 383 ms.

III.V. *MultiThreading en passant par le ForkJoinPool*

III.V.1 Idée

On veut pouvoir maitriser le nombre de parallélisme tout en ayant l'attente de fin des Thread en automatique comme pour l'ExecutorService (Java 1.5) sans à avoir faire explicitement un Shutdown de ce service en fin de résultats.

Pour cela nous allons utiliser la classe ForkJoinPool apparu avec Java 1.7.

III.V.2 Codes

```
Queue<MyCallable> allFunctions = new ConcurrentLinkedQueue<>();
message("Prepare the List of all Thread to start in parallel");
for (int i = 0; i< NB_THREAD_IN_PARALLEL; i++){
    allFunctions.add(new MyCallable());
}
ForkJoinPool forkJoinPool = new ForkJoinPool(parallelism);
message("run Callable in parallel:
".concat(String.valueOf(forkJoinPool.getParallelism())));
try {
    List<Future<String>> futures =  forkJoinPool.invokeAll(allFunctions);
    message("All are terminated");
    for (Future<String> f : futures){
        try {
            message(this.toString().concat(": ").concat(f.get()));
        } catch (ExecutionException e) {
            LOG.error(e.getLocalizedMessage(),e);
        }
    }

} catch (InterruptedException e) {
    LOG.error(e.getLocalizedMessage(),e);
}
```

III.V.3 Exemple à l'exécution

```
15:15:33.365 [main] INFO  eu.pequignat.App - case5MultiThreadingByForkJoin> Use
the number of parallel:100

15:15:33.366 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiTreadingByForkJoinPool> BEGIN

15:15:33.366 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiTreadingByForkJoinPool> Prepare the List of all
Thread to start in parallel

15:15:33.366 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiTreadingByForkJoinPool> run Callable in parallel: 100

15:15:33.368 [ForkJoinPool-1-worker-1] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyCallable>
eu.pequignat.runnable.MyCallable@74a95a58.run()

15:15:33.368 [ForkJoinPool-1-worker-1] DEBUG eu.pequignat.MyFunction -
eu.pequignat.runnable.MyCallable> eu.pequignat.runnable.MyCallable@74a95a58>
loop 5

[…]

15:15:33.960 [main] DEBUG eu.pequignat.MyFunction -
eu.pequignat.function.MultiTreadingByForkJoinPool>
eu.pequignat.function.MultiTreadingByForkJoinPool@7fc2413d: OK

15:15:33.960 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiTreadingByForkJoinPool> END in ms: 594

15:15:33.960 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiTreadingByForkJoinPool> ----------------------------
------------------
```

Ici on revient sur un temps de 594 ms semblable à l'ExecutorService mais avec du code en moins et plus maintenable donc.

IV.Problème Producteur/Consommateur

IV.I. *Principe*

Le principe est maintenant d'exploiter que Java ai une mémoire partager afin d'avoir un Thread qui produit des donner dans une file de taille limité à une constante (2), et un consommateur (un autre Thread) qui consomme les données produites dans l'ordre d'arrivée du producteur.

Si la file est pleine, le producteur est en pause.
Si la ville est vide, le consommateur est en attente de stock.

La difficulté est alors de comment communiquer le remplissage de la file pour notifier le consommateur qu'il peut consommer et vice versa, bloquer le producteur si la file est pleine.

IV.II. *Solution*

Pour cela il est nécessaire d'avoir une mémoire partagée qui est la File elle-même qui réalisera les mises en attente ainsi et les notifications.

Nous avons donc une et une seule instance de classe QueuePC qui représente la File.
Un Thread Procucer
Un Thread Consumer

Enfin nous ajoutons une fonction de Stop pour terminer la journée de travail géré avec un AtomicBoolean qui est ThreadSafe.

IV.III. *Codes*

IV.III.1 La file : QueuePC

```
package eu.pequignat.problem.pc;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.LinkedList;
import java.util.concurrent.atomic.AtomicBoolean;

import static eu.pequignat.constant.Constant.SLEEP_MS;

public class QueuePC {

    private static Logger LOG = LoggerFactory.getLogger(QueuePC.class);
    private int capacity = 2;

    // Create a list shared by producer and consumer
```

```
    // Size of list is 2.
    private LinkedList<Integer> list = new LinkedList<>();

    private final AtomicBoolean stopBool= new AtomicBoolean(Boolean.FALSE);

    // Function called by producer thread
    public void produce() throws InterruptedException
    {
        int value = 0;
        while (! stopBool.get()) {
            synchronized (this) {
                if (list.size() < capacity) {
                    LOG.info("Producer produced-" + value);
                    // to insert the jobs in the list
                    list.add(value++);

                    // notifies the consumer thread that
                    // now it can start consuming
                    notify();

                    // makes the working of program easier
                    // to  understand
                    Thread.sleep(SLEEP_MS/2);
                } else {
                    wait();
                }
            }
        }
    }

    // Function called by consumer thread
    public void consume() throws InterruptedException
    {
        while (!stopBool.get()) {
            synchronized (this) {
                if (list.size() > 0){
                    // to retrieve the first job in the list
                    int val = list.removeFirst();
                    LOG.info("Consumer consumed-" + val);

                    // Wake up producer thread
                    notify();
                    // and sleep
                    Thread.sleep(SLEEP_MS);
                }else{
                    wait();
                }
            }
        }
    }

    public void stop(){
        LOG.info("Stop called");
        stopBool.set(Boolean.TRUE);
    }

}
```

IV.III.2 Le producteur : Producer

```
package eu.pequignat.problem.pc;

import eu.pequignat.runnable.MyRunnable;

public class Producer extends MyRunnable {

    private final QueuePC queuePC;

    public Producer(QueuePC queuePC) {
        this.queuePC = queuePC;
    }

    @Override
    public void run() {
        try {
            this.queuePC.produce();
        } catch (InterruptedException e) {
            LOG.error(e.getLocalizedMessage(),e);
        }
    }
}
```

IV.III.3 Le consommateur : Consumer

```
package eu.pequignat.problem.pc;

import eu.pequignat.runnable.MyRunnable;

public class Consumer extends MyRunnable {

    private final QueuePC queuePC;

    public Consumer(QueuePC queuePC) {
        this.queuePC = queuePC;
    }

    @Override
    public void run() {
        try {
            this.queuePC.consume();
        } catch (InterruptedException e) {
            LOG.error(e.getLocalizedMessage(),e);
        }
    }
}
```

IV.IV. *Exemple à l'exécution*

```
15:16:01.233 [main] INFO  eu.pequignat.App -
eu.pequignat.function.MultiTreadingByForkJoinPool> ----------------------------
-----------------

15:16:01.233 [main] INFO  eu.pequignat.App - case6ConsumerProducerProblem> The
classic demo of resolution of problem Consumer/Producer

15:16:01.234 [main] INFO  eu.pequignat.App -
eu.pequignat.App$$Lambda$132/0x0000017fac0a51f0> BEGIN

15:16:01.237 [main] INFO  eu.pequignat.App - SLEEP

15:16:01.238 [Thread-102] INFO  eu.pequignat.problem.pc.QueuePC - Producer
produced-0

15:16:01.292 [Thread-102] INFO  eu.pequignat.problem.pc.QueuePC - Producer
produced-1

15:16:01.338 [main] INFO  eu.pequignat.App - SLEEP

15:16:01.354 [Thread-101] INFO  eu.pequignat.problem.pc.QueuePC - Consumer
consumed-0

15:16:01.447 [main] INFO  eu.pequignat.App - SLEEP

15:16:01.463 [Thread-101] INFO  eu.pequignat.problem.pc.QueuePC - Consumer
consumed-1

15:16:01.557 [main] INFO  eu.pequignat.App - SLEEP

15:16:01.573 [Thread-102] INFO  eu.pequignat.problem.pc.QueuePC - Producer
produced-2

15:16:01.635 [Thread-102] INFO  eu.pequignat.problem.pc.QueuePC - Producer
produced-3

15:16:01.667 [main] INFO  eu.pequignat.App - SLEEP

15:16:01.698 [Thread-101] INFO  eu.pequignat.problem.pc.QueuePC - Consumer
consumed-2

15:16:01.777 [main] INFO  eu.pequignat.App - STOP

15:16:01.777 [main] INFO  eu.pequignat.problem.pc.QueuePC - Stop called

15:16:01.777 [main] INFO  eu.pequignat.App - How to know when finished

15:16:01.808 [main] INFO  eu.pequignat.App -
eu.pequignat.App$$Lambda$132/0x0000017fac0a51f0> END in ms: 573

15:16:01.808 [main] INFO  eu.pequignat.App -
eu.pequignat.App$$Lambda$132/0x0000017fac0a51f0> ----------------------------
----------------
```

V. Fin du document