

Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique

*** * ***

Université de Carthage

*** * ***

**Institut National des Sciences
Appliquées et de Technologie**



Compte Rendu N°2

Programmation parallèle avec JAVA

Réalisé par :

Yasmine MHIRI

Amal MTIBAA

Mohamed Wassim RIAHI

GL4 – Groupe1

Année Universitaire : 2018/2019

La configuration logicielle et matérielle de la machine utilisée pour cet exercice:

- Processeur : Inter(R) Core(™) i7
- RAM : 6G
- SE : Windows 10 64bits
- IDE : Eclipse java photon

Exercice 1 :

- 1- Pour tester le mode « Fair » et « Unfair » nous allons exécuter la main de l'exercice1 qui utilise le thread Job pour imprimer les documents.

Une instance de Lock est là pour contrôler l'accès à une variable partagée par plusieurs threads à la fois. Une variable contrôlée par une instance de Lock ne peut être modifiée que par un seul thread à la fois, celui qui possède ce verrou.

- L'exécution de du mode unfair :

```
Running example with fair-mode = false
Thread 0: Going to print a document
Thread 0: PrintQueue: Printing a Job during 1 seconds
Thread 1: Going to print a document
Thread 2: Going to print a document
Thread 3: Going to print a document
Thread 4: Going to print a document
Thread 5: Going to print a document
Thread 6: Going to print a document
Thread 7: Going to print a document
Thread 8: Going to print a document
Thread 9: Going to print a document
Thread 0: PrintQueue: Printing a Job during 8 seconds
Thread 0: The document has been printed
Thread 1: PrintQueue: Printing a Job during 0 seconds
Thread 1: PrintQueue: Printing a Job during 8 seconds
Thread 1: The document has been printed
Thread 2: PrintQueue: Printing a Job during 9 seconds
Thread 2: PrintQueue: Printing a Job during 2 seconds
Thread 2: The document has been printed
Thread 3: PrintQueue: Printing a Job during 8 seconds
Thread 4: PrintQueue: Printing a Job during 2 seconds
Thread 4: PrintQueue: Printing a Job during 8 seconds
Thread 4: The document has been printed
Thread 5: PrintQueue: Printing a Job during 6 seconds
Thread 5: PrintQueue: Printing a Job during 5 seconds
Thread 5: The document has been printed
Thread 6: PrintQueue: Printing a Job during 1 seconds
Thread 7: PrintQueue: Printing a Job during 4 seconds
Thread 8: PrintQueue: Printing a Job during 7 seconds
Thread 8: PrintQueue: Printing a Job during 9 seconds
Thread 8: The document has been printed
Thread 9: PrintQueue: Printing a Job during 8 seconds
Thread 3: PrintQueue: Printing a Job during 6 seconds
Thread 3: The document has been printed
Thread 6: PrintQueue: Printing a Job during 4 seconds
Thread 6: The document has been printed
Thread 7: PrintQueue: Printing a Job during 3 seconds
Thread 7: The document has been printed
Thread 9: PrintQueue: Printing a Job during 4 seconds
Thread 9: The document has been printed
```

➔ On remarque que l'ordre des threads est aléatoire.

- L'exécution de du mode fair :

```
Running example with fair-mode = 'true'
Thread 0: Going to print a document
Thread 0: PrintQueue: Printing a Job during 6 seconds
Thread 1: Going to print a document
Thread 2: Going to print a document
Thread 3: Going to print a document
Thread 4: Going to print a document
Thread 5: Going to print a document
Thread 6: Going to print a document
Thread 7: Going to print a document
Thread 8: Going to print a document
Thread 9: Going to print a document
Thread 1: PrintQueue: Printing a Job during 9 seconds
Thread 2: PrintQueue: Printing a Job during 5 seconds
Thread 3: PrintQueue: Printing a Job during 7 seconds
Thread 4: PrintQueue: Printing a Job during 7 seconds
Thread 5: PrintQueue: Printing a Job during 0 seconds
Thread 6: PrintQueue: Printing a Job during 7 seconds
Thread 7: PrintQueue: Printing a Job during 3 seconds
Thread 8: PrintQueue: Printing a Job during 6 seconds
Thread 9: PrintQueue: Printing a Job during 8 seconds
Thread 0: PrintQueue: Printing a Job during 2 seconds
Thread 0: The document has been printed
Thread 1: PrintQueue: Printing a Job during 0 seconds
Thread 1: The document has been printed
Thread 2: PrintQueue: Printing a Job during 9 seconds
Thread 3: PrintQueue: Printing a Job during 2 seconds
Thread 2: The document has been printed
Thread 3: The document has been printed
Thread 4: PrintQueue: Printing a Job during 5 seconds
Thread 4: The document has been printed
Thread 5: PrintQueue: Printing a Job during 3 seconds
Thread 5: The document has been printed
Thread 6: PrintQueue: Printing a Job during 0 seconds
Thread 6: The document has been printed
Thread 7: PrintQueue: Printing a Job during 8 seconds
Thread 7: The document has been printed
Thread 8: PrintQueue: Printing a Job during 5 seconds
Thread 8: The document has been printed
Thread 9: PrintQueue: Printing a Job during 3 seconds
Thread 9: The document has been printed
```

➔ On remarque que les threads sont exécutés selon une file d'attente FIFO.

- 2- La variable partagée par tous les threads de type Job déclarés dans la main est « PrintQueue ».
- L'utilisation des deux méthodes lock() et unlock() :
 - lock() : tente d'acquies ce verrou. Cette méthode rend la main quand le verrou a été acquis. Donc si un autre thread possède ce verrou, le thread courant attendra que cet autre thread le rende.
 - unlock() : libère ce verrou. Il faut donc appeler cette méthode dans une clause finally.

- 3- Le verrou équitable " Fair lock " indique que l'ordre dans lequel le thread acquiert le verrou est alloué en fonction de l'ordre du verrou de thread, c'est-à-dire en utilisant la file d'attente FIFO, premier arrivé premier servi.
- Le verrouillage non équitable " unfair lock " est une sorte de mécanisme de préemption pour l'acquisition de verrous. C'est une acquisition aléatoire de verrous. Contrairement à l'autre type de lock, cette méthode peut empêcher certains threads d'obtenir des verrous. Le résultat est aussi non équitable.

Exercice 2 :

- La création d'une conversation entre un journaliste et une personne qui sont 2 threads héritant de la classe Thread. Notre solution contient 4 classes qui sont les suivantes :

- La classe Main : contient la méthode principale qui instancie et lance les threads, ainsi elle contient la déclaration des verrous et ses conditions.

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

public class Main {
    public static void main(String[] main) throws InterruptedException{
        Lock lock = new ReentrantLock();
        Condition question = lock.newCondition();
        Condition reponse = lock.newCondition();
        Scheduler s = new Scheduler(lock, question, reponse);
        Personne personne = new Personne("personne",s);
        Journaliste journaliste = new Journaliste("journaliste",s);
        journaliste.start();
        personne.start();
    }
}
```

- La classe Personne : étend la classe Thread et contient la méthode reponse() qui appelle à son tour la méthode repondreQ(nom) de la classe Scheduler.

```
public class Personne extends Thread {
    private String nom;
    Scheduler s;

    public Personne(String nom,Scheduler s){
        this.nom = nom;
        this.s=s;
    }

    public void reponse(){
        try{
            s.repondreQ(nom);
        }catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void run(){
        while(true){
            reponse();
        }
    }
}
```

- La classe Journaliste : étend la classe Thread et contient la méthode question() qui appelle à son tour la méthode poserQ(nom) de la classe Scheduler.

```
public class Journaliste extends Thread{
    private String nom;
    Scheduler s;

    public Journaliste(String nom, Scheduler s){
        this.nom = nom;
        this.s=s;
    }

    public void question(){
        try{
            s.poserQ(nom);
        }catch (InterruptedException e) {
            e.printStackTrace();}
    }

    public void run(){
        while(true){
            question();
        }
    }
}
```

- La classe Scheduler : la classe responsable à la synchronisation et l'ordonnancement des threads. Elle contient la variable partagée "attendre" qui informe le journaliste que la personne a répondu à la question précédente ou qu'il doit poser la première question, et qui informe aussi la personne que le journaliste a posé une nouvelle question.

Cette classe contient les deux méthodes :

- poserQ(nom) qui est responsable à la synchronisation des questions du journaliste et contient verrou.lock() et verrou.unlock() pour protéger la section critique.

```
public void poserQ(String nom) throws InterruptedException
{
    while (true)
    {
        verrou.lock();
        try{
            while (attendre==true)
                question.await();
            System.out.println("La question de "+nom+" : ");
            sc.nextLine();
            attendre = true;
            reponse.signalAll();

        }catch (InterruptedException e) {
            e.printStackTrace();
        }finally{
            verrou.unlock();
        }
    }
}
```

- `repondreQ(nom)` qui est responsable à la synchronisation des réponses de la personne et contient `verrou.lock()` et `verrou.unlock()` pour protéger la section critique.

```
public void repondreQ(String nom) throws InterruptedException
{
    while (true)
    {
        verrou.lock();
        try{
            while (attendre==false)
                reponse.await();
            System.out.println("La reponse de "+nom+" : ");
            sc.nextLine();
            attendre = false;
            question.signalAll();
        }catch (InterruptedException e) {
            e.printStackTrace();
        }finally{
            verrou.unlock();
        }
    }
}
```

- Exemple de l'exécution :

```
La question de journaliste :
question1 ?
La reponse de personne :
reponse1.
La question de journaliste :
question2 ?
La reponse de personne :
reponse2.
La question de journaliste :
question3 ?
La reponse de personne :
reponse3.
La question de journaliste :
```

Exercice 3 :

- Dans cet exercice nous allons utiliser la classe Semaphore de java pour la mise en place du système informatique surveillant les entrées et les sorties de véhicules d'un parking. Donc nous avons créé deux classes :
 - Car : qui étend la classe Thread. Donc chaque voiture correspond à un thread ayant un nom "name", une place dans le parking "place", la durée qu'elle va rester dans le parking "sleeping".
 - CarParkControl : la classe qui contrôle le flux entrant et sortant des voitures et qui contient la méthode principale main. Cette classe ayant les attributs suivants ; spaces pour les places vides dans le parking, capacity pour la capacité ou le nombre de places dans le parking, cars[] un tableau de voitures qui sont dans le parking et l'attribut sem qui correspond à la sémaphore qui permet la synchronisation des threads.

Les 2 méthodes arrive et depart utilisent le mot-clé “synchronized” pour protéger les sections critiques qui contiennent la variable partagée spaces ainsi que le tableau de voitures cars.

- La méthode arrive(Car car):

```
void arrive(Car car) throws InterruptedException {
    try {
        int verif;
        sem.acquire();
        synchronized(this) {
            verif = reserver();
            car.setPlace(verif);
            cars[verif] = car;
            spaces--;
            System.out.println(car.getName() + " entrée...");
        }
        Thread.sleep(cars[verif].getSleeping());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

- La méthode depart(Car car):

```
void depart(Car car) throws InterruptedException {
    synchronized(this) {
        int pos = car.getPlace();
        cars[pos] = null;
        spaces++;
        System.out.println(car.getName() + " sortie...");
    }
    sem.release();
}
```

- Un exemple de l'exécution :

```
public static void main(String args[]) {
    CarParkControl carpark = new CarParkControl(4);
    Thread thread[] = new Thread[10];
    for (int i=0; i<10; i++)
        thread[i] = new Car(i, carpark);
    for (int i=0; i<10; i++)
        thread[i].start();
}
```

- Le parking contient 4 places.
- 10 voitures qui veulent entrer.
- Chaque voiture reste dans le parking une durée calculée aléatoirement en utilisant la classe Random.

```

Car-0 Starting...
Car-1 Starting...
Car-2 Starting...
Car-1 entrée...
Car-2 entrée...
Car-0 entrée...
Car-4 Starting...
Car-4 entrée...
Car-6 Starting...
Car-9 Starting...
Car-3 Starting...
Car-5 Starting...
Car-1 sortie...
Car-4 sortie...
Car-6 entrée...
Car-9 entrée...
Car-8 Starting...
Car-7 Starting...
Car-6 sortie...
Car-9 sortie...
Car-3 entrée...
Car-5 entrée...
Car-3 sortie...
Car-7 entrée...
Car-2 sortie...
Car-0 sortie...
Car-8 entrée...
Car-7 sortie...
Car-5 sortie...
Car-8 sortie...

```

Pour mieux comprendre l'exécution de notre solution voici le contenu de tableau cars à chaque entrée et sortie d'une voiture :

null	null	null	null
Car-1	Car-2	Car-0	Car-4
Car-6	Car-2	Car-0	Car-9
Car-3	Car-2	Car-0	Car-5
Car-7	Car-2	Car-0	Car-5
Car-7	Car-8	null	Car-5
null	Car-8	null	Car-5
null	Car-8	null	null
null	null	null	null

Exercice 4 :

1- Testons le fichier Trieur.java qui lance un thread de Tri en ordre croissant d'un tableau comprise entre les éléments d'indices debut et fin. Cette version utilise "join" qui force un thread d'attendre que tel autre thread soit terminé.

La méthode principale "main" :

```

5     }
6     public static void main(String[] args) {
7         int[] t = {5, 8, 3, 2, 7, 10, 1};
8         Trieur trieur = new Trieur(t);
9         trieur.start();
10        try {
11            trieur.join();
12        }
13        catch (InterruptedException e) {}
14        for (int i = 0; i < t.length; i++) {
15            System.out.print(t[i] + " ; ");
16        }
17        System.out.println();
18    }
19 }
20 }

```

Après exécution :

```

<terminated> Trieur [Java Application] C:\Prc
1 ; 2 ; 3 ; 5 ; 7 ; 8 ; 10 ;

```

-> Diviser le tableau en deux. Trier ces parties par appels récursifs et fusionner les solutions de chaque partie en faisant attention de conserver l'ordre.

2- Remplaçons maintenant le join par wait et notify. Donc pour attendre les deux threads cad les trieurs fils on va utiliser wait() à la place de join(), et pour savoir si les deux fils sont exécutés, on teste une variable nbNotify qui s'incrémente lorsque le message est envoyé au parent. Cette incrémentation doit être protégée par le mot-clé synchronized.

- wait():

```

    int milieu = debut + (fin - debut) / 2;
    Trieur2 trieur1 = new Trieur2(this, t, debut, milieu);
    Trieur2 trieur2 = new Trieur2(this, t, milieu + 1, fin);
    synchronized(this) {
        try {
            while (nbNotify < 2) {
                wait();
            }
        }
        catch (InterruptedException e) {}
    }
    triFusion(debut, fin);

```

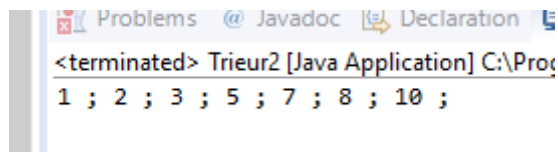
- notify():

```

    if (parent != null) {
        synchronized(parent)
        {
            parent.nbNotify++;
            parent.notifyAll();
        }
    }
}

```

- L'exécution :



```

<terminated> Trieur2 [Java Application] C:\Pro
1 ; 2 ; 3 ; 5 ; 7 ; 8 ; 10 ;

```

3- Créons maintenant une autre version en utilisant le framework « fork-join » qui résout le problème de traitement de larges quantités de données, numériques ou non. Il arrive souvent que ces quantités peuvent être traitées paquet par paquet, chaque paquet pouvant être pris en compte de façon indépendante des autres. Le traitement de chaque paquet fournit un résultat partiel. Ces résultats sont ensuite fusionnés, d'une façon ou d'une autre, dans le résultat global du traitement. Donc nous allons créer un tableau ayant un grand nombre d'entiers puis les trier en utilisant le framework « fork-join » et notre classe Trieur3 qui étend la classe RecursiveAction qui expose principalement une méthode compute() qui porte le code exécuté par cette tâche.

```

@Override
protected void compute() {
    if (fin - debut < 2) {
        trierDirectement();
    }
    else {
        int milieu = debut + (fin - debut) / 2;
        invokeAll(new Trieur3(t, debut, milieu), new Trieur3(t, milieu + 1, fin));
        triFusion(debut, fin);
    }
}

```

La fonction main :

```
public static void main(String[] args) {
    Random random = new Random();
    int nb = 10000;
    int[] t = new int[nb];
    for (int i = 0; i < nb; i++) {
        t[i] = random.nextInt(nb);
    }
    System.out.println("Before sort :");
    System.out.println(Arrays.toString(t));
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(new Trieur3(t));
    System.out.println("After sort :");
    System.out.println(Arrays.toString(t));
    /*for (int i = 0; i < nb; i++) {
        System.out.print(t[i]+" , ");
    }*/
}
```

Après exécution :

```
Before sort :
[0, 7, 19, 27, 39, 44, 20, 49, 18, 47, 27, 20, 20, 44, 25, 24, 41, 1, 3, 31, 23, 48, 1, 25, 37, 32, 46, 25, 45, 9, 40, 28, 32, 6, 24, 42, 33]
After sort :
[0, 0, 1, 1, 2, 3, 4, 6, 7, 8, 9, 9, 9, 10, 16, 18, 18, 19, 20, 20, 20, 23, 24, 24, 25, 25, 25, 27, 27, 28, 31, 32, 32, 33, 37, 39, 40, 41,
```

- La classe `ForkJoinTask<V>` expose deux méthodes principales :
- `join()` : retourne le résultat de l'exécution de cette tâche, de type `V` ;
 - `fork()` : méthode appelée pour lancer une autre tâche dans la même réserve que celle dans laquelle se trouve la tâche courante.
- Une tâche donnée ne doit pas appeler plusieurs fois sa propre méthode `fork()`. Même si ce point n'est pas vérifié par le framework, il peut s'ensuivre des problèmes de fonctionnement, notamment de visibilité des variables modifiées par cette tâche. Enfin la méthode `join()` ne rend la main qu'une fois que la tâche est exécutée, et qu'elle a produit un résultat. Cette sémantique est la même que la méthode `join()` de la classe `Thread`.