Rapport - Programmation concurrente

Elisabeth ZETTOR, L3 informatique

10 octobre 2017

Résumé

Dans ce rapport, je vais vous montrer ce que nous avons pu voir en programmation concurrente cette année, ainsi que les exercices que nous avons effectués, et notamment quelques points que j'ai pu trouver difficiles ou importants.

1 Introduction

En programmation concurrente, nous avons pu apprendre à programmer des threads, notamment en Python et en Java, dont l'intérêt était de pouvoir décomposer un programme en tâches pouvant s'effectuer simultanément. Le but des exercices que nous avons effectués était donc de nous familiariser avec cette notion, et de pouvoir utiliser les threads de manière correcte (i.e. les arrêter convenablement (thread démon ou interrompu), utiliser des verrous...).

2 Exercices

2.1 Première partie des exercices : Bases

2.1.1 Threads simples

Dans cette partie, nous devions écrire un programme qui devait lancer deux threads en fonction des paramètres que nous entrions dans la console. Cet exercice, comme son nom l'indique (2.1.1), nous a permis de créer nos premiers threads. Nous avons pu notamment voir qu'il était nécessaire de mettre les instructions propres à un thread dans une fonction *run()* qui possède un "bloc spécial" afin d'éviter les exceptions et de générer des erreurs :

```
public void run() {
try {
//instructions
} catch (InterruptedException e) { }
}
```

Et, afin de démarrer le thread, il fallait le lancer avec la méthode *start()*.
Voici l'exercice en java :

```
public class Ex1 extends Thread{
  int nb, temps;
  String message;
  public Ex1(int n, int t, String txt){
    nb = n;
    temps = t;
    message = txt;
  }
  public void run(){
    try{
      for(int i = 0; i < this.nb; i++){
        System.out.println(this.message);
        sleep(temps);
      }
    } catch (InterruptedException e) { }
  }
  public static void main(String[] args){
    if(args.length == 6){
      Ex1 th1 = new Ex1(Integer.parseInt(args[0]), Integer.parseInt(args[1]), args[2]);
      Ex1 th2 = new Ex1(Integer.parseInt(args[3]), Integer.parseInt(args[4]), args[5]);
      th1.start();
      th2.start();
    }
    else{
      System.out.println("Erreur, vous n'avez pas entre de parametres");
      System.out.println("Il faut renseigner 6 parametres de type n1, t1, txt1, n2, t2, txt2");
    }
```

2.1.2 Interruption de threads

Ici, le but était de mettre en lumière le principe d'interruption, qui permet d'interrompre un thread prématurément depuis un autre thread, étant donné qu'un thread s'achève normalement naturellement à la fin de l'exécution de sa méthode *run()*. Ici, il fallait interrompre manuellement les threads grâce à la méthode *interrupt()* et la méthode *interrupted()* qui permet de savoir si oui ou non le thread a été interrompu.

Ainsi, dans la fonction *run()*, on rajoute une boucle while, afin d'arrêter le thread lorsqu'on celui est interrompu (déclenché lorsqu'on appuie sur une touche du clavier). Ce qui donne :

```
public void run() {
try {
while(!interrupted()){
//instructions
}
} catch (InterruptedException e) { }
}
```

Voici un morceau de code en Python :

```python
from sys import argv
from threading import Thread
from time import *

class Ex2Thread(Thread):
    def __init__(self, temps, txt):
        Thread.__init__(self)
        self.temps = temps
        self.txt = txt
        self.interrupted = False
    def run(self):
        while(not(self.interrupted)):
            print(self.txt)
            sleep(self.temps)

if(len(argv) == 5):#car argv[0] = nom du programme
    th1 = Ex2Thread(float(argv[1]), argv[2])
    th2 = Ex2Thread(float(argv[3]), argv[4])
    th1.start()
    th2.start()
    input("")
    th1.interrupted = True
    input("")
    th2.interrupted = True
else:
    print("Erreur")
```

2.1.3 Threads démons

Enfin, nous avons montré l'intérêt des threads démons. Rendre un thread démon nous permet de ne pas avoir à l'arrêter manuellement, à l'inverse de l'interruption. Ainsi :

> *"Si tous les threads sont des démons, alors ces derniers sont arrêtés brutalement et le programme se termine."*

Cependant, lorsqu'on utilise des threads démons, il faut absolument utiliser la méthode *setDaemon()* avant d'utiliser la méthode *start()*, sinon on obtient des erreurs.

Voici une portion de code en Java :

```java
public void run(){
  try{
    while(true){
      System.out.println(this.message);
      sleep(this.temps);
    }
  } catch (InterruptedException e) {}
}
public static void main(String[] args){
  if(args.length == 4){
    Ex3 th1 = new Ex3(Integer.parseInt(args[0]), args[1]);
    Ex3 th2 = new Ex3(Integer.parseInt(args[2]), args[3]);
    th1.setDaemon(true);
    th2.setDaemon(true);
    th1.start();
    th2.start();
    (new Scanner(System.in)).nextLine();
  }
  else{
    System.out.println("Erreur, vous devez entrer 4 arguments");
  }
}
```

2.2 Deuxième partie des exercices : Coordination de threads - Producteurs et consommateurs

Dans cette partie, c'est la notion d'attente et de notification qui est démontrée. Grâce à cette méthode, nous pouvons éviter les problèmes d'interblocages et de famine et donc nous pouvons mettre en place le principe d'exclusion mutuelle. Autrement dit :

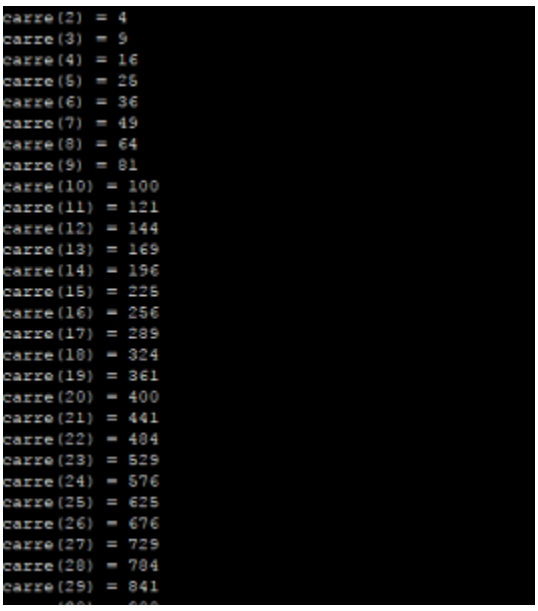
> *"L'exclusion mutuelle est la propriété qu'à tout instant deux tâches ne peuvent se trouver en section critique, i.e. au plus une tâche est en train d'exécuter une instruction de la section critique."*

Cependant, le procédé est différent selon le langage utilisé. Ainsi, en Python, on va utiliser les verrous, avec la classe *RLock()* et utilisation de l'instruction *with* ou bien, on va utiliser la classe *Condition()* qui dispose des méthodes *wait()* et *notifyAll()*, ce qui est assez similaire à Java dans ce cas-ci. En Java, on va pouvoir utiliser les méthodes synchronisées, combinées avec les méthodes *wait()* et *notityAll()*.

2.2.1 Affichage d'un nombre et de son carré

Le but de cet exercice était, comme son nom l'indique, d'afficher un nombre et son carré. Nous avons donc crée une classe Producteur (un thread) qui prenait un nombre, l'incrémentait et calculait son carré, et une autre classe Consommateur (thread). Le rôle de cette classe était d'afficher le nombre et son carré. Cependant, si l'on lance les deux threads sans utiliser d'attente et de notifications, on peut afficher le même nombre et le même carré plusieurs fois ou manquer un nombre et son carré, d'où l'utilisation des méthodes *wait()* et *notifyAll()*.

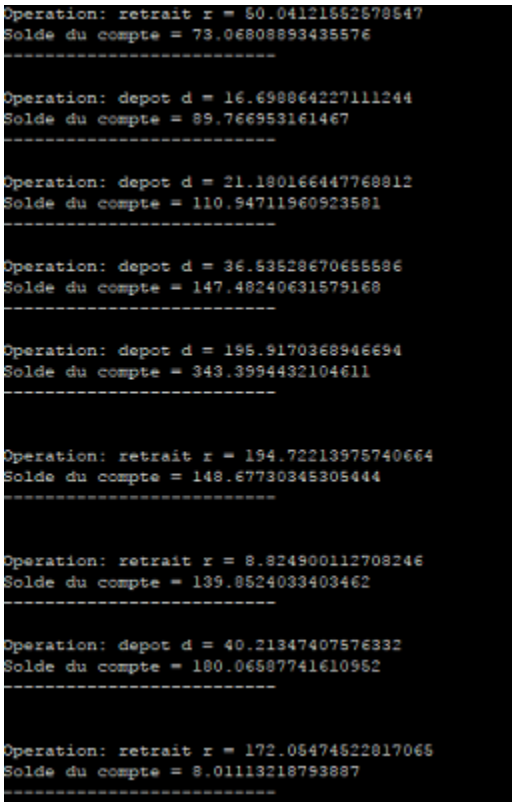
Voici une démonstration du programme dans la console :



2.2.2 Compte en banque

Ici, même principe que pour la section 2.2.1, mais avec un compte en banque, où l'on dépose et où l'on retire aléatoirement une somme, uniquement si c'est possible. Seulement, ici, on lance plusieurs threads consommateurs.

Voici une démonstration de l'exercice dans la console :



2.2.3 File d'entiers

Ici, même principe, avec une file d'entiers. On crée trois classes :

— File : qui comporte plusieurs méthodes : une pour ajouter un entier si la file n'est pas pleine, une autre pour en retirer si la file n'est pas vide, et une autre pour afficher ;

— Producteur : elle se charge d'ajouter les entiers à la file ;

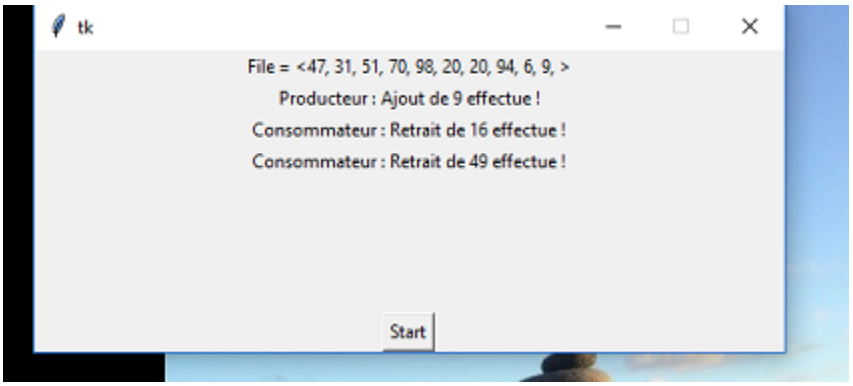
— Consommateur : elle se charge de retirer des entiers.

2.3 Troisième partie des exercices : Threads et interfaces graphiques avec Swing et Tkinter

2.3.1 Producteurs et consommateurs

Le but ici était de reprendre les exercices précédents et de les présenter avec une interface graphique en Python et en Java, en utilisant Swing [1] et Tkinter [2] (python). Ainsi, si nous reprenons l'exercice des files 2.2.3, il fallait uniquement modifier les fonctions d'affichage, afin de pouvoir visualiser le texte dans une fenêtre et associer à un bouton start l'action de lancer les threads, avec en paramètre des threads, les labels qui sont modifiés dans la fenêtre.

Voici une démonstration de l'exercice des files avec interface graphique avec tkinter :



2.3.2 Balles en mouvement

Cet exercice est une mise en pratique de tout ce que nous avons vu plus haut. Ainsi pour mener à bien ce projet (en Java), nous avons dû créer 6 classes :

- Affichage : c'est dans cette classe que nous affichons les balles et qu'on incrémente le score lorsqu'une collision se produit ;
- Balle : c'est dans cette classe qu'on crée notre objet Balle, qui a des coordonnées de position, une couleur générée aléatoirement ;
- Balles : dans cette classe, on crée une liste d'objets Balle, avec une méthode ajouteBalle() afin d'ajouter des éléments lorsqu'on appuie sur le bouton "+" ;
- Fenetre : c'est la classe qui permet de mettre en place notre interface, avec les labels correspondants au score, au temps, les boutons "start" qui permet d'enclencher le mouvement des balles, et qui se transforme en bouton stop pour mettre sur pause, le bouton "+" qui permet de rajouter des balles, et le bouton "-" qui permet d'en retirer ;
- Mouvement : la classe qui permet de mettre en mouvement les balles et faire en sorte qu'elles ricochent sur les bords ;
- Timer : la classe qui permet de lancer le timer.

3 Conlusion

Ce cours était bien utile, notamment en ce qui concerne la création d'interfaces graphiques, pour éviter que le thread principal qui gère l'interface ne soit bloqué en essayant d'effectuer plusieurs tâces à la fois.

Références

[1] Swing. https://docs.oracle.com/javase/tutorial/uiswing/index.html.

[2] Tkinter. https://docs.python.org/2/library/tkinter.html.