Programmation Concurrente : Rapport sur les travaux pratiques

BOYER Luis, L3 Informatique

2017

Résumé

La programmation concurrente est un paradigme de programmation permettant l'accomplissement de tâches simultanées grâce à des piles sémantiques appelés *processus* ou *thread*. Cela peut se révéler crucial par exemple dans la création de programmes comportant une interface graphique. En effet, séparer la partie calculatoire et la partie graphique améliore drastiquement l'expérience de l'utilisateur car l'affichage ne se retrouve pas altéré à cause des calculs que pourraient entreprendre le programme si ces deux parties étaient gérées par le même *processus* ou *thread*. Nous présenterons ci-dessous deux exemples d'utilisation de la programmation concurrente dans des programmes comportant une interface graphique.

1 Introduction

De nos jours, des millions de personnes utilisent chaque jour des applications installées sur leur téléphone, tablette ou encore ordinateurs. Pour se démarquer de la concurrence, les développeurs de même type d'applications doivent pouvoir proposer un service meilleur ou équivalent tout en offrant une expérience d'utilisation que l'on pourrait qualifier d' "agréable" . C'est ici qu'intervient l'utilisation de la programmation concurrente : créer un programme dont les méthodes respectent un ordre d'action ou allier des sous-programmes contenus dans des fichiers séparés afin de créer une application capable de maintenir un affichage "fluide" même lorsque l'application procède à des calculs très importants.

Ce rapport présentera deux programmes. Le premier, décrit dans la section 2, est la représentation graphique d'une file d'attente permettant de saisir contextuellement le principe de programmation concurrente tandis que le second, décrit dans la section 3, aura la tâche de démontrer les résultats de l'application de la programmation concurrente dans une application graphique contenant plusieurs objets à prendre en compte simultanément.

2 File d'attente

Remarque : Pour ce programme nous utiliserons le langage Python dans sa version 3. Toutes les classes que nous utiliserons seront écrites dans le même fichier. De plus, nous utiliserons la librairie *Tkinter* afin de produire un affichage du programme.

Le principe de file d'attente est celui de "premier arrivé, premier servi" que l'on applique chaque jour dans notre société. Cependant nous, humains, l'utilisons tacitement sans obligation réelle d'attendre notre tour. Or, sans le principe de la programmation concurrente, un programme comportant deux méthodes utilisant la même variable accèderaient à cette variable et la modifieraient en même temps. C'est afin d'empêcher cela que nous utilisons la programmation concurrente.

Par exemple, prenons un producteur et deux consommateurs. Le premier consommateur demandant un service serait celui qui serait servi en premier et le second attendrait la fin de ce service pour faire sa demande à son tour.

Dans cet exemple, supposons que le producteur produise un nombre fini de nombres stockés dans une liste et que les consommateurs viennent récupérer à chaque fois le premier nombre de cette liste.

Premièrement, afin de représenter le producteur et les consommateurs nous créons une classe Producteurs et une classe Consommateurs comme ci-dessous :

```
class Producteurs(Thread):
    def __init__(self, temps, file):
        Thread.__init__(self)
        self.temps = int(temps)
        self.file = file
        self.label = Label(fenetre, text="")
        self.label.pack()
        self.daemon = True

    def run(self):
        while(1):
            self.file.deposer(self)
            time.sleep(self.temps)
```

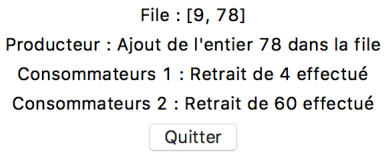
```
class Consommateurs(Thread):
    def __init__(self, temps, file, n):
        Thread.__init__(self)
        self.temps = int(temps)
        self.file = file
        self.n = n
        self.label = Label(fenetre, text="")
        self.label.pack()
        self.daemon = True

    def run(self):
        while(1):
            self.file.retirer(self)
            time.sleep(self.temps)
```

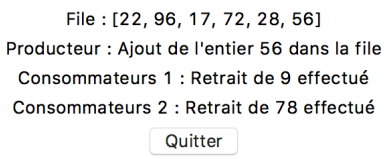
On peut remarquer que ces classes sont des *Thread* qui sont créées avec des paramètres *temps* et *file* afin qu'ils puissent se partager la même liste de nombres et puissent s'endormir *temps* secondes. Leur valeur *daemon* est mise à *True* pour que ces *Thread* s'arrêtent à la fermeture de la fenêtre. Elles comportent aussi un *label* et l'affichage de ce dernier dans la fenêtre. Pour la classe Consommateurs on lui ajoute aussi un attribut *n* pour numéroter les consommateurs.

Ces deux classes comportent chacune une méthode *run* propre aux *Thread* qui existe afin d'indiquer au *Thread* en quoi consiste sa tâche. On voit ici qu'il s'agit de *deposer* pour Producteurs et *retirer* pour Consommateurs. Ces méthodes sont décrites dans la classe File ci-dessous qui crée l'objet *file* que se partagent Producteurs et Consommateurs :

```
class File():
    def __init__(self):
        self.file = []
        self.Cond = Condition()

    def deposer(self,prod):
        with self.Cond :
            b = randint(0,100)
            prod.label["text"] = "Producteur : Ajout de l'entier " + str(b) + " dans la file"
            while(len(self.file)>20):
                prod.label["text"] = "Producteur : File pleine. En attente d'une place pour l'entier : " + str(b)
                self.Cond.wait()
            self.file.append(b)
            label["text"] = "File : " + str(self.file)
            self.Cond.notifyAll()

    def retirer(self,consom):
        with self.Cond :
            while(len(self.file)==0):
                consom.label["text"] = "Consommateurs "+ str(consom.n)+" : File vide. En attente de l'ajout d'un entier"
                self.Cond.wait()
            consom.label["text"] = "Consommateurs "+str(consom.n)+" : Retrait de l'entier " + str(self.file[0]) + " dans la file"
            r = self.file[0]
            self.file.remove(self.file[0])
            consom.label["text"] = "Consommateurs "+str(consom.n)+" : Retrait de " + str(r) + " effectué"
            label["text"] = "File : " + str(self.file)
            self.Cond.notifyAll()
```

L'objet *file* se définit comme une liste possédant une *Condition*. Cette dernière agit comme un *verrou* sur les deux méthodes de File. Un *verrou* est une propriété d'un objet qui rend cet objet disponible que pour une seule méthode de l'objet à la fois. On activera cette *Condition* au début de chaque méthode. Ainsi un Producteur modifiera la File qu'il partage avec des Consommateurs sans que ces derniers puissent avoir accès à cette File au même moment. Inversement, le Producteur ne pourra pas ajouter un nombre à la File si un Consommateur est en train d'en retirer un au même moment. De plus, deux Consommateurs ne pourront pas non plus retirer un nombre en même temps. Voici le rendu final du programme :





3 Balles en mouvement

Remarque : Pour ce programme nous utiliserons le langage Java. Toutes les classes que nous utiliserons seront donc écrites dans des fichiers différents. De plus, nous utiliserons une classe *Fenêtre* héritant de la classe *JFrame* afin de produire un affichage du programme.

Nous venons de voir dans la section 2 comment fonctionnaient les bases de la programmation concurrente. Mais dans l'exemple précédent nous n'avons utilisé qu'un unique objet. Cette fois nous utiliserons un nombre d'objets plus conséquents. En effet, dans cet exemple nous allons mettre en place une production, une suppression et un affichage de balles que nous mettrons en mouvement. De plus, nous allons déterminer à chaque instant si deux balles entrent en collision ou non.

3.1 Balle

La première chose à faire est la création de la classe *Balle* :

```
Color couleur;
int x,y,largeur;
double angleX, angleY;
int dx,dy;

public Balle(Color couleur, int x , int y) {
    this.x = x;
    this.y = y;
    this.largeur = 50;
    this.couleur = couleur;
    this.angleX = Math.random();
    this.angleY = Math.random();
    if(angleX < 0.50) {
        this.dx = 1;
    }
    else this.dx = -1;
    if(angleY < 0.50) {
        this.dy = 1;
    }
    else this.dy = -1;
}

public void paint(Graphics g) {
    g.setColor(couleur);
    g.fillOval(x, y, largeur, largeur);
}
```

On remarquera qu'il s'agit de donner à la classe *Balle* des attributs, avec quelques fioritures comme la direction que prendra l'objet à son départ. On lui rajoute la méthode *paint* afin de pouvoir dessiner l'objet. Les balles créées seront stockées dans un tableau de *Balle* afin de pouvoir garder une trace de chacune d'elles.

3.2 Mouvement et collision

Ci-dessous nous avons les méthodes *move* et *collision* contenues dans la classe *Panneau* qui crée le *Jpanel* dans lequel seront contenues les balles.

La méthode *move* déplace les balles dans le sens de leur attribut *dx* et *dy* . Il est important de ne pas oublier le *repaint* afin de pouvoir afficher le déplacement des balles à chaque instant.

La méthode *collision* vérifie si deux balles entrent en collision en calculant la distance entre une balle *o* et toutes les autres balles *ball* existantes. On fait cela pour chaque balle. Il s'agit donc d'un calcul important. On le place bien dans une autre classe afin de ne pas le mêler à la gestion de l'affichage graphique.

Remarque : On notera le fait que si deux balles entrent en collision alors on déplace tour à tour les deux balles en fin de tableau et on les supprime. On renvoie aussi une valeur booléenne afin

de savoir si oui ou non on doit incrémenter le score de l'utilisateur et faire un rafraîchissement de l'interface graphique

```
public void move() {
    for(Balle ball : Balles) {
        if(ball!=null) {
            if(ball.y + ball.largeur > getHeight() || ball.y < 0) ball.dy = -ball.dy;
            if(ball.x + ball.largeur > getWidth()  || ball.x < 0) ball.dx = -ball.dx;
            ball.y = ball.y - ball.dy;
            ball.x = ball.x - ball.dx;
        }
    }
    repaint();
}

public boolean collision() {
    for(Balle o : Balles){
        for(Balle ball : Balles) {
            if(o!=null && o!=ball && ball!=null) {
                int distance = (int) Math.sqrt( Math.pow( ball.x - o.x, 2 ) + Math.pow( ball.y - o.y, 2 ) );
                if(distance < ball.largeur) {
                    for(int i=0;i<Balles.length;i++) {
                        if(Balles[i]==o) {
                            for(int j=i;j<Balles.length-1;j++) {
                                Balles[j] = Balles[j+1];
                            }
                            Balles[Balles.length-1] = null;
                            nombre_balles--;
                        }
                    }
                    for(int k=0;k<Balles.length;k++) {
                        if(Balles[k]==ball) {
                            for(int l=k;l<Balles.length-1;l++) {
                                Balles[l] = Balles[l+1];
                            }
                            Balles[Balles.length-1] = null;
                            nombre_balles--;
                        }
                    }
                    return true;
                }
            }
        }
    }
    return false;
}
```

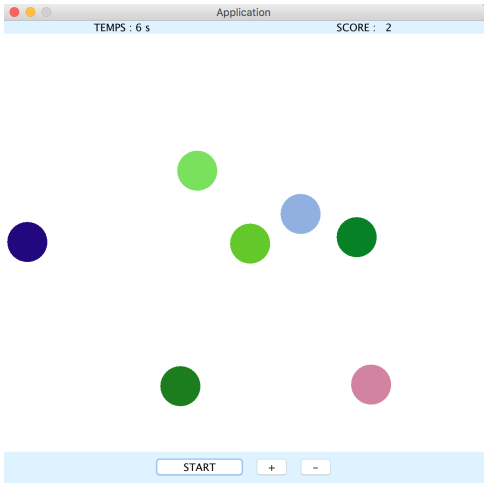
Ces méthodes, bien que contenues dans la classe *Panneau* sont exécutées par la classe *Animation* qui hérite de la classe *Thread*. Cette classe présente la même caractérisitique *daemon* que les *Thread* Producteurs et Consommateurs de l'exemple précédent. Ce *Thread Animation* s'activera d'ailleurs toutes les 10 millisecondes afin de déplacer les balles sauf si l'utilisateur a mis l'application en pause. Aussi, il mettra à jour les boutons d'ajout et de suppression de la *JFrame Fenêtre* en fonction du nombre de balles restantes. Voici ci-dessous le code de cette classe :

```
public class Animation extends Thread{

    Fenetre fen;

    public Animation(Fenetre fen){
        this.setDaemon(true);
        this.fen = fen;
    }

    public void run() {
        try {
            while(true) {
                if(!fen.pause) {
                    fen.pan.move();
                    if(fen.pan.collision()) {
                        if(fen.pan.nombre_balles>0) {
                            if(!fen.plus.isEnabled()) fen.plus.setEnabled(true);
                        }
                        if(fen.pan.nombre_balles==0) {
                            fen.moins.setEnabled(false);
                            fen.plus.setEnabled(true);
                        }
                        if(fen.pan.nombre_balles==8) {
                            fen.plus.setEnabled(false);
                        }
                        fen.points++;
                        fen.score.setText("SCORE :   " + fen.points);
                    }
                    sleep(10);
                }
                else sleep(10);
            }
        } catch (InterruptedException e) {}
    }
}
```

3.3 Fenêtre

Voici le code de la fenêtre de notre application et ce à quoi elle ressemble :

```
Horloge heure = new Horloge();
JLabel score = new JLabel("SCORE :  0 ",JLabel.CENTER);
JButton bouton = new JButton("COMMENCER");
JButton plus = new JButton(" + ");
JButton moins = new JButton(" - ");
int points;

public Fenetre() {
    super("Application");
    setSize(600,600);
    setLocationRelativeTo(null);
    setResizable(false);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container contenu = getContentPane();
    contenu.setLayout(new BorderLayout());
    contenu.add(pan, BorderLayout.CENTER);
    pan.setBackground(Color.white);
    JPanel panel = new JPanel();
    contenu.add(panel,BorderLayout.SOUTH);
    panel.setBackground(new Color(223, 242, 255));
    bouton.addActionListener(this);
    plus.addActionListener(this);
    moins.addActionListener(this);
    bouton.setPreferredSize(new Dimension(120, 30));
    plus.setPreferredSize(new Dimension(50, 30));
    moins.setPreferredSize(new Dimension(50, 30));
    panel.add(bouton);
    panel.add(plus);
    panel.add(moins);
    JPanel panel_haut = new JPanel();
    contenu.add(panel_haut,BorderLayout.NORTH);
    panel_haut.setBackground(new Color(223, 242, 255));
    panel_haut.setLayout(new GridLayout(1,2));
    heure.setText("TEMPS : 0 s");
    heure.setHorizontalAlignment(JLabel.CENTER);
    panel_haut.add(heure);
    panel_haut.add(score);
    plus.setEnabled(false);
    moins.setEnabled(false);
    setVisible(true);
}
```



C'est en utilisant plusieurs classes (*Fenêtre* , *Panneau*, *Balle*, *Animation*) et en utilisant un *Thread* que nous parvenons à séparer la partie calculatoire que représente le mouvement et la gestion de collision et l'affichage.

4 Conclusion

Nous avons pu voir grâce à nos deux exemples que le principe de la programmation concurrente apporte beaucoup à la gestion des accès aux variables ainsi qu'à la répartition des tâches dans un programme informatique. Il est donc possible de penser que dans le futur, les applications et programmes ne pourront se démarquer qu'au niveau de la gestion des ressources que feront leurs développeurs au travers de la programmation concurrente

5 Bibliographie

— OpenClassroom - La programmation parallèle avec threading : https ://openclassrooms.com/courses/apprenez-a-programmer-en-python/la-programmation-parallele-avec-threading
— Développez.com - Introduction à la programmation concurrente en Java : http ://zenika.developpez.com/tutoriels/java/core/javaprogconcurrente/