

Compte rendu - Programmation répartie

Thomas LACAZE DUT Informatique groupe A

[Github] (<https://github.com/LacazeThomas/ProgrammatioRepartie>)

01/02/2019 - Thread Java TD/TP N°1 - FINI

Objectif de la séance : Comprendre le but des threads et son application.

Un processus léger (ou Thread) peut être définie comme le fils d'un processus lourd.

Les ressources allouées à un processus vont être partagées entre les processus légers qui le composent.

Un processus possèdent au moins un thread. Le comportement les processus légers peut changer en fonction de l'OS. Pour éviter ce problème, JAVA propose son propre mécanisme de threads.

Il permet par exemple :

- La création de processus.
- La communication entre eux.
- La synchronisation à l'aide de moniteurs.

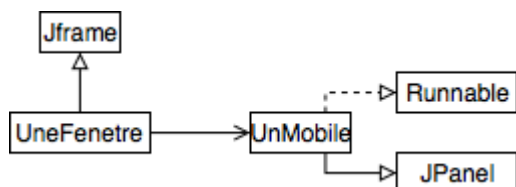


Diagramme de classe UML succinct

Voici la classe UneFenetre résumé grossièrement pour afficher l'élément de la classe UnMobile :

```
public UneFenetre(){
    super("TP01");
    int LARG=400, HAUT=75; //On définit les largeurs de l'écran
    setSize(LARG,HAUT); //On change la taille de la fenêtre
    setDefaultCloseOperation(EXIT_ON_CLOSE); //Fermeture de la fenêtre si
    clique
    setVisible(true); //On rend la fenêtre visible

    UnMobile sonMobile = new UnMobile(LARG, HAUT); //On crée un Mobile
    Thread tache = new Thread(sonMobile); //On l'affecte à un thread
    setContentPane(sonMobile) //On affiche à l'écran le Mobile
    tache.start(); //On démarre le thread
}
```

Voici la classe UnMobile résumé grossièrement qui permet de faire l'aller retour du carré :

```
public void run(){
    while (true) {
        ...
        for (sonDebDessin=saLargeur; sonDebDessin >= 0; sonDebDessin-= sonPas)
        { //Aller de droite à gauche
            repaint(); //On deplace le Mobile en rappelant 'paintComponent'
            try{
                Thread.sleep(sonTemps); //On mets en pause le thread pendant X
            }
            catch (InterruptedException telleExcp){
                telleExcp.printStackTrace();
            }
        }
    }
}
```

08/02/2019 - Thread Java TD/TP N°3 - FINI

Objectif de la séance : Comprendre le but des sémaphores et son application.

Un sémaphore permet de donner des ordres de priorité à certaines parties d'un programme lors de l'utilisation de Threads

Il existe deux types de sémaphore :

- Les sémaphores binaires (mutex) : 1 ou 0.
- Les sémaphores généraux: valeurs positives.

Un sémaphore binaire ou classique continue d'exécuter la tâche si sa valeur est positive.

```
class Exclusion{} //Exclusion est classe scope est donc commun à tous les ensembles de cette classe
static Exclusion exclusionImpression = new Exclusion (); //Classe scope
static SemaphoreBinaire sem = new SemaphoreBinaire(1);
```

Ici on crée un Semaphore binaire donc mutex et on l'initialise à 1. Pour protéger la section critique (partie du code à exécuter dans un certain sens), il suffit de décrémenter la valeur du mutex pour bloquer les tâches qui arrivent: `sem.syncWait()`

Une fois la section passée il suffit de recrémenter le mutex pour le libérer la place aux prochains threads: `sem.syncSignal()`

Il est possible d'afficher des messages d'informations lorsqu'on rentre dans une section critique ou que l'on en sort. Pour cela, lors de la décrémentation ou incrémentation un simple: `System.out.println("J'entre dans une section critique");` suffit.

08/02/2019 - TP combinaison n°1 et n°3 - FINI

Objectif de la séance : Implémenter une section critique avec semaphore dans le déplacement des objets de la classe UnMobile.

Enoncé: Diviser l'écran en 3 parties. Celle du milieu étant une section critique. Il doit y avoir uniquement N classe UnMobile présent au maximum.

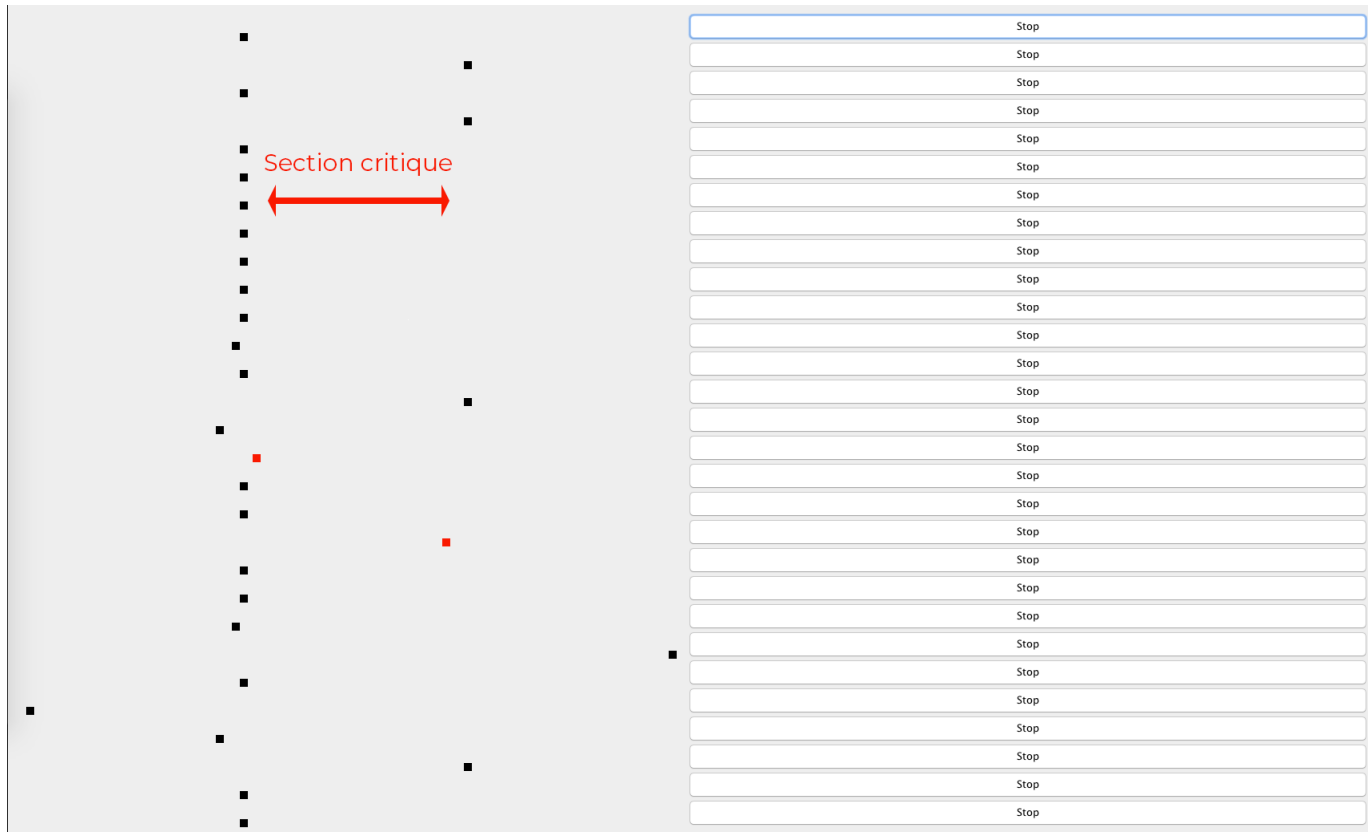


Diagramme succinct

Pour cela, il suffit de diviser les deux boucles `for` en 3 boucles chacune. Ainsi, nous créons un semaphore pour pouvoir laisser passer N classe UnMobile au milieu de l'écran. Un mutex aurait été possible si on voulait uniquement un objet de la classe UnMobile au milieu. A présent nous décrétons lors qu'un élément arrive au milieu `sem.syncWait()` et nous informons le semaphore lorsqu'il sort du milieu `sem.syncSignal()`.

Pour des raisons pratique le semaphore n'est pas une classe scope mais est définie dans UneFenetre. Il est envoyé à chaque Thread.

Voici un exemple de code succinct:

```
public UneFenetre(){
    super("TP3");
    int N = 10; //Nous voulons 10 threads pouvant passer dans la section
    critique
    Semaphore sem = new Semaphore(N);
    this.setLayout(new GridLayout(0CU,2));
    for(int i=0;i<0CU;i++){
        stop[i] = false;
        sonMobile[i] = new UnMobile(LARG, HAUT,sem);
        sonButton[i] = new JButton("Stop");
        this.getContentPane().add(sonMobile[i]);
        this.getContentPane().add(sonButton[i]);
        tache[i] = new Thread(sonMobile[i]);
        tache[i].start();
        sonButton[i].addActionListener(listener);
    }
    ...
}
```

```
public void run(){
    while (true) {
        ##ALLER
        ..

        sem.syncWait(); //On décrémente le sémaphore. Entrée
        section critique
        for (sonDebDessin= saLargeur/3; sonDebDessin <= saLargeur/3*2;
        sonDebDessin+= sonPas){
            ...
        }
        sem.syncSignal(); //On incrémente le sémaphore. Sortie section
        critique
        ...
        ##RETOUR - Même opération
        ...
    }
}
```

22/02/2019 - TP combinaison n°2 - FINI

Objectif de la séance : Illustration d'un modèle producteur consommateur simple.

TODO

Introduction/ Définition

deadlock -> attendre après des sections critiques. Bloquage. Correction: tuer le processus exclusion mutuelle(mutex)

Moniteur: lire écrire dans les sections critiques.

Concurrence et Moniteur:

Un moniteur est un objet de synchronisation. Qui permet l'exclusion mutuelle qui permet d'attendre qu'une condition soit validé

A rajouter image de code (quelques lignes) Parler de queue et de tête.

TP Fini-> I1 - I2 - I3 - Exercice II (trouvable sur git)

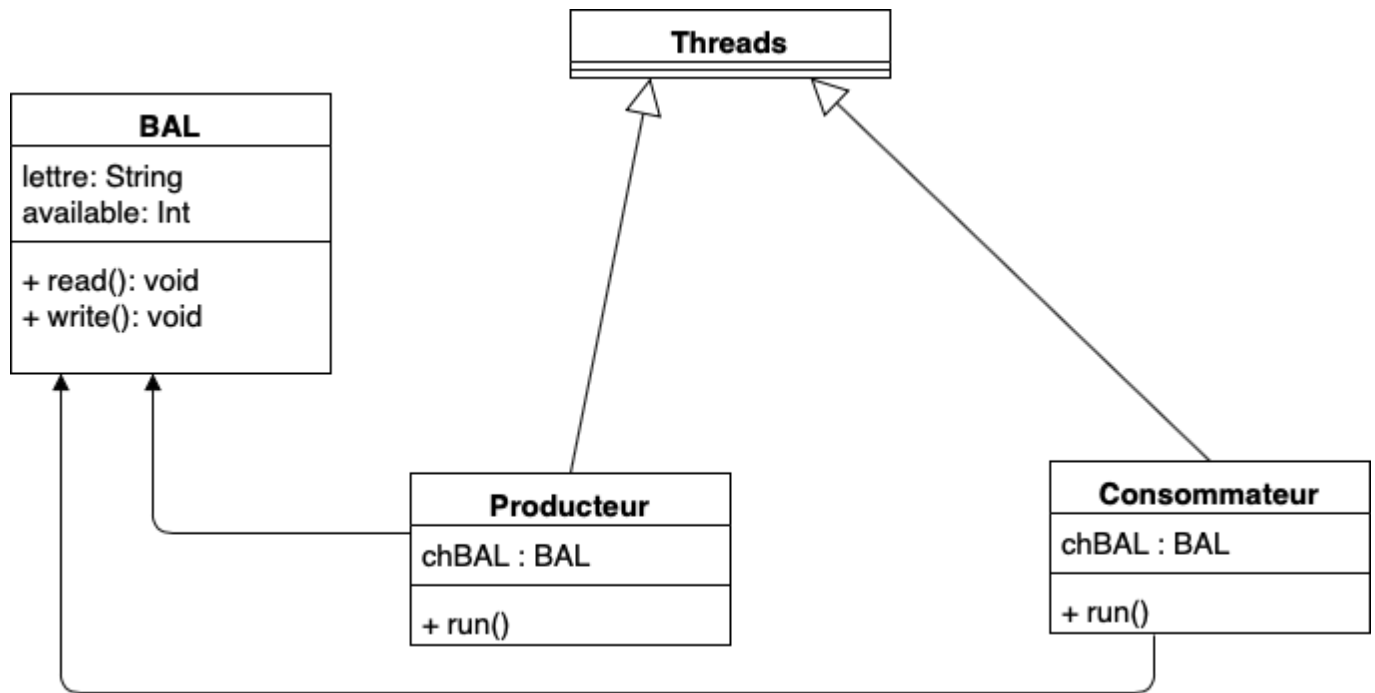


Diagramme de classe UML succinct du TP03