Rapport de Programmation Répartie

BERTHOU Evann

12/05/2022

Contents

Rapport de Programmation Répartie

Introduction

Vers 2008, on voit arriver sur le marché grand public des processeurs *multi-coeurs*. C'est-à-dire qu'un processeur est composé en réalité de plusieurs unités de calculs (appelées "coeur" qui fonctionne en simultané. Cela permet ainsi d'exécuter plusieurs opérations à un même instant, permettant d'augmenter les performances d'un processeur sans devoir créer des unités plus puissantes. Avant les processeurs étaient *mono-coeur*. Ainsi il était impossible d'exécuter plusieurs tâches réellement en même temps.

Avantages des processeurs multi-coeurs

Sur un processeur mono-coeur, une seule tâche peut être exécuter à la fois. On se retrouve donc avec des systèmes, qui malgré des systèmes d'exploitations multi-tâches, qui ont du mal à gérer plusieurs applications à la fois. Il n'était pas fluide d'avoir à la fois un navigateur internet et un logiciel de traitement de texte puisque les deux applications s'exécutent sur le même coeur. Il est tout de même possible d'avoir plusieurs applications lancées en même temps mais il n'y a qu'une illusion qu'elles sont exécutées en même temps. Les processeurs sont juste relativement assez rapide pour que les applications utilisent chacune leur tour le seul coeur disponible.

Avec les processeurs multi-coeurs, les différentes applications ne sont pas en attente qu'une autre tâche soit terminée. Dans le même exemple, le navigateur peut être exécuter sur le coeur 1, et le logiciel de traitement de texte sur le coeur 2. Les deux aplications tournent donc en simultané rendant l'utilisation plus fluide.

Chaque application est appelée un **processus lourd**. Chaque processus est isolé des autres, notamment en possédant son propre espace mémoire. Il est possible pour un processus lourd de posséder de nombreux **processus léger** ou **Thread** qui permettent une exécution en parralèle de sous-tâches dans un processus lourd.

Mais cette exécution parallèle a apporté en même temps de nouvelles difficultés, notamment pour les développeurs. Des contraintes de synchronisation ou d'accès à la mémoire sont alors apparus.

Programmation séquentielle et Programmation répartie

Lorsque l'on apprend la programmation, il est naturel d'écrire des programmes séquentiels. C'est-à-dire des programmes dont les actions sont réalisées les unes à la suite des autres. L'application possède donc un état global et une exécution prévisible.

Avec les processeurs multi-coeurs il est possible, à l'aide des threads, d'exécuter plusieurs tâches. On se retrouve alors dans un contexte de programmation répartie où l'on perd un état global puisque chaque thread possède sa propre pile, son propre contexte d'exécution et peut **partager de la mémoire** avec d'autres threads. On peut aussi retrouver le terme de **programmation concurrente (Voir API Concurrent**

Application parallèle

A l'inverse des applications séquentilles, il existe également des applications parallèles. Dans ce type d'applications, les opérations peuvent s'exécuter de manière indépendante des autres. C'est par exemple l'essence même des cartes graphiques (GPU). Contraitement à un processeur (CPU), le GPU est capable d'exécuter un seul programme avec un très large volume de données. Chaque données étant indépendantes des autres, toutes les opérations peuvent être exécutées en parallèle. Lorsque le GPU génère une image, il n'y pas besoin de le faire pixel par pixel comme le ferait un CPU. Il peut tout simplement prendre tous les pixels, appliquer le programme à tous les pixels en même temps puis afficher l'image génrée.

Néanmoins, la carte graphique ne permet pas de réaliser efficacement des tâches complexes. On peut faire la différence entre les deux types de processeur assez simplement. Un GPU est capable de gérer un volume de données conséquent mais que d'une manière, là où un CPU peut gérer un grand nombre de tâches très rapidement mais est plus limité dans le parallèlisme.

Pour donner un exemple concret, on peut prendre un programme qui additionne deux vecteurs. Chaque addition est indépendante d'une autre. On peut donc très facilement paralléliser ce programme.

Un CPU effectura les addtions les unes à la suite des autres.

```
z1 = x1 + y1
z2 = x2 + y2
z3 = x3 + y3
```

Un GPU effectura toutes les additions en même temps.

```
zn = xn + yn
```

Une exécution en parallèle est tout de même possible sur un CPU mais sera très largement plus lent à cause du nombre de coeurs disponibles. Les meilleurs CPU possède 64 coeurs (128 coeurs logiques), alors qu'un GPU moyen, par exemple la NVIDIA GTX 1060 (la carte graphique la plus populaire d'après Steam, Source) possède 1280 coeurs Source.

On remarque ainsi que les deux composants ne sont tous simplements pas fait pour réaliser le même type de tâches.

Il existe, en fonction du matériel différente, façon de traiter les données dont les plus courants sont :

- *Single Instruction Multiple Data* : Un seul jeu d'instructions pour plusieurs jeux de données. Cas des GPU.
- *Multiple Instructions Single data* : Un seul jeu de données passe à travers plusieurs jeux d'instructions.
- *Multiple Instructions Multiple Data* : Plusieurs jeux d'instructions pour plusieurs jeux de données. C'est le cas de la majorité des CPU modernes et le type de traitement dont l'on parle dans ce rapport.
- *Single Instructions Single Data* : Exécution purement séquentielle avec un seul jeu d'instructions et de données.

Systèmes répartis et applications réparties

Un système réparti est un système que l'on voit comme un seule machine mais qui possède plusieurs machines. Contrairement aux systèmes non réparties qui possède une **mémoire partagée** (mémoire présente sur une seule machine), un système réparti à sa **mémoire distribuée** sur de nombreuses machines. L'object de ses systèmes et de répartir une même tâche (généralement très coûteuse en calcul) à travers de nombreuses machines afin d'en accélerer la réalisation.

Dans le même principe d'un système réparti, une application peut aussi être dite répartie lorsqu'une application est en réalité séparée en plusieurs applications.

Principe de conception pour une application ou un système réparti

Comme dans toute avancée technologique, il faut définir des principes de conception afin de s'assurer que le programme ou l'architecture que nous réaliserons sera de bonne qualité.

5 principes généraux ont été définit dans le cadre d'un système réparti.

Pour chaque bon principe suivant, un exemple sera donné dans le cadre d'une application WEB.

Principe	Définition	Exemple
Transparence à la localisation	Ne pas faire de différence entre une machine distance ou locale	*Liens hypertexte* : on peut sans transition passer d'un site à un autre
Transparence d'accès	Utiliser une interface lorsque l'on souhaite acceder à une ressource	*URL* : On ne doit pas directement ce connecter à un IP pour accéder à un site WEB
Transparence à l'hétérogénéité	La différence de matériel ne doit pas poser problème	*HTTP* : Utiliser un protocol pour communiquer
Transparence aux pannes	Ne pas affecter l'utilisateur si une machine tombe en panne.	*Proxy* : Rediriger vers une autre machine de sauvegarde si le serveur principal est en panne
Transparence à l'extension des ressources	Une application doit pouvoir être étendu ou réduit	*Répartiteur de charge* : Rediger sur une machine, voire lancer de nouvelles instances en fonction de la charge

Processus léger ou Threads

Un **processus léger** ou **threads** est le support d'une tâche. Les threads n'exécutent en parallèle de l'application principale. Chaque thread possède sa propre pile, son propre contexte d'exécution mais peut partager sa mémoire avec d'autres threads du même processus lourd. C'est notamment ce dernier point qui peut causer de difficultés lors du développement d'applications parallélisées.

Les threads peuvent avoir de nombreuses utilités comme rendre une application plus rapide (calcul scientifique, encodage vidéo) ou réaliser des tâches en arrière-plan afin de ne pas bloquer l'utilisateur (auto-complétion des éditeurs de texte, outils d'analyse de code).

Les threads ne sont pas un concept natif aux ordinateurs, c'est donc au système d'exploitation de s'occuper de l'**ordonancement** et de sur quel coeur sera exécuté un thread.

Utilisation en Java

Les systèmes d'exploitation fournissent une API que les langages de programmation peuvent utiliser afin de manipuler des threads. Ici, nous utiliserons Java comme exemple.

Java fournis un ensemble d'outils permettant de manipuler les threads de manière assez simple. Dans cet exemple, nous allons créer un ensemble de *mobiles* se déplaçant horizontale sur l'écran. Chaque *mobile* sera exécuter sur un thread différent.

En Java, pour d'un thread exécute une tâche, il faut que l'objet implémente l'interface Runnable. Cette interface définit une méthode unique "run" qui définit le point d'entrée de la tâche. Il est ensuite possible de créer un nouveau Thread et de lui donner un Runnable en paramètre.

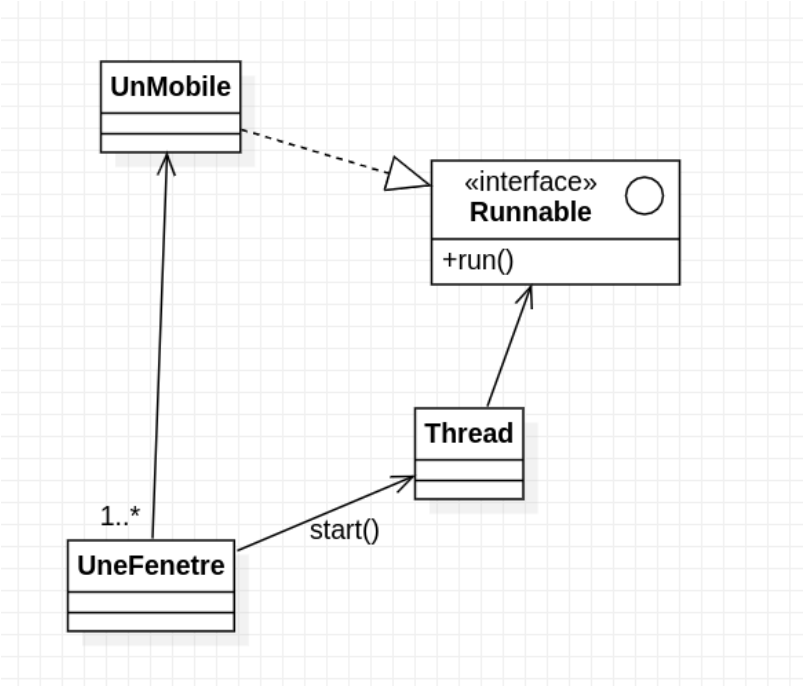

Figure 1: Diagramme de classe de l'utilisation d'un thread en Java

```
// Création d'un nouveau mobile
UnMobile mobile = new UnMobile(LARG, HAUT);

// Ajout du mobile à la fenêtre
add(mobile);

// Création d'un Thread avec mobile comme tâche
Thread thread1 = new Thread(mobile);

// Demande de lancer la tâche
thread1.start();
```

Avec seulements ses 4 lignes de code, un nouveau thread sera créer et exécutera la tâche. Le mobile apparait donc dans la fenêtre et se déplace. On peut ainsi faciclement ajouter de nombreux mobiles qui seront tous exécuter en parallèle et donc se déplacerons en même temps.

Cycle de vie d'un thread

Le cycle de vie d'un thread peut être résumé en seulement 4 étapes.

- Prêt à l'exécution : La tâche est en attente d'être lancée.
- En cours d'exécution : La tâche est en train d'être exécutée.
- En attente : La tâche est mise en pause jusqu'à qu'elle se fasse débloquer par un signal.
- Mort : La tâche est terminée.

Ainsi, lorsqu'on utilise "suspend", le Thread se met en pause jusqu'à ce que l'on appel "resume". Cela peut être utile lorsqu'on souhaite mettre en pause une action pour donner plus de priorité sur un autre Thread tout en gardant le thread pausé en vie. C'est la classe Thread qui s'occupe entièrement de la gestion de la suspension ou de la reprise. A aucun moment c'est au développeur de s'occuper de l'état du Thread.

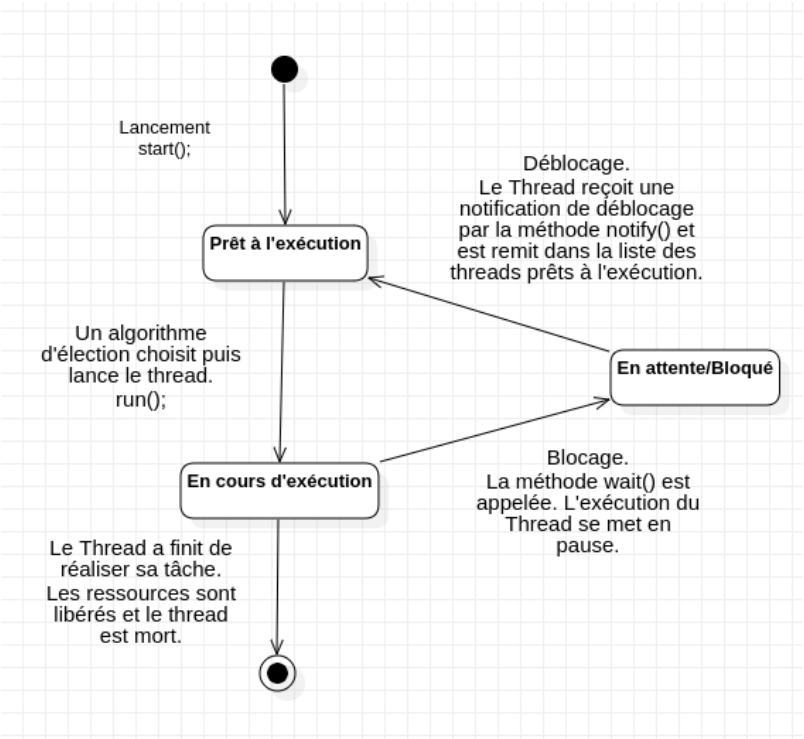

Figure 2: Diagramme du cycle de vie d'un thread en Java

```
public void actionPerformed(ActionEvent e) {
    // Récupére un ID unique au bouton correspond à son mobile
    int id = getButtonId(e);
    if (isRunning[id]) {
        threads[i].resume();
        boutons[i].setText("Suspend");
    } else {
        threads[i].suspend();
        boutons[i].setText("Resume");
    }
}
```

Dans ce morceau de code, à chaque mobile est associé un bouton. Lorsque ce bouton est cliqué, on inverse l'état du mobile (et au passage du bouton). Si le thread est en cours d'exécution, on le suspend, sinon on le relance.

Parallèlisme

Dans la suite de l'exemple utilisant un mobile se déplaçant horizontalement, nous allons maintenant ajouter un certain nombre N de mobiles.

```
for (int i = 0; i < N; i++) {
    // On sépare l'espace vertical entre tous les mobiles
    UnMobile mobile = new UnMobile (LARG, HAUT / N);
    Thread thread = new Thread(sonMobile);

    // Pour gérer le suspend/resume
    JButton bouton = createSuspendButton(i);
    isRunning[i] = true;
    threads[i] = thread;

    // Ajout à la fenêtre
    add(sonMobile);
    add(btn);
}
```

Chaque Thread est indépendant et donc on peut suspend un thread unique sans que ça influence les autres. On peut remarquer également lorsqu'il y a beaucoup (> 1000) mobiles qui bougent en même temps et qu'on les mets tous en suspend en même temps, alors ils ne sont pas tous synchronisés (à savoir tous à la même position). On peut expliquer ce décalage par le fait qu'une machine ne possède pas 1000 coeurs et que donc les threads ne sont pas physiquement tous exécutés en même temps. Au fils de l'execution, tous les threads n'auront pas été exécuter le même temps, créant ainsi un décallage.

De plus, on remarque que lorsqu'on créer un Thread par mobile, que chaque mobile est indépendant

que ça soit dans son exécution ou dans sa mémoire. Lorsqu'on modifie une valeur d'un mobile, les autres mobiles ne changent pas.

Dans une seconde partie, que ce passe-t-il si on utilise le même mobile dans plusieurs Thread. On remarque que le mobile "tremble", que de temps en temps il accelère et des fois retourne en arrière. Ce "tremblement" peut être expliqué par le fait que chaque Thread essaye de modifier la même variable mais pas tout le temps en même temps. On se retrouve ainsi dans ce cas :

- Le Thread 1 charge la position du mobile X dans sa mémoire.
- Le Thread 2 change la position du mobile à X + 1.
- Le Thread 3 change la position du mobile à X + 2.
- Le Thread 1 change la position du mobile à partir de X à X + 1.
- Conclusion : Le Thread 1 fait reculer le mobile en le faisant passer de X + 2 à X + 1.

Situation de compétition

On appel **situation de compétition** (ou *race condition*) une situation où le résultat peut être différent en fonction de l'ordre d'exécution des threads. C'est le cas dans l'exemple ci-dessous où le mobile se téléporte en fonction de l'ordre dans lequel ont été exécuter les threads. Dans ce cas, cela donne juste un rendu amusant, mais dans d'autres cas, cela peut causer des problèmes de sécurité pouvant être très dangereux. (cf. Élévation de privilère à l'aide d'une situation de compétition).

Ordonnanceur

L'objectif d'un thread est d'être executé en parallèle, il lui faut donc un coeur sur le CPU qui exécuter ses instructions. Or le nombre de threads n'est pas limité alors que le nombre de coeurs l'est. Par exemple, sur ma machine Linux, au lancement environ 50 threads sont déjà lancés alors que la machine ne possède que 8 coeurs. Il faut donc un mécanisme afin de gérer plus de thread que de coeurs disponibles.

En programmation, *start* un thread ne lance pas directement la tâche. Le thread est alors mis comme **prêt à l'exécution**.

C'est au rôle de l'**ordonnanceur** de choisir quand commencer la tâche. Le choix de l'ordre d'exécution se fait à l'aide d'un **algorithme d'élection** (FIFO, LIFO…). Ainsi, il n'est pas possible de prédire dans quel ordre seront lancés les tâches. Dans le cas de Java c'est au rôle de la JVM de faire cette élection. Il est également possible de donner une *priorité* à un thread en Java à l'aide de la méthode `setPriority`. La méthode prend en argument un nombre entre 1 et 10 indiquant dans quel thread devrait avoir la priorité. Mais cela ne reste qu'une indication et ne définie en aucun cas un exécution prévisible.

Sections critiques

Une **section critque** est un morceau de code qui ne doit être exécuter que par un seul Thread à la fois. Une section critique peut servir à protéger une **ressource critique**, à savoir une ressource qui peut être utiliser que par un seul processus (donc thread également) à la fois. La majorité du temps, c'est une zone de la mémoire (une variable). Il faut mettre en place des mécanismes afin d'éviter que plusieurs processus accèdent en même temps à la même zone de mémoire car cela pourrait produire des résultats indéfinis.

Mutex

Un **mutex** (de *mutual exclusion* ou *exclusion mutuelle*) est un verrou que l'on déclare autour du code de la une section critique. Lorsqu'un thread rentre dans une section critique, il prend alors possession de ce verrou. Lorsqu'un autre thread arrivera dans cette section critique il se retrouvera bloqué. Une fois que le premier thread quitte la zone critique, il libère le verrou. Ainsi le second peut à son tour prendre le verrou et entrer dans la section critique. Ainsi seulement un seul processus peut se trouver dans la section critique à un même instant.

Une zone protégée par un mutex (ou un système quelconque) est dite comme **atomique**. Un code atomique est un morceau de code qui ne peut pas être interrompu. Ainsi on peut être certain que l'état n'aura pas changé entre le début et la fin de l'exécution de ce morceau de code. Par exemple, il existe la classe `AtomicInteger` en Java, qui permet de stocker un `Integer` et d'être certain dont l'état ne peut pas être changée par deux threads à la fois.

Problème de deadlock

Il faut tout de même faire attention à la façon dont sont utilisés ses mutex. Il est très fréquent lors des premières manipulations de créer des situations de **deadlock**. Un deadlock se produit lorsque 2 thread essaye d'accéder à une ressource critique de l'autre thread. Les deux threads sont alors en attentes et le programme ne pourra jamais reprendre.

Heureusement, en Java, la directive **synchronized** s'occupe automatiquement de la gestion des mutex.

Exemples de protection d'une section critique

A partir de ce morceau de code, nous souhaitons que l'affichage soit "AAABB" ou "BBAAA" mais surtout pas "AABAB" ou "BAABA".

```
public class Main {
    public static void main(String[] args) {
        Affichage TA = new Affichage("AAA");
        Affichage TB = new Affichage("BB");

        TB.start();
        TA.start();
    }
}
```

Ci-dessous la classe `Affichage` s'occupant d'afficher le texte qu'il possède caractère par caractère. On attend après chaque lettre affichée afin de simuler une opératin lente.

```
public class Affichage extends Thread {
    String texte;

    public Affichage(String texte) {
        this.texte = texte;
    }

    // Affiche un caractére puis attend 100 secondes pour simuler des tâ
       ches longues.
    public void run() {
        for (int i = 0; i < texte.length(); i++) {
            System.out.print(text.charAt(i));
            try { sleep(100); } catch { InterruptedException e) {}
        }
    }
}
```

Au lancement du programme on remarque que les lettres sont mélangées. Cela s'explique par le fait qu'après qu'un A ait été affiché, et qu'il est en train de *sleep*, rien ne bloque le second thread d'afficher son B.

Dans un premier temps il faut trouver la ressource critique. Notre problème est que les deux Threads peuvent afficher leur lettre en même temps et donc écrire "ABAAB" au lieu de "AAABB" ou "BBAAA". Cette affichage passe par le buffer présent dans System.out. C'est donc cette classe "out" qui est la ressource critique du programme.

Ensuite il faut détecter quel est l'endroit dit critique dans la méthode run. Il a 2 choix, soit le System.out.print, soit la boucle for. Si on décide de synchroniser le System.out.print, on bloque uniquement le fait qu'on puisse afficher 2 lettres en même temps et non l'affichage en entier de la chaine de caractère.

Alors qu'en décidant de synchroniser la boucle for (qui affiche le String en entier), on fait en sorte qu'un seul Thread à la fois puisse afficher EN ENTIER son texte. Au final on affiche chaque String l'un après

l'autre et non les deux mélangés.

On se retrouve avec le code suivant avec la mention **synchronized**. Cette directive Java, prend un argument X qui est une ressource critique. C'est ensuite Java (à savoir la JVM) qui s'occupe de faire en sorte qu'un seul thread puisse être présent dans ce bloc de code en même temps. **Synchonized** permet de rendre une opération *atomique*

```
public void run() {
    synchronized (X) {
        for (int i = 0; i < texte.length(); i++) {
            System.out.print(texte.charAt(i));
            try { sleep(100); } catch (InterruptedException e) {}
        }
    }
}
```

Il manque plus qu'à savoir sur quel objet X on souhaite faire la synchronisation. On peut créer une classe dite "bidon" qui servira d'objet de synchronisation

```
class Exclusion {}
```

Il faut maintenant ajouter cet objet à la classe Affichage :

```
public class Affichage extends Thread {
    String text;
    Exclusion ex = new Exclusion();

    public void run() {
        synchronized (ex) {
            for (int i = 0; i < texte.length(); i++) {
                System.out.print(texte.charAt(i));
                try { sleep(100); } catch (InterruptedException e) {}
            }
        }
    }
}
```

Le problème est que ce morceau de code ne fonctionnera pas. Il y a maintenant un nouveau problème. En effet, *synchronized* synchronise un objet, sauf que dans le cas présent, les 2 threads `Affichage` (TA et TB) on chacun leur propre instance de `Exclusion` et font donc la synchronisation sur 2 objets différents (et par extension aucune synchronisation). Pour régler ce dernier problème, il faut marquer l'objet `Exclusion` comme **`static`** pour que tous les threads `Affichage` utilisent la même instance de la classe `Exclusion`. *Synchronized* fera donc la synchronisation sur une instance commune se qui bloquera tous les autres threads qui arrivent sur la section critique tant que le thread déjà dedans n'a pas terminé.

```java
public class Affichage extends Thread {
    String text;
    static Exclusion ex = new Exclusion();

    public void run() {
        synchronized (ex) {
            for (int i = 0; i < texte.length(); i++) {
                System.out.print(texte.charAt(i));
                try { sleep(100); } catch (InterruptedException e) {}
            }
        }
    }
}
```

Après ce dernier changement, la synchronisation fonctionne et on obtient dans le terminal soit “AABBB” soit “BBAAA” en fonction de quel thread s’est lancé en premier.

Sémaphores binaire

Au lieu d’utiliser la directive “*synchronized*” proposée par Java, dans cette partie nous allons utiliser notre propre implémentation des sémaphores afin de mieux en comprendre le fonctionnement.

Un sémaphore possède une valeur, correspond au nombre de ressources disponibles. Dans le cas d’un sémaphore binaire, la valeur peut être sois 0 (aucune ressource disponible), soit 1 (la ressource est disponible).

`syncWait()` permet de demander au sémaphore de prendre possession d’une ressource. Si la *valeur* du sémaphore est déjà à 0, alors le thread est bloqué jusqu’à qu’une ressource soit libérée.

`syncSignal()` permet de dire aux autres threads qu’une ressource vient d’être libérée.

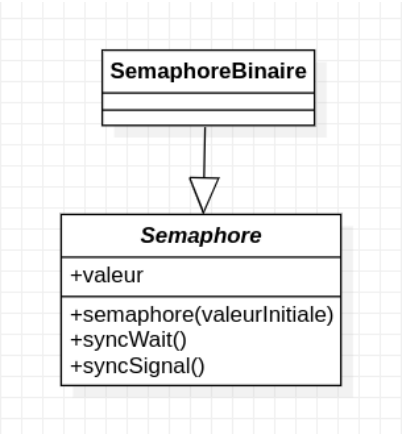

Figure 3: Diagramme de classe des sémaphores binaires

Il faut, comme dans le premier exemple avec *synchronized*, utiliser un sémaphore commun entre tous les threads. Si chaque thread possède sa propre instance de sémaphore, alors les threads n'ont pas de moyen de communiquer entre eux et donc savoir s'ils peuvent entrer ou non dans une section critique.

```
public class Affichage extends Thread {
    static semaphoreBinaire semaphore = new semaphoreBinaire(1);
    String texte;
}
```

On peut maintenant utiliser ce sémaphore commun pour qu'un seul thread puisse accéder à une section donnée.

```
public void run(){
    semaphore.syncWait();
    for (int i = 0; i < texte.length(); i++){
        System.out.print(texte.charAt(i));
        try { sleep(100); } catch(InterruptedException e) {};
    }
    semaphore.syncSignal();
}
```

Le `syncWait()` permet de prévenir les autres threads que l'on vient de rentrer dans une section critique. Ainsi, dès qu'un autre thread arrive sur cette même ligne, il sera bloqué tant que la ressource n'est pas libérée.

Le `syncSignal` permet de libérer tous les threads en attente. Juste après la libération, un thread executera cette ligne est prendra à son tour le contrôle de la section critique. On peut s'assurer que seulement 1 thread sera présent dans cette section car syncWait est lui même synchonized.

De plus, il faut remarquer que la valeur 1 à été donnée au constructeur de semaphoreBinaire. Si on passe cette valeur à 0, alors aucun thread ne pourra rentrer dans une section critique car le 0 indique que la ressource est prise. Il y a alors aucun moyen de libérer une ressource si personne ne peut la prendre dans un premier lieu. On se retrouve dans une situation de **dead lock**

Sémaphore général

Précédemment, de par le sémaphore binaire, uniquement un seul thread pouvait être présent dans une section critique. Mais rien ne nous oblige à ce qu'un seul thread puisse rentrer dans cette section.

Nous allons maintenant ajouter un sémaphore général qui peut autoriser un nombre N de thread dans une section critique.

La première étape est d'implémenter ce sémaphore général. La seule différence avec le sémaphore

binaire est que `valeur` peut prendre n'importe quelle valeur positive. Ainsi, le sémaphore ne sera pas bloquant dans que des ressources sont disponibles (à savoir tant que `valeur > 0`).

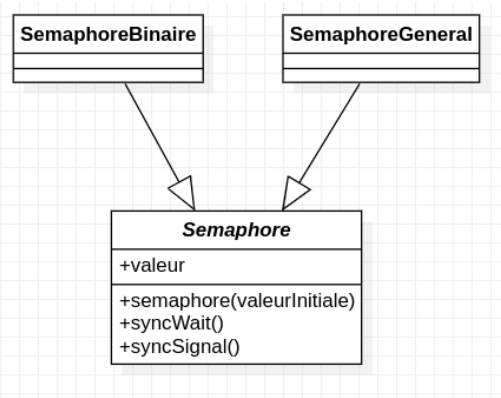

Figure 4: Diagramme de classe des sémaphores généraux

```
public final class semaphoreGeneral extends semaphore {
    public semaphoreGeneral(int valeurInitiale){
        super(valeurInitiale);
    }
}
```

Afin d'utiliser ce sémaphore, nous allons reprendre l'exemple du mobile. Un mur est présent au centre de la zone horizontale de chaque mobile et limite le nombre de mobiles à l'intérieur. Seulement N mobiles peuvent être présent dans cette zone (qui est donc la section critique). Tous les autres attendent devant le mur qu'une ressource soit libérée afin de rentrer.

Dans l'exemple, N = 3

avancerTier() est une méthode fictive qui avance le mobile d'un tier de sa largeur maximum.

```
class UnMobile extends JPanel implements Runnable {
    static semaphore sem = new semaphoreGeneral(3);

    int saLargeur, saHauteur, sonDebDessin;
    final int sonPas = 10, sonCote = 50, sonTemps = 40;
    int sonTemps;

    UnMobile(int telleLargeur, int telleHauteur) {
        super();
        saLargeur = telleLargeur;
        saHauteur = telleHauteur;
        setSize(telleLargeur, telleHauteur);
    }

    public void run() {
        while (true) {
            sonDebDessin = 0;
            // Avance jusqu'au premier tier.
            avancerTier();

            sem.syncWait();
            // Avance jusqu'au 2éme tier.
            // Ici nous sommes dans une section critique.
            avancerTier();
            sem.syncSignal();

            // Avance jusqu'à la fin.
            avancerTier();
        }
    }
}
```

Le *syncWait()* permet de bloquer l'accès à la section critique lorsque N mobiles sont déjà présents. Ainsi, il n'est pas possible d'avoir plus de N mobiles présent dans le tier central.

Modèle producteur/consommateur

Les sémaphores sont ce qu'une solution parmi d'autres afin de gérer les problèmes de synchronisation entre plusieurs tâches. Sauf que cette solution n'est pas valable dans tous les cas.

Une autre solution existante est le modèle producteur/consommateur (ou ecrivain/lecteur). Dans ce modèle, le producteur fournis de la donnée et le consommateur la récupère quand elle est disponible.

On peut prendre l'exemple d'un *buffer* dans lequel le producteur peut écrire du texte et le consomma-

teur peut lire le texte. Le consommateur ne doit pas pouvoir lire plus loin que ce que le producteur à écris. De plus, plusieurs producteurs ne peuvent pas écrire en même temps dans le *buffer* afin d'éviter d'écrire dans la même cellule. Également, deux consommateurs ne peuvent pas lire en même temps. Toutefois, un consommateur peut lire pendant que le producteur écris et réciproquement.

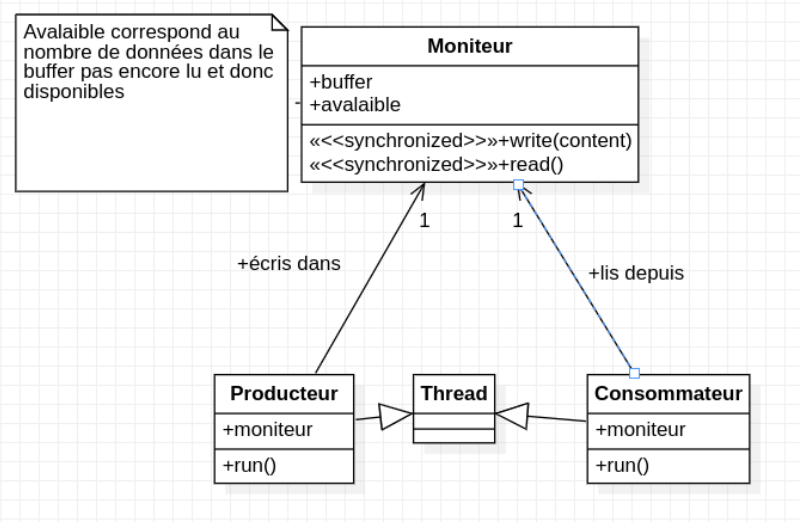

Figure 5: Diagramme de classe du modèle producteur/consommateur

Les méthodes **write** et **read** doivent être *synchonized* afin d'éviter comme dit précédemment que deux producteurs ou consommateurs soient exécutés en même temps.

API Concurrent

En 2004 sort la version 5 de Java dans laquelle une nouvelle API est ajoutée. Cette API ajoute un certain nombre de classes et d'outils facilitant la *programmation concurrente*.

On peut par exemple retrouve l'interface `BlockingQueue` qui peut être utilisée afin d'implémenter le modèle producteur/consommateur. La `BlockingQueue` sert de fille d'attente permettant au producteur d'envoyer ses données et au consommateur de les récuper. La `BlockingQueue` bloque jusqu'à ce qu'un élément soit ajouter et bloque également jusqu'à ce que de la place soit disponible. Donc l'interface `BlockingQueue` sert de **moniteur**.

Différentes implémentations sont disponibles en fonction de nos besoins comme la `ArrayBlockingQueue`, `PriorityBlockingQueue` ou `LinkedBlockingQueue`.

La `BlockingQueue` fournit plusieurs méthodes avec principalement `offer` et `poll`. `offer` permet d'ajouter un élément à la queue et `poll` permet d'en récupérer.

Il existe pour ses deux méthodes des surcharges dans lesquels il est possible d'également passer en argument un temps à attendre avant d'arrêter l'opération. Par exemple `poll(200, TimeUnit.MILLISECONDS)` attendra 200 millisecondes, après lesquels il arretera d'attendre et quittera la méthode.

Exemple d'utilisation

Dans cet exemple servant à montrer comment implémenter le modèle producteur/consommateur et aussi l'utilisation de l'API concurrent, nous allons définir une `Boulangerie`. La boulangerie ne peut avoir qu'un certain nombre de pain max à vendre, peut préparer de nouveaux pains et vendre du pain à des clients.

Premièrement, nous allons implémenter la classe `Boulangerie`. Une `Boulangerie` peut avoir au maximum 20 pains à la fois. Si un nouveau pain est cuit, et qu'il n'y a plus de place libre, alors on jete le pain. Si aucun pain n'est disponible, le client part. Il existe plusieurs boulangeries et plusieurs clients, tous représentés par un thread.

```
public class Boulangerie {
    // Seulement 20 pains peuvent être mis en vente à la fois.
    private BlockingQueue<Pain> queue = new ArrayBlockingQueue<Pain>(20) ;

    // Ajoute un pain à vendre. Renvoie true si l'opération à fonctionnée.
    public boolean depose(Pain pain) {
       return queue.offer(pain);
   }

    // Vend un pain à un client. Si aucun pain est disponible, le client
        attend qu'un pain soit mis en vente.
    // Renvoie null si aucun pain n'est disponible
    public Pain achete() {
       return queue.poll();
   }

    // Récupére le nombre de pains disponibles
    public int getStock() {
       return queue.size();
   }
}
```

Il faut maintenant implémenter le `Boulanger`, qui toutes les 1 seconde produit un pain et le met en vente.

```
public class Boulanger implements Runnable {
    public void run() {
        while (true) {
            // Toutes les secondes un boulanger produit un pain
            Thread.sleep(1000) ;
            if (boulangerie.depose(new Pain())) {
                System.out.println("Le boulanger à ajouté un pain");
            }
        }
    }
}
```

Et enfin, de façon aléatoire, un client va venir acheter un pain.

```
public class Client implements Runnable {
    public void run() {
        while (true) {
            // Un client arrive de façon aléatoire entre 0 et 1000ms.
            Thread.sleep(rand.nextInt(1000));
            Pain pain = boulangerie.achete();
            if (pain != null) {
                System.out.println("miam miam") ;
            }
        }
   }
}
```

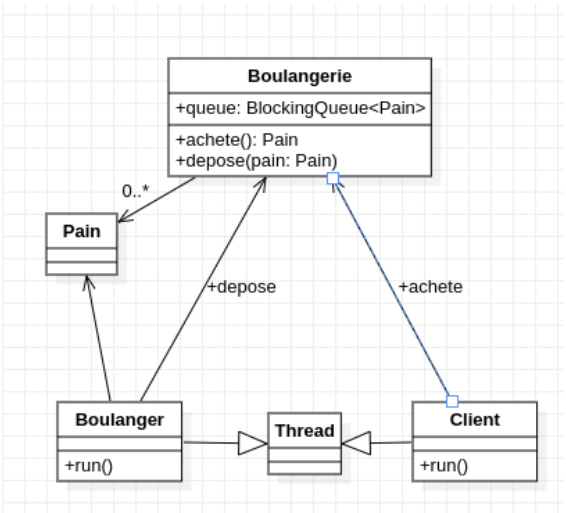

Figure 6: Diagramme de classe de la Boulangerie

Conclusion

En contrepartie d'une forte augmentation des puissances de calculs à l'aide des *processeurs multi-coeurs*, de nombreuses contraintes sont apparus pour les développeurs. Il faut désormais contrôle l'ordre d'exécution imprévisible des tâches sur une machine. Heuresement, au fil des années, des solutions ont fait leur apparition tel que les *mutex*, *semaphores* ou l'*API concurrent* afin de gérer ses difficultés. Nous pouvons aujourd'hui écrire des programmes bien plus performants et parfois répartis sur plusieurs machines.

Références

- Cours de Mr.Dufaud
- Systèmes répartis : http://benjamin.dautrif.free.fr/contenu/reseaux/poly_sr05.pdf
- Architectures : https://perso.isima.fr/~locrombe/pdf/prog_rep_cours1.pdf
- Utilisation de l'API concurrent : http://blog.paumard.org/cours/java-api/chap05-concurrent-queues.html
- Situation de compétition : https://fr.wikipedia.org/wiki/Situation_de_comp%C3%A9tition.
  - De plus voici une vidéo montrant un cas réel d'utilisation d'une situation de compétition afin de réaliser une *élévation de privilèges* : https://www.youtube.com/watch?v=oIAP1_NrSbY
- Les morceaux de codes (des TP) utilisés dans le rapport sont disponibles sur mon GitHub : https://github.com/EvannBerthou/ProgrammationRepartieS4