SYNCJ: Mécanismes de synchronisation avancée pour Java

MEMBRES DE L'ÉQUIPE: Nour Romdhane, Liantsoa Rasata, Mathilde Leparquier, Ibrahim Benali, Othmane Kabir.

ENCADRANT: Jean-Louis Pazat

23 octobre 2014

Introduction

Dans le cadre de notre deuxième année d'étude au département Informatique de l'INSA de Rennes, nous allons réaliser un projet technique en équipe. Le sujet de notre projet porte sur les mécanismes de synchronisation avancée en Java. Ainsi nous développerons des solutions pour faciliter l'apprentissage des outils de synchronisation souvent fastidieux pour les programmeurs.

Ce projet est pour nous relativement exploratoire, de ce fait il fait appel à des compétences de programmation sur le parallélisme que nous devons acquérir.

Ce rapport présente la phase d'analyse comportant ainsi la phase de pré-étude et la spécification générale du projet.

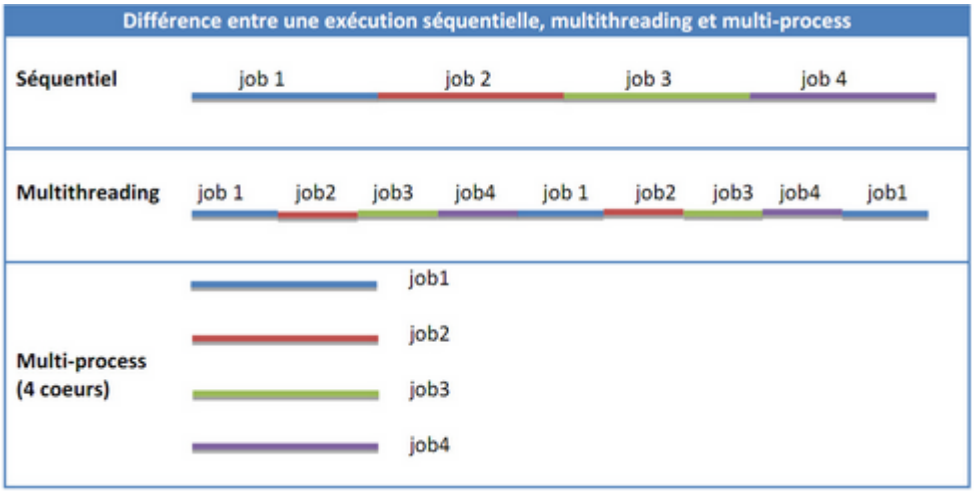
1 Contexte du projet

1.1 Notion de parallélisme

Le parallélisme est une propriété fondamentale des applications qui a une influence sur l'architecture des ordinateurs. Elle est souvent attachée à la notion de la performance d'exécution des applications. Ce dernier terme recouvre plusieurs notions selon les besoins des applications. En effet, quel que soit le domaine d'application, le parallélisme peut être exploité pour répondre à deux grands besoins : la disponibilité et/ou la puissance de traitement.

Sur la figure suivante, nous pouvons observer qu'une exécution séquentielle va accomplir chaque tâche *(job)* un par un sur un seul coeur. L'exécution multi-threading permet quand à elle sur un seul coeur d'exécuter

plusieurs tâches. Enfin l'exécution multi-process qui consiste à utiliser plusieurs coeurs pour exécuter les tâches en parallèle.



Il existe quatre types principaux de parallélisme [1] : SISD, SIMD, MISD et MIMD.

Cette classification permet d'expliquer les bases de l'architecture des ordinateurs séquentielle et parallèlle. Elle est basée sur les notions de flot de contrôle (deux premières lettres, I voulant dire "Instruction") et flot de données (deux dernières lettres, D voulant dire "Data").

Par ailleurs, le moindre micro-processeur courant inclut lui-même plusieurs formes de micro-parallélisme et par conséquent cette classification peut paraître artificielle.

`SISD` : architecture séquentielle avec un seul flot d'instructions, un seul flot de données, exécution déterministe.

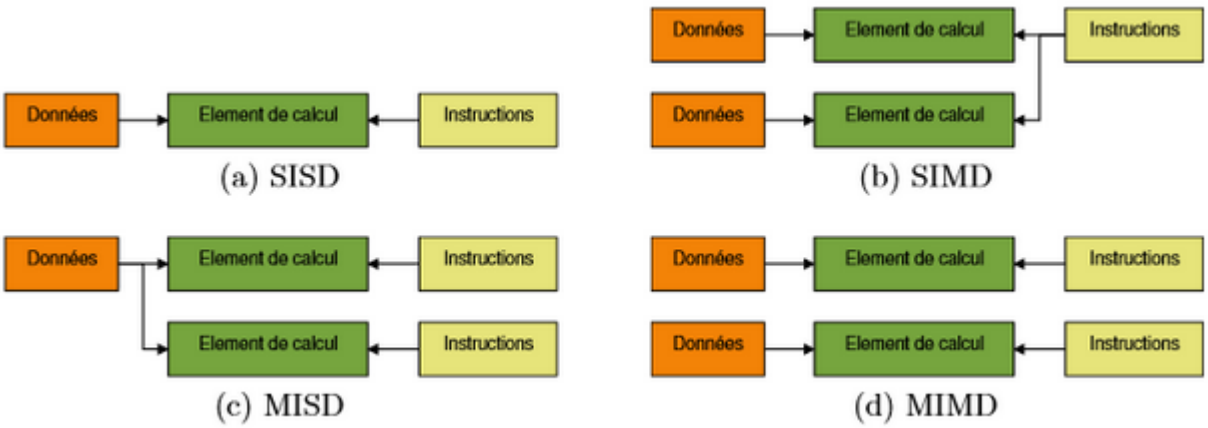
`MISD` : un seul flot de données alimente plusieurs unités de traitement, chaque unité de traitement opère indépendamment des autres, sur des flots d'instructions indépendants. Cette architecture est peu implémentée.

`SIMD` : architecture parallèle, toutes les unités de traitement exécutent la même instruction à un cycle d'horloge donné, chaque unité peut opérer sur des données différentes, exécution déterministe.

`MIMD` : architecture la plus courante. Chaque unité de traitement peut gérer un flot d'instructions différent. Chaque unité peut opérer sur un flot de données différent. L'exécution peut être synchrone ou asynchrone, déterministe ou non-déterministe.

Le schéma suivant illustre ces 4 types principaux de parallélisme :

1. Taxonomie de Flynn Tanenbaum : classification des architectures d'ordinateur, proposée par Michael J. Flynn en 1966



1.2 Quelques exemples de machines parallèles

Dans cette section, l'objectif est de donner une idée des super-calculateurs utilisés dans le monde industriel et du calcul scientifique.

1.2.1 Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom

Blue Gene est une architecture de superordinateurs. Le projet est cofinancé par le Département de l'Énergie des États-Unis (DoE) et développé par IBM.

Il contient 1,572,864 coeurs avec une puissance de 7,890.00 kW, une mémoire de 1,572,864 GB, un processeur Power BQC 16C 1.6GHz et un système sous linux. Il a atteint un pic théorique de 20,132.7 TFlop/s et des performances linpack de 17,173.2 TFlop/s. Il a pris la tête du classement TOP500 du 16 septembre 2004 au 26 octobre 2004 (dans une version non finalisée), avec 36,01 téraflops au test Linpack qui est une bibliothèque de fonctions pour l'algèbre linéaire servant à classer les plus puissants superordinateurs du monde dans le TOP500.

La figure suivante montre la machine Blue Gene :

FIGURE 1 – *Blue Gene*

1.2.2 Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x

Titan est un superordinateur construit par Cray Inc. pour le Laboratoire national d'Oak Ridge dans le Tennessee. À partir de 2013, il effectue des calculs pour des projets scientifiques.

Il s'agit en fait d'une mise à jour du système Jaguar installé à Oak Ridge, notamment par l'installation d'accélérateurs GPU Tesla K20X de Nvidia. Il contient 560,640 coeurs avec une puissance de 8,209.00 kW, une mémoire de 710,144 GB, un processeur Opteron 6274 16C 2.2GHz et un système sous Cray Linux Environment. Il a atteint un pic théorique 27,112.5 TFlop/s TFlop/s et des performances linpack de 17,590 TFlop/s.

La figure suivante montre la machine Titan :

FIGURE 2 – *Titan*

1.2.3 Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P

Le Tianhe-2 est un supercalculateur situé à Changsha dans la province de Hunan en République populaire de Chine, à l'Université nationale de technologie de la défense .

Il est un assemblage muni de 32000 processeurs Intel Ivy Bridge de 48000 Intel Xeon Phi E5-2692v2 12C 2.2GHz pour un total de 3 120000 cœurs avec une puissance de 17,808.00 kW, une mémoire de 1,024,000 GB et un système sous Kylin Linux. Il a atteint un pic théorique 54,902.4 TFlop/s TFlop/s et des performances linpack de 33,862.7 TFlop/s.

La figure suivante montre la machineTianhe-2 :

FIGURE 3 – *Tianhe-2*

1.3 Le GPU

Le GPU ou *Graphical Processing Unit* désigne le processeur dédié au traitement des données graphiques. Avec l'augmentation gigantesque de leurs performances, les GPU peuvent maintenant effectuer beaucoup de calculs parallèles bien qu'au départ, il était utilisé à des fins de décrytage. Généralement, le GPU est embarqué sur une carte dite graphique, mais il se trouve de plus en plus intégré au chipset des cartes mères.

Le calcul par le GPU consiste à utiliser le processeur graphique (GPU) en parallèle du CPU pour accélérer des applications professionnelles de science, d'ingénierie et d'entreprise. Lancé en 2007 par NVIDIA, le calcul par le GPU s'est imposé comme un standard de l'industrie. Dans le monde entier, la plupart des centres de données à basse consommation y ont recours, aussi bien dans les laboratoires gouvernementaux et universitaires que dans les petites et moyennes entreprises.

Le calcul par le GPU permet de paralléliser les tâches et d'offrir un maximum de performances dans de nombreuses applications : le GPU accélère les portions de code les plus lourdes en ressources de calcul, le reste de l'application restant affecté au CPU. Les applications des utilisateurs s'exécutent ainsi bien plus rapidement.

La figure suivante présente comment un GPU accélère le travail :

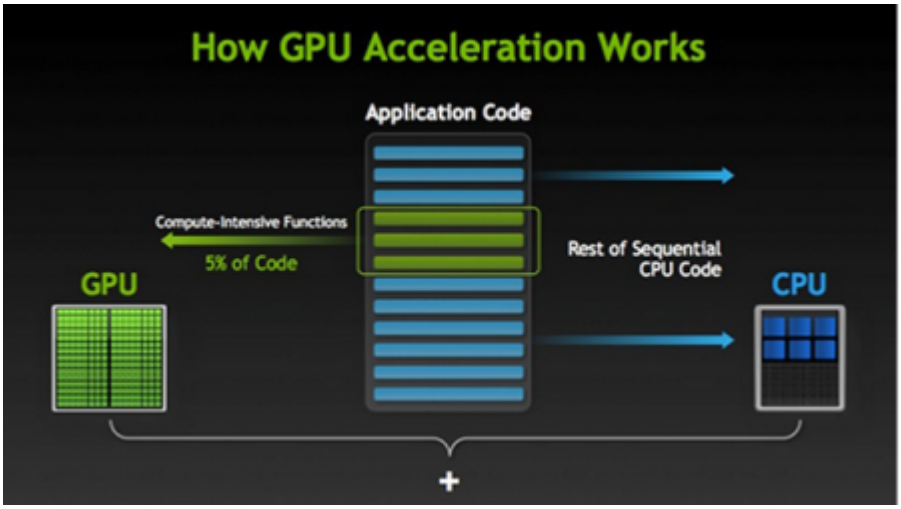


FIGURE 4 – *Mode de fonctionnement d'un GPU*

Différences entre CPU et GPU

Pour comprendre les différences fondamentales entre un CPU et un GPU, il suffit de comparer leur manière de traiter chaque opération. Les CPU incluent un nombre restreint de cœurs optimisés pour le traitement en série, alors que les GPU intègrent des milliers de cœurs conçus pour traiter efficacement de nombreuses tâches simultanées.

La figure suivante illustre cette différence :

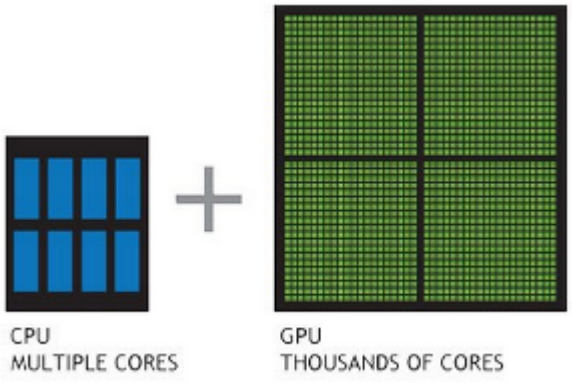


FIGURE 5 – *Différence entre un CPU et un GPU*

Exemples de GPU

Serveur avec accélérateurs bullx R421 E4 : des GPU ou coprocesseurs intégrés :

Conçus spécifiquement pour le calcul intensif, les serveurs avec accélérateurs bullx R421 E4 associent sur 1U :

- deux processeurs Intel Xeon E5-2600 v3 d'une part, et
- deux GPU NVIDIA Tesla ou
- deux coprocesseurs Intel Xeon Phi de dernière génération.

Moins denses que les lames avec accélérateurs bullx DLC B715, le R421 E4 offre en revanche une flexibilité supérieure avec un choix plus riche d'options et de connectivité.

Lames bullx DLC B715 : des GPU ou coprocesseurs intégrés associés à l'efficacité énergétique du refroidissement liquide direct :

Développées par les équipes R&D de Bull, les lames avec accélérateurs bullx B715 s'appuient sur le principe de refroidissement liquide direct en associant dans une même lame :

- deux processeurs Intel Xeon d'une part, et
- deux GPU NVIDIA Tesla ou
- deux coprocesseurs Intel Xeon Phi.

1.4 Notions de processus et de thread

Il est important de savoir différencier la notion de processus et la notion de thread.

Un processus est un programme en cours d'exécution auquel est associé un environnement processeur et un environnement mémoire.

Tandis qu'un thread est une portion de code qui se déroule en parallèle au thread principal, comme l'image de la fonction fork() sur Linux mais sans

recopie du processus père. Ainsi nous pouvons faire de la programmation multitâche en définissant des fonctions qui vont se lancer en même temps.

Par conséquent, un thread est plus avantageux qu'un processus car un processus possède une copie unique de ses propres variables tandis qu'un Thread partage tous ses variables. De plus en terme d'économie d'espace mémoire et de temps, la communication entre processus reste très lente par rapport aux Threads.

Ce mécanisme est expliqué par le schéma suivant :

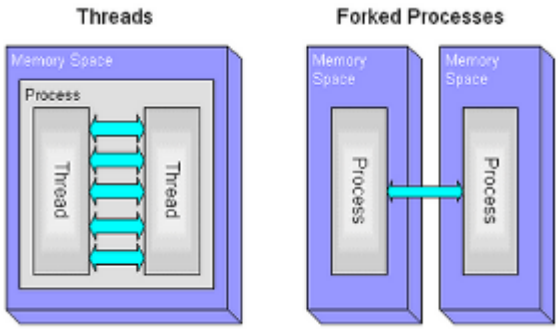


FIGURE 6 – *Différence entre un processus et un thread*

Cependant, la programmation multithread nécessite une maîtrise du mécanisme de fonctionnement des threads afin d'éviter toute sorte d'anomalie à l'exécution. D'ailleurs, il est souvent très difficile de trouver les lignes qui ont été la cause de l'erreur dans le programme.

1.5 Notion de concurrence d'accès

Comme nous l'avons vu précédemment, la programmation multithread permet à différents threads s'exécutant au sein d'une application d'accéder en lecture comme en écriture à toutes les données présentes dans l'espace mémoire réservée par cette application. En conséquence, il peut arriver que deux threads tentent d'accéder simultanément à une même donnée. Deux problèmes fondamentaux surviennent lors des accès en écriture : d'abord la modification apportée par un thread peut ne pas être immédiatement visible par les autres threads, ensuite il y a le problème de cohérence lors de la modification concurrente.

2 Les outils de synchronisation existants

Pour réaliser la synchronisation entre les processus, il existe plusieurs mécanismes.

2.1 Sémaphore :

Un sémaphore est un mécanisme proposé par E.W.Dijkstra en 1965.Il se présente comme un distributeur de jetons, mais le nombre de jetons est fixe et non renouvelable. Les processus doivent restituer leur jeton après utilisation. Les deux opérations sur un sémaphore sont traditionnellement dénotées P(s)et V(s).

- `P(s)` : permet à un processus d'obtenir un jeton, s'il y en a de disponibles. Si aucun n'est disponible, le processus est bloqué.
- `V(s)` : permet à un processus de restituer un jeton. Si des processus étaient en attente de jeton, l'un d'entre eux est réactivé et le reçoit.

2.2 Moniteur

Un moniteur c'est une structure composée des variables partagées par plusieurs processus et les opérations qui les manipulent. Ces opérations sont définies par le programmeur à qui l'on fournit l'exclusion mutuelle à l'intérieur du moniteur. La structure du moniteur donne la permission d'accés à un seul processus à l'intérieur d'un moniteur. Par conséquent, le programmeur n'a pas besoin de coder explicitement la synchronisation ; elle est réalisée dans le type moniteur.

Comme le montre la figure ci-dessous, le moniteur se compose des différentes parties suivantes :

- déclaration des variables partagée qui ne sont pas accessibles en dehors du moniteur.
- des procédures et des fonctions internes au moniteur. Elles sont les seules à manipuler des variables partagées. Leurs paramètres constituent le lien avec le programme.
- un corps comportant l'initialisation des variables.

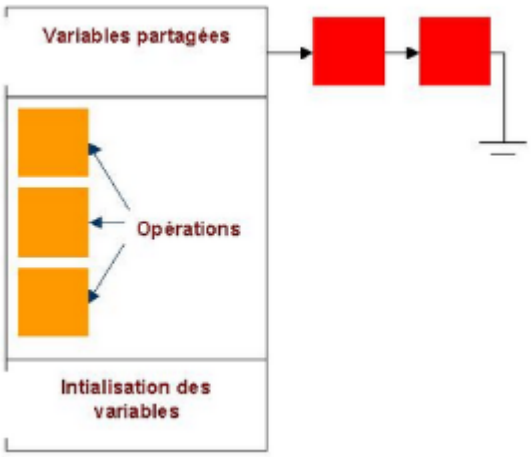


FIGURE 7 – *Moniteur*

2.3 Régions critiques

Il s'agit de gérer l'accés aux variables partagées. Afin qu'une variable partagée ne soit pas modifiable par plusieurs processus en même temps, on utilise la notion de région critique. Afin de réaliser cette exclusion mutuelle, les régions critiques associées à une même variable sont mutuellement exclusive et les variables partagées ne sont accessibles qu'à l'intérieur des régions critiques.

Les régions critiques conditionnelles permettent de bloquer les processus jusqu'à ce qu'ils répondent à une condition, cette condition est réévaluée à chaque fois que la variable est modifiée, si elle est vérifiée, le processus est débloqué. L'intérêt est de limiter la synchronisation, elle ne s'effectue que lorsque c'est nécessaire et pas automatiquement (lorsqu'il n'y a pas de manipulation de variable par exemple).

2.4 Evénements futurs

L'événement futur est présenté par une variable à assignation unique :
- un seul processus peut l'initialiser.
- tout processus peut la lire.
- lecture blocante jusqu'à ce que la variable ait reçu une valeur.

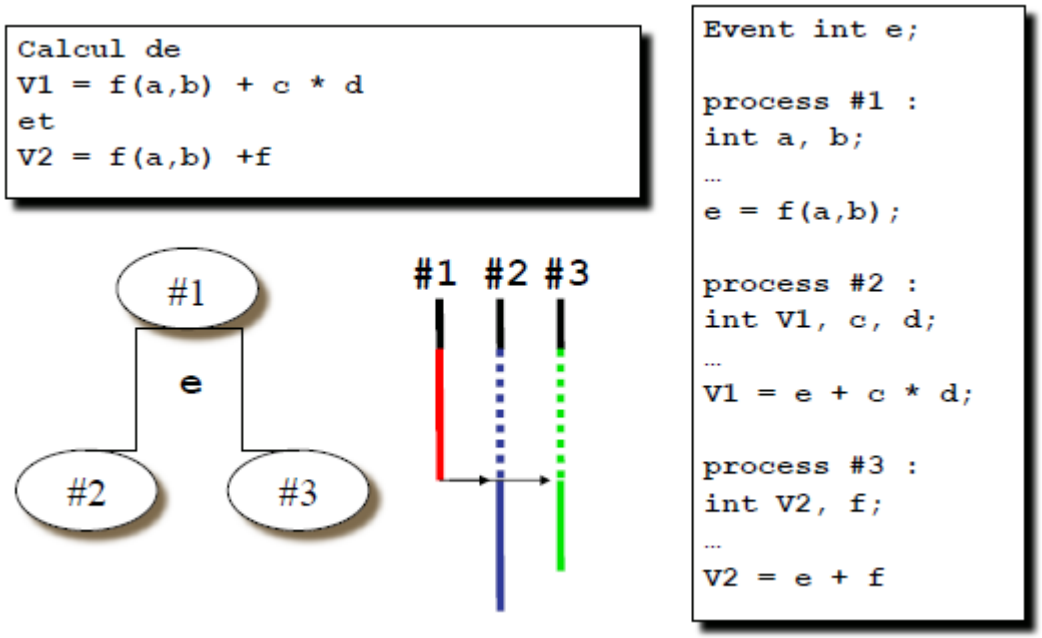


FIGURE 8 – *Illustration de la notion d'événement futur*

3 Problèmes de synchronisation typiques

3.1 Exclusion mutuelle :

L'exclusion mutuelle permet d'éviter que des ressources partagées d'un système ne soient utilisées en même temps.

Exclusion mutuelle = 1 seule permission accordée.

Le problème de l'exclusion mutuelle peut être simplement résolu à l'aide d'un sémaphore dont la valeur initiale est fixée à 1. Un tel sémaphore est couramment appelé sémaphore d'exclusion mutuelle (mutex).

La valeur de compteur =1 => Section critique libre

La valeur de compteur =0 => Section critique occupée

Les deux figures suivantes illustrent l'algorithme de l'exclusion mutuelle :

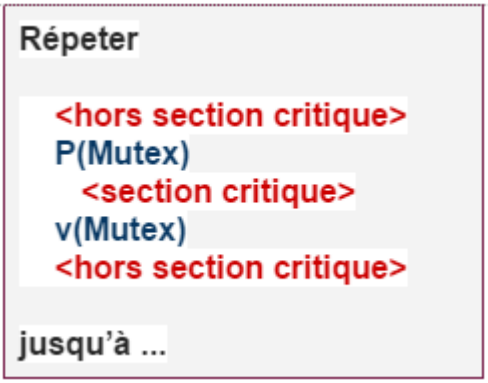


FIGURE 9 – *Algorithme*

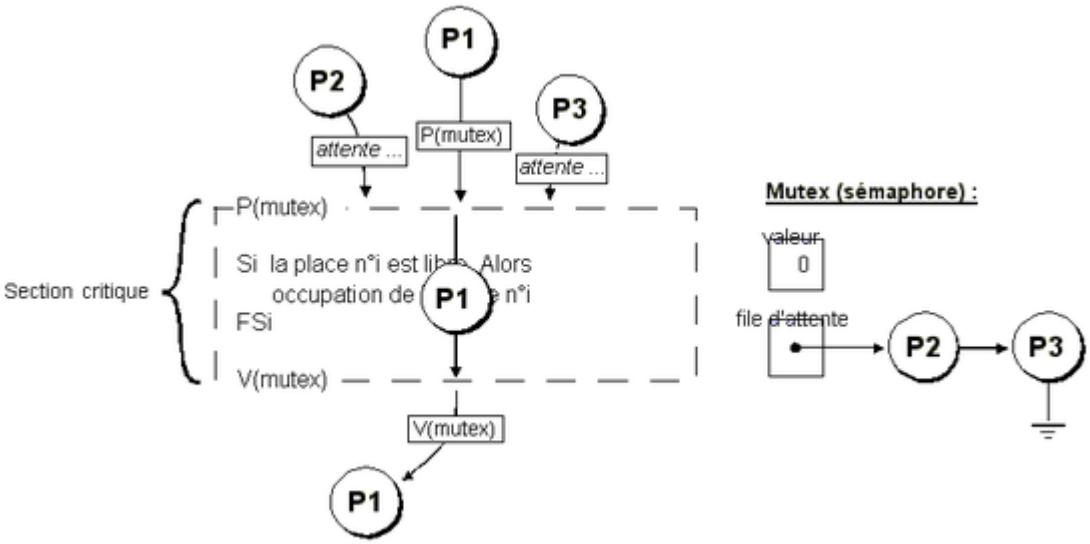


FIGURE 10 – *Utilisation des sémaphores pour l'exclusion mutuelle*

3.2 Lecteur/Rédacteur :

Le système Lecteur/Rédacteur permet de résoudre le problème d'accès à une ressource commune (un fichier par exemple) devant être partagée par deux types de processus (les « lecteurs »et les « rédacteurs ») :

- les lecteurs qui consultent le fichier sans le modifier, ils peuvent donc y accéder en parallèle.
- les rédacteurs qui accèdent au fichier pour le modifier, ils doivent accéder au fichier en exclusion mutuelle entre eux et en exclusion mutuelle avec les lecteurs.

Pour résoudre le problème des lecteurs et des rédacteurs, on peut utiliser les sémaphores, dans notre exemple illustré à la figure 11, on donnera la priorité aux lecteurs, donc on utilise trois sémaphores et une variable partagée :

- un sémaphore M_Lect, initialisé à 1 qui permet de protéger la variable Lect. Il s'agit donc d'un mutex.
- un sémaphore M_Red, initialisé à 1 qui permet de bloquer les tâches de rédaction (en induisant une priorité aux lecteurs). Il s'agit donc d'un mutex.
- un sémaphore Red, initialisé à 1 qui permet de bloquer les tâches de lecture si une rédaction est en cours.
- une variable Lect qui compte le nombre de lecteurs.

Lecteur	Rédacteur
Commencer_Lire : P(M_Lect) Lect++ SI Lect==1 ALORS P(Red) FIN SI V(M_Lect) **Finir_Lire :** P(M_Lect) Lect-- SI Lect==0 ALORS V(Red) FIN SI V(M_lect)	**Commencer_Ecrire:** P(M_Red) P(Red) **Finir_Ecrire:** V(Red) V(M_Red)

FIGURE 11 – *Exemple pour un lecteur et un rédacteur.*

3.3 Producteur/Consommateur

Le système Producteur/Consommateur permet de résoudre le problème de partage d'une zone mémoire entre deux processus (le consommateur et le producteur).

Les producteurs crééent des messages qui sont traités par les consommateurs . Les productions et les consommations peuvent être en parallèle. Un message ne peut être consommer qu'une seule fois.

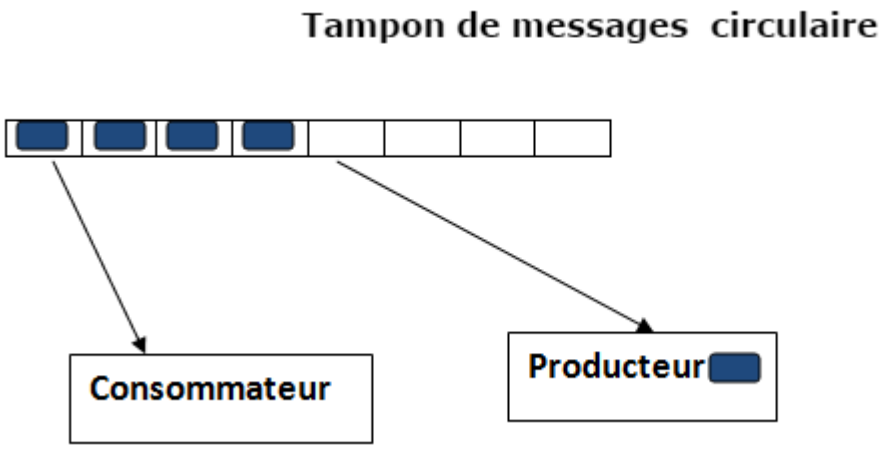


FIGURE 12 – *Tampon de messages circulaires*

Pour résoudre le problème des producteurs et des consommateurs, illustré à la figure 13, on utilise deux sémaphores :

- un sémaphore nvide, initialisé à n la taille du tampon , qui permet de bloquer la production si le tampon est plein.
- un sémaphore nplein, initialisé à 0 qui permet de bloquer la consommation si le tampon est vide.

Producteur	Consommateur

Produire (data m){
p(nvide);
Tampon[tete]:=m;
tete:=tete+1;
v(nplein);
}
	data consommer(){
	p(nplein);
	m:=Tampon[queue];
	queue:=queue+1;
	v(nvide);
	return (m);
	}

FIGURE 13 – *Exemple pour un producteur et un consommateur*

4 Mécanismes de synchronisation en Java

La synchronisation est une méthode que le langage java aussi bien que d'autres comme C++ ont implémenté et développé afin de régler le problème de l'accès concurrentiel des Threads aux ressources partagées notamment des variables, fonctions. . .

Le mécanisme de la synchronisation consiste à organiser la consommation de la section critique par les Threads d'une manière à ce que plusieurs threads n'exécutent pas en même temps cette partie de programme pour éviter toute sorte de conflits entre les différents Threads qu' on a créé.

Cependant une bonne maitrise du fonctionnement de la synchronisation est exigée afin de garantir qu'aucun bug ne sera généré lors de l'exécution du programme ce qui est la partie la plus délicate lors de programmations concurrente.

D'une manière générale lorsque le nombre de thread augmente, la programmation concurrente ne permet plus d'augmenter la vitesse d'exécution du programme ou plus précisément de diminuer la durée de son chemin critique. En effet, le programme passe son temps à mettre en attente les threads qui le composent et à écrire des informations qui permettent l'échange d'information entre thread. Ce phénomène est appelé le *parallel slowdown*.

Il existe de différentes primitives qui sont utilisées et appelée lors de la synchronisation parmi lesquelles on peut citer :

- `wait()` : bloque un thread jusqu'à ce qu'un autre thread le débloque.
- `wait(n)` : bloque un thread jusqu'à ce qu'un autre thread le débloque ou bien lorsque les n millisecondes termine.
- `notify()` : débloque un seul thread.
- `notifyAll()` : débloque tous les threads.

– `yield() :` rend la main à l'ordonnanceur (et donc laisse une chance aux autres threads de s'exécuter).
– `interrupt() :` provoque soit la levée de l'exception InterruptedException si l'activité est bloquée sur une opération de synchronisation, soit le positionnement d'un indicateur interrupted.
– `join() :` attente bloquante de la terminaison de l'exécution du thread (jusqu'à ce que la méthode run() associée au Thread ait fini de s'exécuter).

4.1 Exemple de l'utilisation des méthodes wait() et notify() :

La classe ListeTab contient deux méthodes getPremierElementBloquant qui fait appel à la primitive wait() et ajoute qui fait appel à la prmitive notify ().

Pour comprendre le fonctionnement de notre programme, on suppose qu'un thread T1 sur un objet ListeTab appelle ajoute et qu'un thread T2 appelle getPremierElementBloquant :

Si T2 commence en premier et vu que la méthode est définie synchronized en trouvant index == 0 vrai, il exécute wait(). Ce wait() est bloquant en attendant u'un notify() sur le même objet le libère . Après T1 exécute notify(). Ce notify () débloque T2 de manière asynchrone .

La figure ci-dessous montre le code de la classe ListeTab :

```java
class ListeTab {

    private String[] tab = new String[50];
    private int index = 0;

    synchronized void ajoute(String s) {
        tab[index] = s;
        index++;
        notify();
        System.out.println("notify() exécuté");
    }

    synchronized String getPremierElementBloquant() {
        //tant que la liste est vide
        while(index == 0) {
            try {
                //attente passive
                wait();
            } catch(InterruptedException ie) {
                ie.printStackTrace();
            }
        }
        return tab[0];
    }

}
```

FIGURE 14 – *Exemple de l'utilisation des méthodes wait() et notify()*

4.2 Exemple de l'utilisation de la methode join() :

La classe X contient l'activité d'un thread. La fonction foo() crée un thread X et doit attendre ce dernier jusqu'à ce qu'il finisse son activité grâce à la méthode join().

Voici le code de la classe X :

```
class X extends Thread{
public void run()
... code de l'activité...
}
}
//Utilisation
foo(){
X x = new X();
x.start();

x.join();
```

}

4.3 Exemple de l'utilisation de la méthode yield() :

Le thread créé par la classe ThreadDemo s'exécute en affichant son nom et la phrase "yielding programme...", puis il se met en pause en donnant la main à un autre thread pour s'exécuter à l'aide de la fonction yield. A la fin chaque thread affiche son nom ainsi que la phrase "has finished executing".

La figure suivante montre le code du programme et l'exécution :

```
   Thread t;

   ThreadDemo(String str) {

      t = new Thread(this, str);
      // this will call run() function
      t.start();
   }

   public void run() {

      for (int i = 0; i < 5; i++) {
         // yields control to another thread every 5 iterations
            if ((i % 5) == 0) {
               System.out.println(Thread.currentThread().getName() + "
               yielding control...");

               /* causes the currently executing thread object to temporarily
               pause and allow other threads to execute */
               Thread.yield();
            }
      }

      System.out.println(Thread.currentThread().getName() + " has
      finished executing.");
   }

   public static void main(String[] args) {
      new ThreadDemo("Thread 1");
      new ThreadDemo("Thread 2");
      new ThreadDemo("Thread 3");
   }
}
```

```
Thread 1 is yielding control...
Thread 2 is yielding control...
Thread 3 is yielding control...
Thread 1 has finished executing.
Thread 2 has finished executing.
Thread 3 has finished executing.
```

FIGURE 15 – *Exemple de l'utilisation de la méthode yield()*

5 Spécification générale

5.1 Cahier des charges

Pour atteindre l'objectif du projet qui est de faciliter l'apprentissage des outils de synchronisations, nous allons mettre en oeuvre les compteurs de synchronisation en Java. En effet, ces mécanismes ne sont pas présents nativement et on ne peut que faire des exercices "papier" pour aborder l'utilisation de ces compteurs. Pour la mise en oeuvre de ce mécanisme de synchronisation il faut :

- comptabiliser les accès aux méthodes automatiquement
- réaliser le contrôle d'accès aux méthodes (blocage et réveils) qui est basé sur des expressions booléennes.

Spécifications des compteurs de synchronisation

L'utilisation des compteurs de synchronisation est une méthode moins restrictive que les sections critiques et l'exclusion mutuelle. Ce mécanisme rend possible la gestion automatique des réveils et la mise à jour des variables de synchronisation. Autrement dit on effectue une séparation entre les opérations et la synchronisation, ainsi les threads peuvent évoluer indépendamment, excepté dans certains cas fixés par le programmeur, cela est comparable à des « rendez-vous ».

Nous comptabiliserons :

- le nombre d'invocations de requêtes depuis le début de l'exécution du programme : m.req.
- le nombre d'invocations qui ont été effectuées ou sont en cours d'exécution : m.aut.
- le nombre d'invocations qui ont été réalisés et terminés : m.term.
- le nombre d'invocations actuellement en attente : m.att.
- le nombre d'invocations en cours : m.act.

Pour être cohérent, ces compteurs vérifierons les prorpiétés suivantes :

- m.req>= m.aut >= m.term
- m.att = m.req - m.aut
- m.act = m.aut - m.term

Le processus d'exécution commence lors de l'invocation de m, on commence par mettre à jour et de réévaluer le compteur du nombres d 'invocations de requêtes depuis le début du programme (m.req) et ainsi que ceux qui sont en attente (m.att). Ensuite, on évalue si la requête vérifie les conditions d'accès, à l'issue duquel il y a blocage éventuel. Quand une invocation vérifie les conditions d'accès, on met à jour le nombre d'invocation de m qui ont été autorisée (m.aut) et le nombre d'invocation en attente (m.att). A la fin de l'exécution du corps de m, on met à jour le nombre d'invocations qui ont été réalisées (m.term) et ainsi que ceux qui sont en cours (m.act).

Les compteurs de synchronisation peuvent être utilisés pour spécifier des conditions de manière concise avant d'utiliser d'autres mécanismes.

Exemple sur les compteurs de synchronisation :
`Lecteur/Rédacteur:`

```
Class LecRecSimple implements MdC {

        private condition lire{
                (ecrire.act = 0)};
        private condition ecrire{
                (ecrire.act = 0 ) && ( lire.act = 0)};
        public elt lire(){...}
        public ecrire(elt e){...};
}
```

FIGURE 16 – *Lecteur/Rédacteur*

On impose une condition sur la lecture : le nombre des invocations de ecrire qui sont en cour doit étre égale à 0 pour pouvoir lire,cadr aucune ecriture en cours .

On impose aussi une condition sur l'ecriture :le nombre des invocations de lire et ecrire qui sont en cour doit étre égale à 0 pour pouvoir ecrire,cadr aucune ecriture et aucune lecture en cours .

```
Class LecRecPri implements MdC {
        private condition lire{
                ((ecrire.act = 0) && (ecrire.att = 0))};
        private condition ecrire{
                (ecrire.act = 0 ) && lire.act = 0)};
...
}
```

FIGURE 17 – *Lecteur/Rédacteur*

Ici on ajoute les priorités par rapport à l'exemple précedent. Pour la lecture, on impose une autre condition : le nombre des invocations de ecrire qui sont en attente doit étre égale à 0 pour pouvoir lire, c'est-à-dire aucune écriture en cours, et aucune requête d'écriture en attente.On donne la priorité à l'écriture.

5.2 Génération du code :

5.2.1 Utilisation de la compilation :

Le compilateur a pour but de transformer le code source dans un langage informatique de haut niveau d'abstraction, facilement compréhensible par l'humain vers un autre langage informatique de plus bas niveau (code cible).

Il permet aussi d'optimiser le code et va chercher à améliorer la vitesse d'éxecution ou réduire l'occupation mémoire du programme. On distingue plusieurs type de compilateur :

- AOT (Ahead-of-time) ou la compilation se fait avant le lancement de l'application.
- JIT (Just-in-Time) ou la compilation se fait à la volée

Dans notre cas, le code source est en langage java où l'utilisateur va utiliser les compteurs de synchronisation et le code cible est en langage java plus évoluée où nous modifierons le code source pour optimiser le parallélisme du programme. Le compilateur effectue d'abord le prétraitement, qui prend en charge la substitution de macro et de la compilation conditionnelle. Ensuite, il passe à l'analyse lexicale qui découpe le code source en petits morceaux appelés jetons (tokens). Chaque jeton est une unité atomique unique de la langue (unités lexicales ou lexèmes), par exemple un mot-clé, un identifiant ou un symbole.

Puis a lieu l'analyse syntaxique, impliquant l'analyse de la séquence jeton pour identifier la structure syntaxique du programme. Cette phase s'appuie généralement sur la construction d'un arbre d'analyse ; on remplace la séquence linéaire des jetons par une structure en arbre construit selon la grammaire formelle qui définit la syntaxe du langage. Pour cette étape, nous utiliserons un analyseur syntaxique du langage java.

Enfin, l'analyse sémantique est la phase durant laquelle le compilateur ajoute des informations sémantiques à l'arbre d'analyse et construit la table des symboles. Cette phase vérifie le type (vérification des erreurs de type), ou l'objet de liaison (associant variables et références de fonction avec leurs définitions), ou une tâche définie (toutes les variables locales doivent être initialisées avant utilisation), peut émettre des avertissements, ou rejeter des programmes incorrects.

A l'issue de ces étapes, nous pourrons transformer le code source en code intermédiaire et appliquer les techniques d'optimisation.

5.2.2 Utilisation de l'introspection :

L'introspection est la capacité d'un programme à examiner son propre état. Elle permet à un programme d'évoluer automatiquement en fonction des besoins et de l'environnement. Cette capacité existe dans le langage Java. L'API de réflexion permet l'introspection en donnant l'accès aux classes, à leurs champs, méthodes ou encore constructeurs, et à toutes les informations les caractérisant, même celles qu'on pensaient inaccessibles. Elle est aussi utilisée pour instancier des classes dynamiquement, dans la génération de code. La réflexion est également potentiellement intéressante pour l'écriture d'un générateur de code générique pour un ensemble de classe. Pour utiliser l'introspection, le paquetage java.lang.reflect offre plusieurs éléments indispensables.

Exemple : Une classe Secret avec un champ privé priv (String). L'exemple suivant va modifier ce champ privé :

```
void modifierChamp(Secret s, String val)
{
   Field f = s.getClass().getDeclaredField("priv");// un objet de type Field
contenant le champ priv
   f.setAccessible(true);
   f.set(s,val);
}
```

`getDeclaredField(nom)` : renvoie un objet Field correspondant au champ privé"nom" `getClass()` : une méthode de la classe Object qui renvoie un objet de type Class qui correspond à la classe de cet objet. `set(o,val)` : permet de modifier le contenu du champ d'un objet (ici o) en lui attribuant la valeur passée en second paramètre.

5.2.3 Utilisation de classe spécifiques : le Runtime

Le runtime comprend tout le code et toutes les données du langage dont un programme pourrait avoir besoin pendant son exécution. Chaque implémentation d'un langage ou d'une famille de langages fournit donc en général un runtime pour les programmes qu'elle va compiler et/ou interpréter.

Nous allons donc créer notre propre runtime qui gérent automatiquement les réveils qui sont implicites dans le code source et doivent être explicites dans le code généré par utilisation des mécanismes de synchronisation Java notify() et notifyAll().

Conclusion

Cette phase d'analyse nous a permis de mieux cerner le contexte du parallélisme, d'étendre nos connaissances dans la programmation parallèle et de nous familiariser avec les outils de synchronisation.

A la suite du raffinement de cette analyse, nous établirons une estimation de la planification des tâches.

Appendices

Quelques terminologies

`Tâche:` une portion de travail à exécuter sur un ordinateur, type un ensemble d'instructions d'un programme qui est exécuté sur un proc.

`Exécution séquentielle :` exécution d'un programme séquentiel, une étape à la fois.

`Exécution parallèle :` exécution d'un programme par plusieurs tâches (threads) ou processus (process), chaque tâche pouvant exécuter la même instruction (mode SIMD) ou une instruction différente (mode MIMD), à un instant donné.

`Mémoire partagée :` d'un point de vue matériel, se réfère à une machine dont tous les processeurs ont un accès direct à une mémoire commune (généralement via un bus). D'un point de vue modèle de programmation : toutes les tâches ont la même image mémoire et peuvent directement adresser et accéder au même emplacement mémoire logique, peu importe où il se trouve en mémoire physique..

`Mémoire distribuée :` d'un point de vue physique, basée sur un accès mémoire réseau pour une mémoire physique non commune. D'un point de vue modèle de programmation, les tâches ne peuvent voir que la mémoire de la machine locale et doivent effectuer des communications pour accéder à la mémoire d'une machine distante, sur laquelle d'autres tâches s'exécutent..

`Communications :` les tâches parallèles échangent des données, par différents moyens physiques : via un bus à mémoire partagée, via un réseau. Quelque soit la méthode employée, on parle de « communication ».

`Granularité :` mesure qualitative du rapport calcul / communications.

`Grain grossier (coarse) :` relativement bcp de calculs entre différentes communications.

`Grain fin (fine) :` relativement peu de calcul entre différentes communications.

`Accélération (Speedup) :` mesure de l'amélioration des performances due au parallélisme. (wall clock time of serial execution) / (wall clock time of parallel execution*nombre de processeurs)

`Scalability :` réfère à la capacité d'un système parallèle à fournir une augmentation de performance proportionnelle à l'augmentation du nombre de processeurs.

`Synchronisation :` la coordination des tâches en temps réel est souvent associée aux communications, elle est souvent implémentée en introduisant un point de synchronisation au-delà duquel la tâche ne peut continuer tant qu'une ou plusieurs autres tâches ne l'ont pas atteint.

`Linpack :` bibliothéque de fonctions pour l'algébre linéaire servant à classer les plus puissants superordinateurs du monde dans le TOP500.