UNIVERSITÉ DU QUÉBEC À MONTRÉAL

DÉTECTION IMMÉDIATE DES INTERBLOCAGES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MOURAD DAHMANE

JUILLET 2010

REMERCIEMENTS

Ce mémoire a bénéficié de la lumière miséricordieuse d'Allah et de l'aide précieuse de plusieurs personnes. Je tiens à remercier, en premier lieu, mon directeur de recherche, le professeur Étienne M. Gagnon, qui a encadré ce mémoire avec enthousiasme, et qui m'a conseillé efficacement durant toute la durée de ma recherche.

Je tiens à remercier tous les professeurs, les étudiants et les administrateurs de l'Université du Québec à Montréal qui m'ont soutenu durant mon processus d'apprentissage au département d'informatique.

Je tiens à remercier mon père Abdelkader Dahmane pour son suivi à chaque occasion, et à évoquer la mémoire de ma mère, à qui je dédie ce travail.

Je tiens à remercier mon épouse Bakhta Taieb Errahmani pour son soutien et pour les jours où j'ai pris de son temps pour finir ce mémoire ; à elle aussi je dédie ce travail ainsi que mon fils Abdellah et ma fille Asmaa.

Je tiens à remercier mes frères et sœurs, mes beaux-frères et belles-sœurs pour leur soutien, en leur dédiant également ce travail.

Je tiens à remercier mes amis qui m'aiment et qui demandent toujours de mes nouvelles.

Je tiens à remercier Véronique Boily pour son travail professionnel de correction.

Je tiens à remercier toute personne qui m'a aidé de proche ou de loin à réaliser ce travail.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

RÉSUMÉ

L'utilisation des fils d'exécution est devenue importante avec l'apparition des ordinateurs multiprocesseurs sur le marché. Les langages de programmation modernes, comme Java, offrent une souplesse dans l'écriture des programmes multifils d'exécution. Cette souplesse n'a pas éliminé les problèmes liés à la synchronisation des fils d'exécution. L'interblocage est l'un des problèmes majeurs dont la résolution nécessite temps et argent. La contribution principale de notre travail est la conception d'un algorithme efficace de détection immédiate des interblocages et l'implémentation de celui-ci dans une machine virtuelle libre.

Dans notre mémoire, nous parlons de la structure de forêt d'attente et de la façon de construire, à l'aide de cette structure, les relations d'acquisition et de libération des verrous par les fils d'exécution. Cette structure permet la détection immédiate de l'interblocage. Dans notre travail, le brisement de l'interblocage est réalisé par le soulèvement d'une exception qui pourra être interceptée, une fois l'interblocage détecté. Ce brisement permet aux développeurs de gérer les exceptions liées aux interblocages sans que leurs programmes s'arrêtent.

Notre expérimentation avec la version 1.13 de la machine virtuelle *SableVM* [GE02] et notre version améliorée en implémentant notre algorithme nous ont montré que notre détection immédiate a un coût nul dans une majorité de cas, et coûte une surcharge de temps d'exécution de 0,04 % à 0,2 % par rapport à la version 1.13 dans les pires des cas testés. Nous avons utilisé la suite de mesures (*benchmark*) *Ashes* et nos propres programmes de mesure de performance qui utilisent au maximum les fils d'exécution et les opérations d'acquisition et de libération des verrous. Nos programmes furent développés spécialement pour montrer les coûts additionnels de l'utilisation de notre algorithme.

Mots clés : fil d'exécution, synchronisation, interblocage, détection immédiate, brisement.

INTRODUCTION ET CONTRIBUTIONS

Introduction

La plupart des langages de programmation modernes sont des langages orientés objets tels que C++, Java, C# et d'autres, et la plupart de ces langages s'exécutent sur des machines virtuelles comme Java et C#. Les ordinateurs personnels, maintenant, sont dotés de processeurs puissants et même de processeurs doubles, ce qui signifie qu'un programme exécute plusieurs parties de son code en même temps sur plusieurs processeurs en parallélisme réel ou par partage de processeur. Nous connaissons ces parties du code sous plusieurs noms : les processus, les fils d'exécution (*threads*) ou les tâches (*tasks*). Le besoin de parallélisme est devenu énorme dans le monde de l'informatique. Internet est un réseau qui se base sur des serveurs Web, et chaque serveur Web exécute plusieurs fils d'exécution pour servir chaque session ouverte d'un client Internet (un explorateur d'Internet, une application qui utilise un service Web, etc.).

Ce parallélisme des fils d'exécution implique un partage des ressources entre les fils d'exécution ; ce partage crée une course vers les ressources pour les obtenir, et cette course cause des problèmes comme l'interblocage (*deadlock*), des données erronées, la famine des fils d'exécution (*starvation*). Imaginons qu'un interblocage survienne sur le serveur Web de Google ou de Microsoft, ou de n'importe quel serveur Web : cela implique que personne ne peut naviguer vers ces serveurs Web, et il en va ainsi pour plusieurs autres applications qui utilisent les fils d'exécution et qui peuvent entraîner une telle situation. Notre recherche porte sur un cas de ces problèmes liés aux fils d'exécution : l'interblocage.

Motivations

Plusieurs recherches ont été faites sur l'interblocage, surtout sur les systèmes d'exploitation, mais la plupart de ces algorithmes étaient complexes, lents et non efficaces. Beaucoup d'entre eux utilisent des théories mathématiques complexes, des recherches lentes dans des graphes avec des centaines de nœuds et d'arcs où ils trouvent qu'un problème existe, mais ils ne déterminent pas par quel fil d'exécution, ce qui signifie résoudre le problème d'une manière aléatoire.

Notre objectif est de résoudre le problème d'interblocage par un algorithme simple, efficace et rapide, simple à comprendre et à implémenter, efficace dans son exécution à trouver le problème et le fil d'exécution qui a causé l'interblocage, et rapide à résoudre celui-ci, sans forcément arrêter le fil d'exécution ou l'application, mais plutôt en laissant le choix de prendre des décisions dans le code du programme lui-même.

Intérêt de notre travail

Notre travail est intéressant sous plusieurs angles :

- Il permet de détecter l'interblocage et de le briser immédiatement.
- Il permet à des applications importantes, comme les serveurs Web, de rester en exécution tout le temps, de résoudre l'interblocage immédiatement et de revenir à leur exécution normale.
- Il permet de développer des applications complexes avec un support de détection et brisement de l'interblocage au lieu de développer des applications reconnues pour être libres d'interblocage.
- Il laisse le choix au programmeur de prendre des décisions dans son code quant à la façon de résoudre le problème d'interblocage.
- Il permet de développer des applications et des bibliothèques libres d'interblocage.
- Il minimise les coûts et le temps consacré au développement des applications multifils d'exécution, parce que l'interblocage sera trouvé en quelques minutes.

Contributions

Les contributions de ce mémoire sont les suivantes :

- La conception d'un algorithme efficace de détection immédiate des interblocages qui fonctionne en présence d'opérations de sychronisation concurrentes.
- Le coût nul de l'algorithme dans la majorité des cas concrets.
- La preuve de rectitude de l'algorithme.
- L'analyse de la complexité de l'algorithme développé.
- Le brisement des interblocages par le soulèvement d'une exception immédiatement lors de la tentative de création d'un interblocage.
- L'implémentation de l'algorithme dans la machine virtuelle *SableVM*.
- La mesure de la performance réelle de l'algorithme dans la machine virtuelle *SableVM*.
- Le développement de programmes de test de performance.

Organisation du mémoire

Notre mémoire est organisé comme suit :

Dans le chapitre 1, nous parlerons de la synchronisation en Java et de la façon dont s'écrit un code Java qui utilise les fils d'exécution et la synchronisation ; nous discuterons aussi des problèmes et des approches de synchronisation. Dans le chapitre 2, nous décrirons la synchronisation en code-octet et montrerons comment un programme Java qui utilise les fils d'exécution et la synchronisation est compilé en code-octet. Dans le chapitre 3, nous décrirons les problèmes d'interblocage et leurs détections en Java, nous citerons les conditions nécessaires pour un interblocage ainsi que les méthodes de détection. Dans le chapitre 4, nous discuterons de notre travail de détection immédiate et de brisement d'interblocage. Dans le chapitre 5, nous parlerons de l'expérimentation et de la comparaison des résultats de mesure de performance. Dans le chapitre 6, nous décrirons les travaux reliés au nôtre. Finalement, nous conclurons notre travail.

CHAPITRE I

SYNCHRONISATION EN JAVA

1.1 Introduction

La plupart d'entre nous, sur un système d'exploitation tel que Linux ou Windows, travaillons en multitâches. Nous écrivons du texte en même temps que nous écoutons de la musique ou que nous téléchargeons des fichiers de l'Internet ou d'un réseau tout en écrivant un programme, etc. Ces programmes en exécution s'appellent des processus. La plupart des systèmes d'exploitation offrent une gestion des processus et de multitâches.

Dans ce chapitre, nous allons voir des notions proches des notions de processus et de multitâches : ce sont les fils d'exécution et les multifils d'exécution. Nous avons utilisé quelques exemples de programmes Java pris de l'Internet (les références sont perdues) et nous avons modifié ces programmes pour les adapter à nos exemples donnés dans les différents chapitres.

Dans la section 1.2, nous définissons un fil d'exécution et sa relation à un processus et à d'autres fils d'exécution, ainsi que la façon de déterminer le code d'un fil d'exécution dans un programme Java, sa création et son arrêt. Dans la section 1.3, nous allons voir les problèmes liés à la synchronisation des fils d'exécution dans un programme Java. Finalement, dans la section 1.4, nous détaillons les approches utilisées en Java pour résoudre les problèmes de synchronisation des fils d'exécution.

1.2 Les fils d'exécution en Java

Un processus est un programme en exécution. Si un ordinateur est doté de plusieurs processeurs, ces processus s'exécutent en même temps : on parle d'un vrai parallélisme, tel que, dans l'exemple précédent, le processus tableur qui s'exécute sur un processeur et le processus lecteur de musique qui s'exécute sur un autre processeur ; mais si un ordinateur est doté d'un seul processeur (uniprocesseur), on parle d'un partage du temps processeur ou d'un faux parallélisme. Ainsi, le système d'exploitation alloue une tranche de temps du processeur pour permettre au processus de s'exécuter ; une fois que cette tranche de temps est terminée, le système d'exploitation suspend le processus en cours et alloue une tranche de temps du processeur à un autre processus pour s'exécuter, et ainsi de suite pour tous les processus. (Dans les paragraphes suivants, on utilisera le terme «parallélisme» pour parler du vrai parallélisme et du partage du temps processeur.)

Chaque processus a son propre code, sa mémoire et ses registres. Des processus peuvent communiquer entre eux (en utilisant des pipelines, des réseaux, etc.), mais cette communication est lente. Comme ils peuvent faire une demande au système d'exploitation pour partager un espace mémoire, le partage n'est pas automatique entre les processus.

Un processus est constitué d'un ou plusieurs blocs de codes. Dans un même processus, nous pouvons avoir aussi plusieurs blocs de codes qui s'exécutent en parallèle ; on appelle chaque bloc de code qui s'exécute en parallèle «un fil d'exécution».

1.2.1 Définition d'un fil d'exécution

Un fil d'exécution (*thread*) est une séquence de codes (bloc de code) qui s'exécute indépendamment et en concurrence avec d'autres séquences de code du même processus ou d'autres processus.

Un fil d'exécution s'exécute indépendamment, ce qui signifie qu'il est tout à fait indépendant, comme un processus dans son exécution ; il a ses propres registres processeur et son exécution est indépendante des autres fils d'exécution. Un fil d'exécution s'exécute

en concurrence avec d'autres fils d'exécution.

Dans un processus, nous pouvons avoir un ou plusieurs fils d'exécution. Ceux-ci ont le même espace d'adressage et ils ont accès à une mémoire globale commune, mais chacun des fils d'exécution a ses propres variables locales.

Le langage Java est parmi les langages de programmation modernes qui offrent une gestion avancée des fils d'exécution et qui rendent plus facile l'utilisation des fils d'exécution. La plupart de ceux qui écrivent un premier programme en Java affichant « Bonjour, tout le monde ! » ne savent peut-être pas qu'ils utilisent des fils d'exécution. La méthode principale `main()` d'un programme Java est exécutée en tant qu'un fil d'exécution, et plusieurs fils d'exécution sont exécutés automatiquement par la machine virtuelle Java sans que nous le sachions.

Dans un programme Java, on peut définir un ou plusieurs fils d'exécution. Ces fils d'exécution ont accès aux mêmes objets, y compris les variables statiques et d'instance, et chacun des fils d'exécution a ses propres variables locales.

La manipulation des fils d'exécution passe par deux étapes :

- La première étape consiste à spécifier le code dans un programme en Java qui va être le code du fil d'exécution.
- La deuxième étape consiste à créer une instance d'un fil d'exécution, après avoir spécifié le code selon la première étape, et ensuite à démarrer le fil d'exécution.

Nous détaillons la manipulation des fils d'exécution dans les deux sous-sections suivantes : dans la sous-section 1.2.2, nous allons voir comment spécifier la séquence du code d'un fil d'exécution dans un programme en Java ; ensuite, dans la sous-section 1.2.3, nous allons voir comment créer des fils d'exécution.

1.2.2 Spécification d'un fil d'exécution en Java

Cette étape de spécification d'un fil d'exécution constitue la première étape ; elle consiste à spécifier le code dans un programme en Java, qui va être le code du fil d'exécution

une fois le fil d'exécution exécuté. Comme nous l'avons vu dans la sous-section 1.2.1, le fil d'exécution est une séquence de code ou un bloc de code que nous devons spécifier en premier lieu dans la manipulation des fils d'exécution en Java.

La classe `java.lang.Thread` de la bibliothèque Java est la classe clé pour l'utilisation des fils d'exécution en Java. Cette classe contient les méthodes pour la manipulation des fils d'exécution en Java, comme le démarrage des fils d'exécution, la suspension et d'autres méthodes utiles (voir la sous-section 1.2.3).

Nous savons que Java ne permet pas un héritage multiple (ce qui signifie que nous ne pouvons pas hériter de plus d'une classe à la fois). Le problème qui se pose est celui-ci : comment utiliser les fils d'exécution en même temps que nous pouvons hériter d'une autre classe que la classe `java.lang.Thread`? La réponse est d'utiliser un constructeur parmi la liste des constructeurs de la classe `Thread` (Voir 1.2.3), ce constructeur qui accepte comme paramètre une classe qui implémente l'interface `java.lang.Runnable`. Cette interface définit une méthode abstraite `run()`.

En Java, nous spécifions le code d'un fil d'exécution par l'écriture du code de la méthode `run()` de la classe[1] :

- soit la classe qui hérite de la classe `java.lang.Thread`

```
public class MaClassThread extends Thread {

    public void run() {
        // code de la méthode
    }
}
```

- soit la classe qui implémente l'interface `java.lang.Runnable`

```
public class MaClassRunnable implements  Runnable {

    public void run() {
        // code de la méthode
    }
}
```

1. Les codes Java sont écrits selon la convention *Sun Microsystems*, 1997

La vie d'un fil d'exécution se situe entre le début et la fin d'exécution de la méthode `run()`. Pour le fil d'exécution principal qui est la méthode `main()`, sa vie se situe entre le début et la fin d'exécution de la méthode `main()`.

1.2.3 Création d'un fil d'exécution

La spécification du code du fil d'exécution ne signifie pas que notre fil d'exécution est en exécution. La spécification signifie que notre fil d'exécution a un code défini ; une fois que notre fil d'exécution est démarré pour s'exécuter, c'est ce code spécifié qui va être exécuté. Une deuxième remarque est que l'appel de la méthode `run()` ne signifie pas que notre fil d'exécution est en exécution ; l'appel de la méthode `run()` ne fait qu'exécuter le code dans le contexte actuel, c'est-à-dire dans le fil d'exécution courant (si la méthode `run()` est appelée à partir de la méthode `main()`, alors le fil d'exécution est celui de la méthode `main()`) et il ne va pas y avoir un nouveau fil d'exécution.

L'exécution du fil d'exécution passe par deux étapes :

- La première étape est d'instancier la classe du fil d'exécution (soit la classe dérivée de la classe `java.lang.Thread` ou la classe qui implémente l'interface `java.lang.Runnable` ; voir la sous-section 1.2.2).
- La deuxième étape est de démarrer le fil d'exécution pour qu'il s'exécute.

La première étape est une étape qu'on a l'habitude d'exécuter dans chaque programme Java ; c'est celle de créer une instance de la classe du fil d'exécution comme dans les deux exemples suivants :

```
// Exemple 1
MaClassThread mc = new MaClassThread();
```

Dans cet exemple, nous avons déclaré une variable `mc` de type `MaClassThread`, qui est une classe dérivée de la classe `java.lang.Thread`, ensuite nous avons instancié la classe en utilisant le mot clé `new`.

```
// Exemple 2
```

```
Runnable rc = new MaClassRunnable();
Thread th = new Thread(rc);
```

Dans ce deuxième exemple, qui est différent du premier, nous avons déclaré une variable `rc` de type `java.lang.Runnable` qui est une interface, ensuite nous avons instancié la classe `MaClassRunnable` en utilisant le mot clé `new`. La classe `MaClassRunnable` implémente l'interface `java.lang.Runnable`; enfin, nous avons déclaré une variable `th` de type `java.lang.Thread`, suivie par une instanciation de la classe `java.lang.Thread` en passant en paramètre la variable `rc` au constructeur de la classe.

La deuxième étape est de démarrer le fil d'exécution pour qu'il s'exécute ; cette étape est facile, il suffit seulement d'appeler la méthode `start()` de la classe `java.lang.Thread`, qui va créer le fil d'exécution et ensuite démarrer son exécution, comme dans les deux exemples suivants par rapport aux deux exemples précédents d'instanciation :

```
// Exemple 1
mc.start();
```

Voici le deuxième exemple :

```
// Exemple 2
th.start();
```

Nous allons présenter maintenant deux exemples complets, et voir comment les fils d'exécution sont exécutés :

1.2.3.1 Exemple `Bonjour.java`

```
class BonjourThread extends Thread {
    private String name;

    public BonjourThread(String name){
        this.name=name;
    }
```

```
    public void run(){
        for (int i = 1; i <= 4; i++) {
            System.out.println(this.name + ", " + i);
        }
    }
}

class Bonjour {

    public static void main(String[] args) {
        BonjourThread t1, t2;

        System.out.println("Début de la méthode main()!");
        t1 = new BonjourThread("Mourad");
        t2 = new BonjourThread("Etienne");
        t1.start();
        t2.start();
        System.out.println("La fin de la méthode main()!");
    }
}
```

Cet exemple permet de démarrer les deux fils d'exécution `t1` et `t2`. Chaque fil d'exécution exécute le code de la méthode `run()` de la classe `BonjourThread`. Le code de la méthode `run()` consiste à afficher quatre fois un message avec deux paramètres : la variable privée `name` et le compteur `i`. Dans la méthode `main()`, nous avons déclaré deux variables `t1` et `t2` de type `BonjourThread`, de même que dans l'instanciation de `t1` et `t2`, nous avons passé un paramètre de type chaîne de caractères. Par la suite, nous avons exécuté la méthode `start()` des deux instances `t1` et `t2`. Voyons maintenant deux exécutions de ce programme :

Exécution 1

```
Début de la méthode main()!
La fin de la méthode main()!
Etienne, 1
Mourad, 1
Etienne, 2
Mourad, 2
Etienne, 3
Etienne, 4
Mourad, 3
Mourad, 4
```

Exécution 2

```
Début de la méthode main()!
Mourad, 1
Mourad, 2
La fin de la méthode main()!
Etienne, 1
Etienne, 2
Mourad, 3
Mourad, 4
Etienne, 3
Etienne, 4
```

Comme vous le remarquez, dans la première exécution, l'exécution de la méthode `main()` s'est terminée avant le début d'exécution des fils d'exécution `t1` et `t2`, alors que dans la deuxième, la méthode `main()` s'est terminée après que le fil d'exécution `t1` ait commencé à s'exécuter. La deuxième remarque sur ces deux exécutions est que l'ordre d'exécution des fils d'exécution `t1` et `t2` est aléatoire ; dans la première exécution, c'était le fil d'exécution `t1` qui a commencé le premier, alors que dans la deuxième, c'était le fil d'exécution `t2` qui a commencé en premier.

En Java, nous pouvons donner des priorités aux fils d'exécution en utilisant la méthode `setPriority` de la classe `java.lang.Thread` pour donner à un fil d'exécution plus de priorité de s'exécuter avant d'autres fils d'exécution. Les valeurs des priorités sont entre deux constantes : `MIN_PRIORITY` et `MAX_PRIORITY`. La priorité par défaut est `NORM_PRIORITY`.

1.2.3.2 Exemple `CompteBancaire.java`

Dans ce deuxième exemple, nous allons voir la procédure de retrait d'argent d'un compte bancaire. Le programme est constitué de deux classes essentielles, la classe `CompteClient`, qui contient toutes les informations concernant un compte client, comme le solde *balance* du compte client, plus une méthode `retirer()`, qui permet de retirer de l'argent d'un compte client. La deuxième classe est `RetraitCompteBancaireThread`, qui implémente l'interface `java.lang.Runnable` ; cette classe nous permet de lancer des

```
class Bonjour {

    public static void main(String[] args) {
         BonjourThread t1, t2;

         System.out.println("Début de la méthode main()!");
         t1 = new BonjourThread("Mourad");
         t2 = new BonjourThread("Etienne");
         t1.setPriority(MAX_PRIORITY);
         t1.start();
         t2.start();
         System.out.println("La fin de la méthode main()!");
    }
}
```

fils d'exécution de retrait d'argent à partir d'un compte client.

```
class CompteClient {
    int numeroCompte;
    double balance;

    public CompteClient(int numeroCompte) {
        this.numeroCompte = numeroCompte;
        this.balance = 500;
    }

    public double getBalance() { return balance; }

    public void retirer(double somme) {
        if (balance > somme) {
            balance -= somme;
        }
    }
}

class RetraitCompteBancaireThread implements Runnable {
    CompteClient client;
    double somme;
    String nomClient;

    public RetraitCompteBancaireThread(String nomClient, CompteClient client,
                                       double somme) {
        this.nomClient = nomClient;
        this.client = client;
        this.somme = somme;
    }
```

```
    public void run() {
        System.out.println(this.nomClient +"(avant), " + client.getBalance());
        client.retirer(somme);
        System.out.println(this.nomClient +"(après), "+ client.getBalance());
    }
}

class CompteBancaire extends Thread {

    public static void main(String[] args) {
        Runnable A, B;

        CompteClient client = new CompteClient(1);
        A = new RetraitCompteBancaireThread("A", client, 100);
        B = new RetraitCompteBancaireThread("B", client, 200);
        new Thread(A).start();
        new Thread(B).start();
    }
}
```

Voyons deux exemples d'exécution :

```
A(avant), 500.0
B(avant), 500.0
A(après), 400.0
B(après), 200.0
```

Dans ce premier exemple, les deux fils d'exécution `A` et `B` sont exécutés en concurrence pour faire des retraits de 100 et 200 à partir du compte client numéro 1. Le solde final *balance* est de 200.

Voyons maintenant une deuxième exécution :

```
A(avant), 500.0
B(avant), 500.0
A(après), 400.0
B(après), 300.0
```

Nous remarquons que le compte 1 contient un solde initial de 500 alors que deux retraits de 100 et 200 sont faits ; normalement, le solde final *balance* devrait être de 200, mais

l'exécution nous montre que le solde final est de 300. Nous répondons à ce problème dans la section 1.3

1.2.4 Arrêt d'un fil d'exécution

Dans les exemples précédents, nous avons vu qu'un fil d'exécution se termine une fois que sa méthode `run()` est terminée. Voici les autres situations où un fil d'exécution peut se terminer :

- Un fil d'exécution est interrompu par une exception non capturée.
- Sa méthode `stop()` est appelée. Cette méthode était désaprouvée en Java 2, parce qu'elle causait des problèmes graves. Supposons par exemple qu'on arrête un fil d'exécution qui est en train d'écrire dans un fichier ; cette action peut corrompre le fichier. La méthode `run()` doit être conçue pour se terminer, par exemple en vérifiant une variable booléenne pour finir la méthode `run()`.

Il y a d'autres méthodes qui permettent de suspendre et de redémarrer un fil d'exécution (les méthodes `suspend()` et `resume()`).

Pour plus d'informations, référez-vous à un livre de référence en Java [LD00].

1.2.5 Besoin des fils d'exécution

Du point de vue du programmeur, il est essentiel d'utiliser des fils d'exécution pour augmenter la vitesse d'exécution des programmes, surtout si l'ordinateur est doté de plusieurs processeurs. Nous pouvons augmenter la vitesse d'exécution de deux, trois ou plus par rapport à une exécution séquentielle d'un programme.

Donnons un exemple pour voir la différence. Un programme qui reconnaît les rues d'une grande ville à partir d'une image satellite. La reconnaissance des rues va prendre des heures et des heures avant qu'elle finisse si le programme s'exécute séquentiellement, même si l'ordinateur est doté de plusieurs processeurs. Mais l'utilisation d'un programme multifils d'exécution est une solution parfaite. Ainsi, chaque fil d'exécution traite une

partie de l'image. Avec un ordinateur multiprocesseur, l'exécution va être plus rapide par rapport à une exécution séquentielle.

Un deuxième point avantageux en ce qui concerne les fils d'exécution est celui des interfaces graphiques (*GUI = Graphic User Interface*). Une interface graphique doit réagir aux clics de la souris et aux frappes du clavier en même temps qu'un traitement en arrière se fait, sinon le programme sera bloqué jusqu'à ce que le traitement en cours soit fini. Les fils d'exécution permettent que notre interface graphique interagisse avec l'utilisateur.

Dans une *applet* (une *applet* est un programme Java qui s'exécute sur un explorateur d'Internet), il y a un fil d'exécution pour le chargement des images, un autre pour le rafraîchissement de l'écran, un autre pour la production du son, et ainsi de suite. Les fils d'exécution nous permettent d'accomplir plusieurs tâches en même temps sans avoir besoin, pour des tâches indépendantes, de faire la file, une tâche après l'autre. D'autres avantages sont liés à l'utilisation des fils d'exécution dans les programmes décisionnels (comme le jeu d'échecs, où nous créons un fil d'exécution pour chaque déplacement d'une pièce et où nous devons suivre les conséquences de ce déplacement pour prendre une décision sur le déplacement à faire).

1.3 Problèmes de synchronisation

Plusieurs fils d'exécution s'exécutent en concurrence sur des ressources partagées ; cela peut causer des problèmes de conflit d'accès aux ressources, ce qui peut corrompre celles-ci, comme nous l'avons vu dans l'exemple 1.2.3.2 du compte bancaire, dans le deuxième exemple d'exécution où le solde final du compte client était corrompu. Voyons un autre exemple de l'impression d'un livre, ensuite nous expliquerons le problème.

1.3.0.1 Exemple `DocumentPrinter.java`

```
import java.util.*;

class Printer {
```

```
    public void print(String texte) {
        System.out.print(texte + ";");
    }
}

class Page {
    private String page;

    public Page(String page) {
        this.page=page;
    }

    public void print(Printer printer) {
        printer.print(page);
    }
}

class Livre implements Runnable {
    private ArrayList<Page> pages;
    private Printer printer;

    public Livre(Printer printer) {
        pages=new ArrayList<Page>();
        this.printer=printer;
    }

    public void add(Page page) {
        pages.add(page);
    }

    public void print() {
        for (int i = 0; i < pages.size(); i++) {
            pages.get(i).print(printer);
        }
    }

    public void run() {
        print();
    }
}

class DocumentPrinter {

    public static void main(String[] args) {
        Printer printer = new Printer();
        Livre l1 = new Livre(printer);
        Livre l2 = new Livre(printer);
        l1.add(new Page("L1P1"));
        l1.add(new Page("L1P2"));
```

```
        l2.add(new Page("L2P1"));
        l2.add(new Page("L2P2"));
        new Thread(l1).start();
        new Thread(l2).start();
    }
}
```

L'exemple consiste en une classe `Printer`, qui permet d'imprimer un texte ; nous avons utilisé l'écran pour une simulation, et les deux classes `Page` et `Livre` pour la construction du texte d'un livre ; un livre est constitué de plusieurs pages. La classe `Livre` implémente l'interface `Runnable`, que nous utilisons pour lancer des fils d'exécution d'impression des pages. Dans l'exemple 1.3.0.1, tous les fils d'exécution utilisent le même pilote d'impression (la variable `printer` de type `Printer`).

Voyons deux exemples d'exécution :

Exécution 1

```
L1P1;L1P2;L2P1;L2P2;
```

Exécution 2

```
L1P1;L2P1;L2P2;L1P2;
```

Dans la première exécution, l'impression des deux livres était correcte dans l'ordre, mais dans la deuxième exécution, il y a eu un mélange dans l'impression, de sorte que la page 1 du premier livre était imprimée en premier, suivies des 2 pages du deuxième livre, et à la fin on retrouvait la page 2 du premier livre. Pour des documents d'une centaine de pages, il n'y a rien d'amusant à retirer les pages d'un livre contenant des milliers de pages qui se trouvent sur l'imprimante...

Revenons à l'exemple 1.2.3.2 du compte bancaire pour voir où est le problème ; ce dernier a eu lieu au niveau de la méthode `retirer` de la classe `CompteClient`

```
public void retirer(double somme) {
    if (balance > somme) {
```

```
        balance -= somme;
    }
}
```

Il apparaît spécialement sur la ligne `balance-=somme`. Voyons par la figure 1.1 l'exécution des deux fils d'exécution.

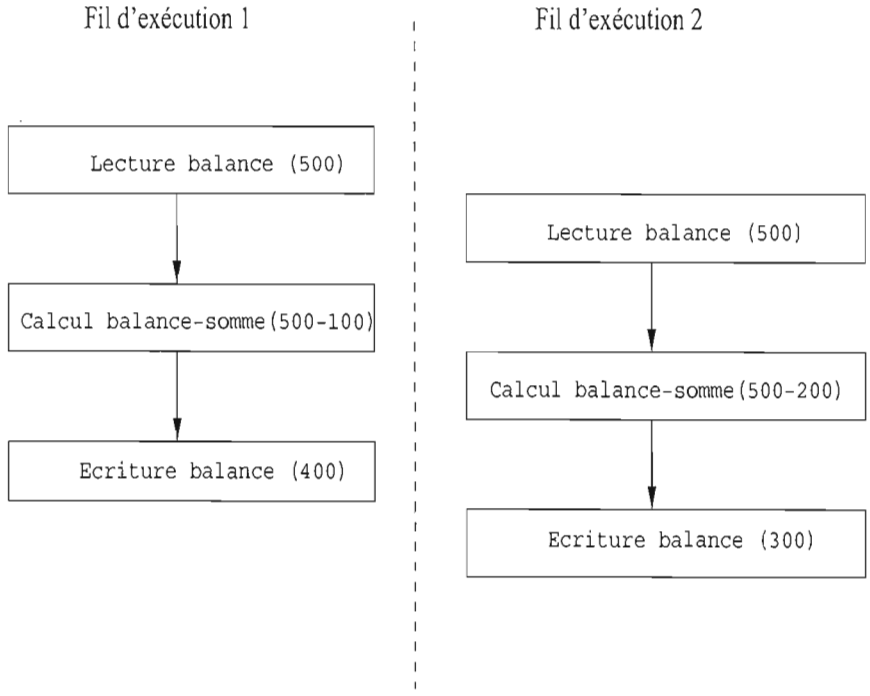


Figure 1.1 Exemple d'exécution du programme CompteClient.java

Voyons en détail l'instruction `balance-=somme` (`balance=balance-somme`). Cette instruction est exécutée en plusieurs instructions machine :

- Lecture de la valeur de la variable `balance`.
- Soustraction de la valeur `somme` de la valeur `balance`.
- Écriture du résultat dans la variable `balance`.

Nous remarquons sur la figure 1.1 que l'opération de calcul du nouveau solde *balance* n'est pas atomique, c'est-à-dire que plusieurs fils d'exécution exécutent la même

opération en concurrence, et c'est pour cette raison que le solde final *balance* est erroné : l'opération n'est pas synchronisée.

Le même problème de synchronisation s'est passé avec l'exemple de l'impression d'un livre. L'utilisation du pilote d'impression (la variable `printer`) n'était pas synchronisée entre les fils d'exécution.

Pour résoudre ce problème, il faut synchroniser les fils d'exécution ; ainsi, quand un fil d'exécution utilise le pilote, les autres fils d'exécution attendent jusqu'à ce qu'il ait terminé son impression pour qu'un autre fil d'exécution en attente utilise le pilote, alors que les autres attendent, et ainsi de suite. Dans la section 1.4, nous allons voir comment synchroniser les fils d'exécution en Java.

1.4 Approches de synchronisation

Dans la section 1.3, nous avons vu les problèmes liés à l'utilisation des ressources partagées et comment cette utilisation peut causer des résultats erronés. Les fils d'exécution qui utilisent la même ressource en même temps conduisent, si l'accès à la ressource n'est pas synchronisée, à des résultats non attendus.

La synchronisation permet qu'un seul fil d'exécution utilise une ressource partagée à la fois, de sorte que les autres fils d'exécution qui veulent utiliser cette ressource attendent jusqu'à ce que le fil d'exécution en cours libère cette ressource partagée.

En Java, il y a deux approches de synchronisation :

- L'exclusion mutuelle.
- L'attente et la notification.

1.4.1 Exclusion mutuelle

Avant d'expliquer l'exclusion mutuelle, nous définissons quelques notions :

- Une section critique est un bloc de code qu'un seul fil d'exécution doit exécuter à

la fois.

- Chaque classe en Java a une instance unique de type `java.lang.Class`.
- Chaque objet a son propre verrou. Ce verrou est utilisé pour verrouiller l'objet contre les utilisations concurrentielles de l'objet en même temps.

L'exclusion mutuelle signifie qu'un fil d'exécution empêche un autre fil d'exécution d'utiliser le même objet en même temps que lui, et qu'il essaie de verrouiller l'objet avant d'exécuter une section critique. Si l'objet est libre, le fil d'exécution acquiert le verrou de l'objet, exécute le code de la section critique et, à la fin, libère le verrou de l'objet. Tout autre fil d'exécution qui essaie de verrouiller l'objet pendant que l'objet est verrouillé va être mis en attente. Une fois que le fil d'exécution qui détient le verrou libère le verrou, un des fils d'exécution en attente sera réveillé pour essayer d'acquérir ce dernier.

En Java, il y a deux façons d'obtenir l'exclusion mutuelle :

1.4.1.1 Synchronisation des méthodes

La synchronisation d'une méthode signifie que toute une méthode, du début jusqu'à la fin de celle-ci, est une section critique. L'objet qui va être verrouillé en appelant cette méthode est :

- Si la méthode est une méthode virtuelle, alors c'est l'objet sur lequel la méthode a été appelée.
- Si la méthode est une méthode statique, alors c'est l'instance unique de la classe de cette méthode.

L'objet va être verrouillé avant l'exécution de la méthode et va être libéré après la fin d'exécution de la méthode.

Il suffit d'ajouter le mot réservé «`synchronized`» comme descripteur d'une méthode virtuelle ou statique.

Exemple :

```
synchronized void f() {
    // code de la méthode
}
```

Voyons maintenant la solution du problème de l'exemple 1.2.3.2 du compte bancaire (`CompteBancaire.java`). Reprenons seulement la méthode `retirer` de la classe `CompteClient` :

```
class CompteClient {

    public synchronized void retirer(double somme) {
        if (balance > somme) {
            balance -= somme;
        }
    }
}
```

Nous avons ajouté le mot clé `synchronized` comme descripteur de la méthode `retirer`, ce qui rend cette méthode synchronisée. Voyons maintenant la figure 1.2

1.4.1.2 Synchronisation des blocs

L'inconvénient de la synchronisation d'une méthode est que nous verrouillons tout l'objet pour utiliser une méthode synchronisée, ce qui signifie que si nous avons deux méthodes synchronisées, alors deux fils d'exécution qui utilisent le même objet ne peuvent pas exécuter les deux méthodes en même temps ; c'est-à-dire que si un des fils d'exécution exécute une des deux méthodes, il va verrouiller l'objet en premier, et l'autre fil d'exécution devra attendre jusqu'à ce que le premier termine l'exécution de sa méthode. Quand le premier fil d'exécution termine l'exécution de sa méthode, l'autre fil d'exécution verrouille le même objet et exécute l'autre méthode.

Une autre méthode de synchronisation est offerte en Java : c'est la synchronisation d'un bloc. Celle-ci consiste à synchroniser un bloc de code, qui n'est pas forcément tout le code d'une méthode, mais seulement une partie de la méthode. Dans ce type

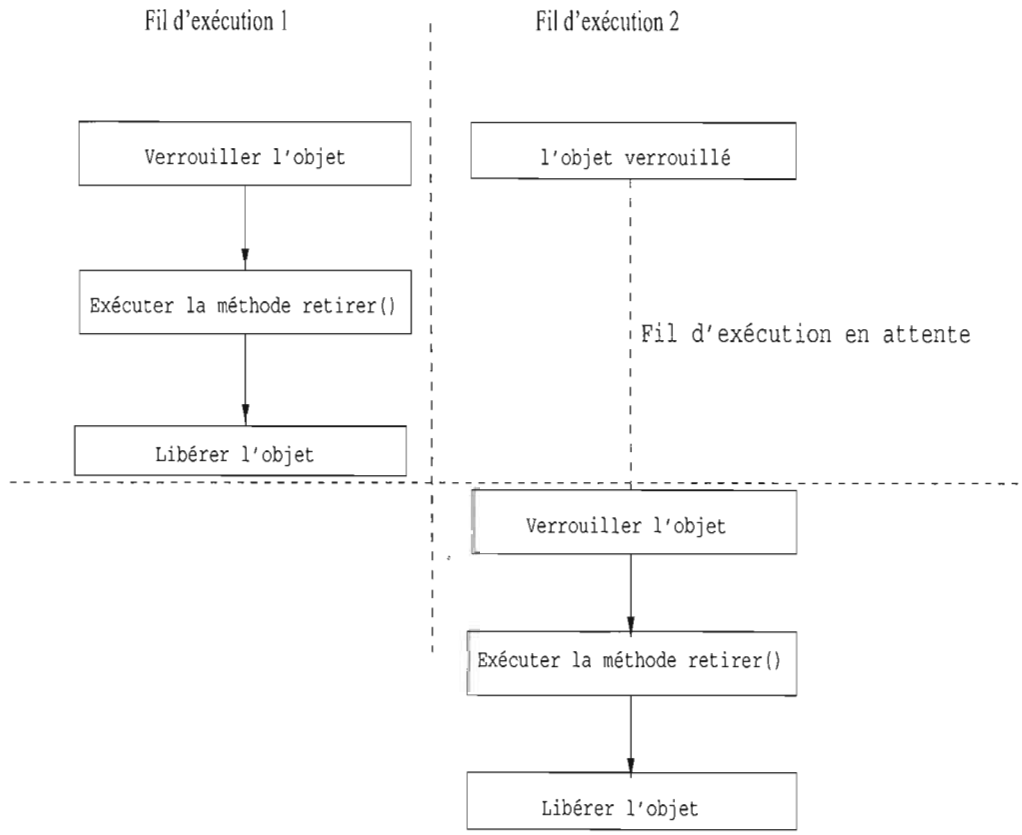


Figure 1.2 Exécution synchronisée du programme CompteClient.java

de synchronisation, nous utilisons l'objet que nous voulions verrouiller. Pour verrouiller l'objet courant, nous utilisons l'objet `this`, qui représente l'objet en cours.

Il suffit d'utiliser l'instruction « `synchronized(paramètre) code` » avec un paramètre qui représente une référence vers un objet, et le code est délimité par un bloc. Le fil d'exécution qui va exécuter le bloc va verrouiller l'objet en paramètre.

Exemple :

```
public class MaClasse {
    Object obj= new Object();

    void f() {
        synchronized(obj) {
            // code du bloc
        }
    }
}
```

Exemple utilisant l'objet `this` :

```
public class MaClasse {

    void f() {
        synchronized(this) {
            // code du bloc
        }
    }
}
```

Voyons maintenant la solution du problème de l'exemple 1.3.0.1 de l'impression d'un livre (`DocumentPrinter.java`). Reprenons seulement la méthode `print` de la classe `Livre` :

```
class Livre implements Runnable {

    public void print() {
        synchronized(printer) {
            for (int i = 0; i < pages.size(); i++) {
                pages.get(i).print(printer);
```

```
            }
        }
    }
}
```

Nous avons ajouté l'instruction `synchronized(printer)` avec comme paramètre l'objet `printer`, qui est utilisé par plusieurs fils d'exécution pour imprimer des livres. Nous n'avons pas délimité toute la méthode par l'instruction `synchronized`, mais seulement la partie du code qui utilise l'objet `printer`. De cette manière, tout livre va être imprimé au complet avant qu'un autre livre commence à s'imprimer.

1.4.2 Attente et notification

Dans l'approche de l'exclusion mutuelle, les fils d'exécution ne communiquent pas entre eux, mais ils verrouillent et libèrent les objets en concurrence. Java offre un moyen de communication entre les fils d'exécution : c'est l'approche d'attente et de notification.

Cette approche consiste en ce que deux ou plusieurs fils d'exécution collaborent entre eux dans un but commun, et qu'ils communiquent en utilisant l'attente et la notification. C'est le principe du producteur et du consommateur. Des fils d'exécution produisent des informations, et des fils d'exécution consomment ces informations. Les consommateurs attendront jusqu'à ce que les informations soient disponibles ; une fois les informations prêtes, les producteurs informent les consommateurs de la disponibilité des informations.

En Java, nous utilisons les méthodes `wait`, `notify` et `notifyAll` de la classe `java.lang.Object` pour l'attente et la notification. Un fil d'exécution en attente d'une condition va exécuter la méthode `wait()`, qui va le mettre en attente jusqu'à ce qu'un autre fil d'exécution lance une notification, par l'exécution de la méthode `notify` ou `notifyAll`, pour le réveiller. La méthode `notify` notifie un fil d'exécution quelconque parmi les fils d'exécution en attente de l'objet de la notification, alors que `notifyAll` va réveiller tous les fils d'exécution en attente de l'objet. Le fil d'exécution réveillé ou les fils d'exécution réveillés vont essayer d'acquérir le verrou de l'objet si le verrou est libre. Le fil d'exécution

qui acquiert le verrou de l'objet va continuer son exécution à partir du point suivant le `wait` qui le met en attente.

Pour qu'un fil d'exécution exécute les méthodes `wait`, `notify` et `notifyAll`, il faut que le fil d'exécution courant ait acquis le verrou de l'objet avant l'exécution.

Voyons un exemple d'attente et de notification :

1.4.2.1 Exemple `ProducteurConsommateur.java`

```
class Buffer {
    private int nombre;
    private String[] chaines;

    public Buffer() {
        nombre = -1;
        chaines = new String[5];
    }

    synchronized void put(String value) {
        while (nombre >= 5) {
            try {
                wait();
            } catch(InterruptedException e) { }
        }

        nombre++;
        chaines[nombre] = value;
        System.out.println("Insérer: " + value);
        notify();
    }

    synchronized String get() {
        while (nombre < 0) {
            try {
                wait();
            } catch(InterruptedException e) { }
        }

        String chaine = chaines[nombre--];
        System.out.println("Retirer: "+ chaine);
        notify();
        return chaine;
    }
}
```

```
class Producteur implements Runnable {
    private Buffer buffer;

    public Producteur(Buffer buffer) {
        this.buffer=buffer;
        new Thread(this, "Producteur").start();
    }

    public void run() {
        int i=0;
        while (true) {
            buffer.put("" + i);
            i++;
        }
    }
}

class Consommateur implements Runnable {
    Buffer buffer;

    public Consommateur(Buffer buffer) {
        this.buffer=buffer;
        new Thread(this, "Consommateur").start();
    }

    public void run() {
        String chaine;
        while (true) {
            chaine=buffer.get();
        }
    }
}

class ProducteurConsommateur {

    public static void main(String[] args) {
        Buffer buffer=new Buffer();
        new Producteur(buffer);
        new Consommateur(buffer);
    }
}
```

Dans cet exemple, nous avons une classe `Buffer` qui sert à un tampon de 5 cases pour déposer et retirer des chaînes de caractères. Cette classe est composée de deux méthodes essentielles, la méthode `put` et la méthode `get`. La méthode synchronisée `put` sert à insérer une chaîne de caractères dans le tableau de chaînes de dix cases ; si les 5

cases sont remplies, alors le fil d'exécution en cours va se mettre en attente (en appelant la méthode `wait()` de l'objet en cours). Nous vérifions si le tableau est rempli par la variable privée `nombre`, qui stocke le nombre des cases remplies ; si ce nombre est égal ou supérieur à 5, alors le tableau est rempli à chaque insertion d'une nouvelle chaîne de caractères, et la variable `nombre` est incrémentée. Alors que la méthode `get` sert à retirer des chaînes de caractères du tableau à chaque retrait, la variable `nombre` est décrémentée. Si la variable `nombre` est inférieure à zéro, alors le fil d'exécution en cours va se mettre en attente.

Notre exemple est constitué aussi de deux autres classes, `Producteur` et `Consommateur`, qui implémentent l'interface `Runnable`. Les deux classes utilisent le même objet `buffer` de type `Buffer`. La classe `Producteur` produit des chaînes de caractères dans l'objet `buffer`, et la classe `Consommateur` consomme des chaînes de caractères de l'objet `buffer`. La méthode `put` notifie (en appelant la méthode `notify()` de l'objet en cours, c'est-à-dire l'objet `buffer`) les consommateurs quand elle dépose une chaîne de caractères dans le tampon, alors que la méthode `get` notifie les producteurs quand elle retire une chaîne de caractères du tampon.

Exemple d'exécution

```
Insérer: 0
Retirer: 0
Insérer: 1
Retirer: 1
Insérer: 2
Insérer: 3
Retirer: 3
Retirer: 2
Insérer: 4
Retirer: 4
```

1.5 Conclusion

Dans ce chapitre, nous avons défini ce qu'est un fil d'exécution et en quoi il diffère par rapport à un processus. Ensuite, nous avons vu comment créer un fil d'exécution en

Java, ce qui a été suivi par des exemples sur la façon d'écrire un fil d'exécution dans un programme Java en utilisant la classe `Thread` ou l'interface `Runnable`. Après, nous avons mentionné la nécessité des fils d'exécution. Nous avons parlé aussi des problèmes liés à la concurrence entre les fils d'exécution et la nécessité de la synchronisation. Nous avons vu aussi les deux approches utilisées en Java : l'exclusion mutuelle ainsi que l'attente et la notification.

CHAPITRE II

SYNCHRONISATION EN CODE-OCTET

2.1 Introduction

La compilation d'un programme Java est la deuxième étape après l'écriture du programme. Cette étape nous permet de vérifier les erreurs dans notre programme. Le résultat de cette compilation est un ou plusieurs fichiers classe (extension `.class`); chaque classe est générée dans un fichier. La compilation se fait par un compilateur Java (comme `javac.exe` ou tout autre compilateur).

Le fichier classe est constitué d'une suite d'octets; ces octets constituent la structure d'un programme Java en code-octet.

Nous avons vu dans le chapitre 2 comment écrire un programme en Java qui implémente de la synchronisation entre les fils d'exécution. Nous allons voir dans cette section comment la synchronisation est traitée en code-octet.

Dans les deux sections 2.2 et 2.3, nous présenterons les deux approches de synchronisation : l'exclusion mutuelle et l'attente et la notification. Ensuite dans la section 2.4, nous représenterons la sychronisation dans la machine virtuelle Java.

2.2 Exclusion mutuelle en code-octet

Dans la section 1.4.1, nous avons vu la définition de l'exclusion mutuelle et les deux méthodes d'exclusion utilisées en Java. La synchronisation d'un bloc est traitée en code-

octet par deux *opcodes*, mais la synchronisation d'une méthode est traitée par la machine virtuelle elle-même.

2.2.1 Bloc synchronisé

Les *opcodes* `moniterenter` et `moniterexit` sont utilisés en code-octet pour délimiter le bloc synchronisé. Avant chaque exécution du bloc synchronisé, l'instruction `moniterenter` est exécutée et, à la fin d'exécution du bloc synchronisé, l'instruction `moniterexit` est exécutée, quel que soit l'état d'exécution du bloc synchronisé : soit par une fin d'exécution normale du bloc synchronisé, soit par une exception lancée pendant l'exécution du bloc synchronisé. Le terme «moniteur» d'un objet est l'équivalent du terme «verrou» d'un objet.

Prenons un exemple simple pour voir comment le code-octet est généré pour un bloc synchronisé :

```
class BlocSynchronized extends Thread {
    Object lock=new Object();

    public void run() {
        synchronized(lock) {
            // le code du bloc synchronisé
        }
    }
}
```

Voyons le code-octet généré pour le bloc synchronisé `synchronized(lock)`.

```
1:   getfield        #4; // créer une référence vers l'objet lock
2:   dup                 // dupliquer la référence vers l'objet lock
3:   astore_1            // stocker la référence vers l'objet lock
4:   monitorenter        // le début du bloc synchronisé
5:   ...                 // le code du bloc synchronisé
6:   aload_1             // empiler la référence stockée (lock)
7:   monitorexit         // la fin du bloc synchronisé
```

Dans la ligne 4, le bloc commence par une instruction `moniterenter`. L'*opcode* `moniterenter` verrouille l'objet `lock` s'il est libre, sinon le fil d'exécution en cours se met en

attente jusqu'à ce que l'objet `lock` soit libre. `moniterenter` dépile une référence vers l'objet, qui va être verrouillé, à partir de la pile ; dans notre exemple, c'est l'objet `lock`. Dans la ligne 1, l'*opcode* `getfield` crée une référence vers l'objet `lock` et la met au sommet de la pile ; ensuite, l'*opcode* `dup` duplique le sommet de la pile et l'empile aussi. Dans la ligne 3, l'*opcode* `astore_1` stocke la référence vers l'objet `lock` dans la variable locale d'index 1. La ligne 5 (plusieurs lignes de code) représente le code-octet du corps du bloc synchronisé. Dans les lignes 6 et 7, `aload_1` empile la référence vers l'objet `lock` (stockée dans la variable locale d'index 1) dans la pile, ensuite l'*opcode* `moniterexit` déverrouille l'objet `lock`.

2.2.2 Méthode synchronisée

Le code-octet généré pour une méthode synchronisée est différent de celui d'un bloc synchronisé. Il ne présente pas de différence par rapport à une méthode non synchronisée, outre l'invocation de la méthode à l'intérieur de la machine virtuelle de Java. Une méthode synchronisée est connue par un drapeau `ACC_SYNCHRONIZED` mis à 1, alors qu'avec une méthode non synchronisée, ce drapeau est à zéro. Quand un fil d'exécution invoque une méthode synchronisée, il verrouille l'objet de la méthode, invoque celle-ci et déverrouille l'objet de la méthode synchronisée une fois que l'invocation est complétée.

Voyons un exemple de programme Java et son code-octet généré pour une méthode synchronisée et son invocation :

```
class MethodSynchronized extends Thread {

    public synchronized void somme(){
        // code de la méthode synchronisée
    }

    public void run() {
        somme();
    }
}
```

Voici maintenant le code-octet généré pour la méthode synchronisée `somme` et son invo-

cation dans la méthode run :

```
public synchronized void somme();
   1:   return  // corps de la méthode somme.

public void run();
   1:   aload_0
   2:   invokevirtual   #2; // invocation de la méthode somme.
   3:   return
```

Le code-octet généré sera le même si la méthode somme n'était pas synchronisée, ainsi que l'invocation de la méthode somme dans la méthode run.

2.3 Attente et notification

Le code-octet généré pour l'attente et la notification n'est autre que l'invocation d'une méthode de classe `java.lang.Object`. Les méthodes `wait`, `notify` et `notifyAll` sont des méthodes de la classe `java.lang.Object`. Ces méthodes sont invoquées à l'intérieur des méthodes synchronisées, et nous avons vu dans la section précédente (2.2.2) le code-octet généré pour une méthode synchronisée.

Voyons un exemple de programme Java et son code-octet généré pour l'utilisation de la méthode `wait` :

```
class WaitNotify extends Thread {
    Object lock = new Object();

    public void run() {
        try {
            synchronized (lock) {
                lock.wait();
            }
        } catch (InterruptedException ex) { }
    }
}
```

Voici maintenant le code-octet généré pour la méthode run :

```
1:   aload_0
2:   getfield        #4; // créer une référence vers l'objet lock
```

```
3:   dup                // dupliquer la référence vers l'objet lock
4:   astore_1           // stocker la référence vers l'objet lock
5:   monitorenter       // le début du bloc synchronisé
6:   aload_0
7:   getfield       #4; // créer une référence vers l'objet lock
8:   invokevirtual  #5; // invocation de la méthode Object.wait
9:   aload_1            // empiler la référence stockée (lock)
10:  monitorexit        // la fin du bloc synchronisé
```

2.4 Synchronisation dans la machine virtuelle Java

La machine virtuelle Java utilise des bibliothèques pour offrir un environnement multi-fils d'exécution aux programmes Java. Parmi les bibliothèques utilisées on retrouve les bibliothèques conformes aux POSIX [BD97] (*SableVM* utilise la bibliothèque conforme POSIX *Pthread*).

La plupart des systèmes d'exploitation sont soit complètement conformes aux POSIX (comme AIX, BSD/OS, HP-UX, LynxOS, Mac OS X, Solaris, OpenSolaris, UnixWare, FreeBSD et la plupart des distributions de Linux, NetBSD, OpenBSD), soit partiellement (comme Symbian OS et Noyau Windows NT quand Microsoft Services pour Unix 3.5 est activé).

POSIX (*Portable Operating System Interface*) est un standard produit par IEEE ; son but est d'assurer la portabilité des codes sources des applications d'un système d'exploitation vers un autre. Assurer le fonctionnement de l'application en la déplaçant d'un système vers un autre par une simple compilation du code source de l'application. Alors, une application qui est conforme aux POSIX peut être facilement portable d'un système d'exploitation conforme aux POSIX vers un autre, conforme lui aussi.

Les systèmes d'exploitation conformes aux POSIX offrent des bibliothèques (API) ; ainsi, une application qui utilise ces bibliothèques peut être facilement portable sur un autre système d'exploitation conforme aux POSIX qui offre des bibliothèques similaires.

Parmi les bibliothèques POSIX qui nous intéressent dans ce chapitre, nous retrouvons la bibliothèque de traitement des fils d'exécution connue sous le nom *Pthread*. Cette

bibliothèque offre des fonctions pour créer un fil d'exécution `pthread_create()` et terminer l'exécution d'un fil d'exécution `pthread_exit()`, et d'autres fonctions pour la jointure et le détachement des fils d'exécution.

Elle offre aussi des fonctions pour la synchronisation des fils d'exécution en utilisant des variables *mutex* de type `pthread_mutex_t` (exclusion mutuelle, voir section 1.4) pour créer un *mutex* `pthread_mutex_init()`, détruire un *mutex* `pthread_mutex_destroy()`, verrouiller un *mutex* `pthread_mutex_lock()` (le *opcode* `moniterenter` appelle cette fonction) avant d'aller dans une section critique, et déverrouiller un *mutex* `pthread_mutex_unlock()` (le *opcode* `moniterexit` appelle cette fonction) après la fin de la section critique.

Cette bibliothèque offre aussi des fonctions de communication entre les fils d'exécution (attente et notification, voir section 1.4) en utilisant les variables de conditions ; une variable de condition de type `pthread_cond_t` permet de faire dormir un fil d'exécution sur une condition quelconque jusqu'à ce qu'un autre fil d'exécution le réveille, créer une variable de condition `pthread_cond_init()`, la détruire `pthread_cond_destroy()`, faire dormir le fil d'exécution courant `pthread_cond_wait()` (invocation de la méthode `wait` de classe `java.lang.Object`) et faire notifier un fil d'exécution `pthread_cond_signal()` (invocation de la méthode `notify` de classe `java.lang.Object`) ou plusieurs fils d'exécution `pthread_cond_broadcast()` (invocation de la méthode `notifyAll` de classe `java.lang.Object`) qui étaient endormis sur une variable de condition. Une variable de condition est toujours en conjonction avec un verrou *mutex*.

2.5 Conclusion

Dans ce chapitre, nous avons vu les deux approches de synchronisation : l'exclusion mutuelle et l'attente et la notification. Nous avons vu la synchronisation en code-octet pour l'exclusion mutuelle par rapport à un bloc synchronisé et aussi une méthode synchronisée et, à la fin, nous avons vu le code-octet généré pour l'attente et la notification. Ensuite nous avons vu ce qu'est POSIX et les principales fonctions de la bibliothèque

Pthread, qui gère les fils d'exécution, leur synchronisation et la communication entre eux dans les systèmes d'exploitation qui sont conformes aux POSIX.

CHAPITRE III

LES INTERBLOCAGES ET LEURS DÉTECTIONS EN JAVA

3.1 Introduction

Parmi les problèmes rencontrés dans la vie des programmeurs et des utilisateurs, nous retrouvons celui où une application bloque à un moment donné. Il y a plusieurs causes à cela et parmi ces causes l'interblocage des fils d'exécution. Le programmeur essaie à droite et à gauche de retracer le problème, ce qui n'est pas une solution pratique surtout pour ce type de problème, parce qu'il n'y a pas une erreur qui se génère et que l'application parfois se bloque et parfois non.

L'interblocage peut être défini de la manière suivante : Un ensemble de fils d'exécution est en interblocage si chaque fil d'exécution est bloqué sur un objet détenu par un autre fil d'exécution de l'ensemble qu'il est le seul à pouvoir libérer [AT01].

La plupart des livres d'apprentissage de Java traitent des règles à suivre pour éviter les interblocages dans les programmes multifils d'exécution, par exemple de respecter l'ordre de verrouillage des objets et de réduire, le plus possible, le temps de verrouillage de ceux-ci.

Des recherches ont été faites pour résoudre ce problème, dont nous allons parler dans ce chapitre. Dans la section 3.2, nous parlerons des types d'interblocage et des conditions nécessaires pour qu'un interblocage ait lieu. Ensuite, dans la section 3.3, nous décrirons les différentes méthodes utilisées pour détecter et résoudre l'interblocage.

3.2 Types d'interblocage

Avant de commencer à parler des types d'interblocage, voyons les conditions nécessaires à un interblocage.

3.2.1 Conditions nécessaires d'un interblocage

Coffman et *al.* [CES71] ont montré qu'il faut quatre conditions pour provoquer un interblocage entre les processus, et elles sont les mêmes pour les fils d'exécution :

- L'exclusion mutuelle : le fil d'exécution a le contrôle total sur l'objet qu'il a verrouillé (Voir section 1.4.1).
- La détention et l'attente : un fil d'exécution qui verrouille un objet peut demander à verrouiller un autre objet.
- Pas de réquisition : un objet verrouillé doit être explicitement libéré par le fil d'exécution qui l'a verrouillé.
- Attente circulaire : il doit y avoir un cycle d'au moins deux fils d'exécution, chacun attendant un objet qui est verrouillé par un autre fil d'exécution du cycle.

Les trois premières conditions sont nécessaires pour provoquer un interblocage, de sorte que chaque fil d'exécution empêche tous les autres fils d'exécution d'utiliser le même objet qui le détiennent en même temps que lui, ce qui les laisse en attente (première condition), et aucun autre fil d'exécution ne peut retirer l'objet du fil d'exécution qui détient cette ressource (troisième condition) ; ce dernier demande d'autres ressources (objets) verrouillées par les autres, ce qui le met en attente aussi (deuxième condition). La quatrième condition s'applique pour certains types d'interblocage (Voir section 3.2.3).

3.2.2 Graphe ressource-allocation

Un graphe ressource-allocation représente la relation entre les fils d'exécution et les ressources verrouillées et disponibles. Comme le montre la figure 3.1, les fils d'exécution

(noeuds) sont représentés par des carrés, et les ressources (noeuds) par des cercles. Un arc orienté de la ressource vers le fil d'exécution représente l'allocation de la ressource au fil d'exécution, et un arc orienté du fil d'exécution vers la ressource représente l'attente de la disponibilité de la ressource par le fil d'exécution. les fils d'exécution sont représentés

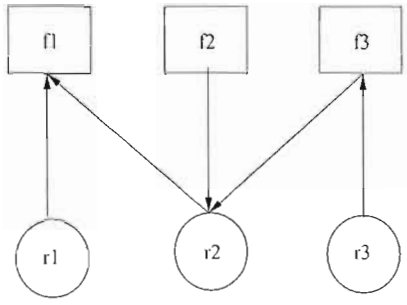


Figure 3.1 Graphe ressource-allocation

par f_1, f_2 et f_3 et les ressources par r_1, r_2 et r_3. Le fil d'exécution f_1 détient les ressources r_1 et r_2, alors que le fil d'exécution f_2 demande la ressource r_2 et que le fil d'exécution f_3 détient la ressource r_3 et demande la ressource r_2.

3.2.3 Types d'interblocage

Dans les sous-sections précédentes, nous avons parlé des conditions nécessaires pour causer un interblocage (Voir 3.2.1) et du graphe allocation-ressource qui schématise la relation d'allocation et d'attente entre les fils d'exécution et les ressources (Voir 3.2.2). Dans cette sous-section, nous allons voir les types d'interblocage connus et quelles sont les conditions nécessaires pour provoquer l'interblocage.

Nous allons voir les types d'interblocage suivants :

- Attente cyclique.
- Attente imbriquée.

3.2.3.1 Attente cyclique

Parmi les types d'interblocage on trouve l'attente cyclique, comme nous l'avons vu dans la sous-section 3.2.1 (quatrième condition). Chaque fil d'exécution dans le cycle détient une ressource (objet) et attend une autre ressource détenue par un autre fil d'exécution qui appartient au même cycle ; aucun fil d'exécution ne peut retirer la ressource qui a besoin de leur détenteur (deuxième condition). Cette attente cyclique crée un interblocage. Comme le montre la figure 3.2, le fil d'exécution f_1 détient la

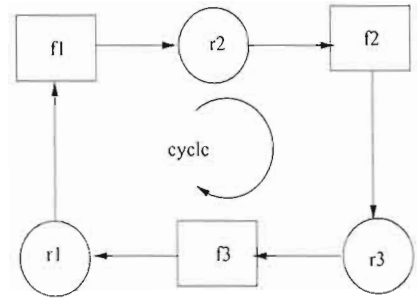


Figure 3.2 Attente cyclique

ressource r_1 et attend la ressource r_2, alors que le fil d'exécution f_2 détient la ressource r_2 et attend que la ressource r_3 soit disponible ; et il en va ainsi pour le fil d'exécution f_3 qui détient la ressource r_3 et attend la ressource r_1. Prenons un exemple en Java pour voir ce type d'interblocage de près.

```
class Resource implements Runnable {
    Resource superviseur_;

    public void setSuperviseur(Resource superviseur {
        superviseur_ = superviseur;
    }

    public Resource getSuperviseur() {
        return superviseur_;
    }

    public synchronized void assignTache(String tache) {
        System.out.println(tache);
    }
```

```
    public synchronized void run() {
        getSuperviseur().assignTache(" -> tâche ");
    }
}

public class InterblocageCyclique {

    public static void main(String argv[]) {
        Resource employe1 = new Resource();
        Resource employe2 = new Resource();
        employe1.setSuperviseur(employe2);
        employe2.setSuperviseur(employe1);
        Thread employe1Thread = new Thread(employe1);
        Thread employe2Thread = new Thread(employe2);
        employe1Thread.start(); employe2Thread.start();
    }
}
```

Nous remarquerons dans cet exemple que les deux fils d'exécution `employe1Thread` et `employe2Thread` appellent la méthode `getSuperviseur()` qui retourne l'objet `superviseur_` de chaque objet (`employe2` pour `employe1`, `employe1` pour `employe2`) ; ensuite, chaque fil d'exécution appelle la méthode `assignTache()` de l'autre fil d'exécution, qui est une méthode synchronisée, et la méthode `run()` est aussi synchronisée. Au moment où chaque fil d'exécution exécute sa méthode `run()`, les deux appellent en même temps la méthode `assignTache()` de l'autre fil d'exécution ; les deux vont être bloqués, parce qu'ils seront en attente que l'objet se libère, alors que les deux objets sont en exclusion mutuelle à cause de la méthode `run()`.

3.2.3.2 Attente imbriquée

Ce type d'interblocage advient dans le cas d'utilisation de l'attente et de la notification (Voir 1.4.2), quand un ou plusieurs fils d'exécution utilisent un objet intermédiaire pour utiliser un autre objet et que ce dernier dort en attendant une notification d'un fil d'exécution qui se trouve dans les fils d'exécution qui utilisent l'objet intermédiaire.

```
class Manager {
    boolean receiveNotification = false;
```

```
    public synchronized void attente() {
        while (!receiveNotification) {
            try {
                wait();
            } catch(InterruptedException e) { }
        }
    }

    public synchronized void notification() {
        receiveNotification = true;
        notifyAll();
    }
}

class Intermediaire {
    Manager manager_;

    public void setManager(Manager manager) {
        manager_ = manager;
    }

    public Manager getManager() {
        return manager_;
    }

    public synchronized void attenteManager() {
        getManager().attente();
    }

    public synchronized void notificationManager() {
        getManager().notification();
    }
}

class Employe implements Runnable {
    Intermediaire intermediaire_;
    boolean premier_ = true;

    public void setIntermediaire(Intermediaire intermediaire) {
        intermediaire_ = intermediaire;
    }

    public Intermediaire getIntermediaire() {
        return intermediaire_;
    }

    public Employe(boolean premier) {
        premier_ = premier;
    }
```

```
    public synchronized void run() {
        if (premier_) {
            getIntermediaire().attenteManager();
        }

        getIntermediaire().notificationManager();
    }
}

public class InterblocageImbrique {

    public static void main(String argv[]) {
        Manager manager = new Manager();
        Intermediaire intermediaire = new Intermediaire();
        Employe employe1 = new Employe(true);
        Employe employe2 = new Employe(false);
        intermediaire.setManager(manager);
        employe1.setIntermediaire(intermediaire);
        employe2.setIntermediaire(intermediaire);
        Thread employe1Thread = new Thread(employe1);
        Thread employe2Thread = new Thread(employe2);
        employe1Thread.start();
        employe2Thread.start();
    }
}
```

Dans l'exemple précédent, les deux fils d'exécution `employe1` et `employe2` vont utiliser l'objet `intermediaire_` pour appeler une méthode de l'objet `manager`. Le fil d'exécution `employe1` va appeler la méthode `attenteManager()`, qui appelle la méthode `attente()` du `manager`, suivie de la méthode `notification()` du `manager`, alors que le fil d'exécution `employe2` va appeler la méthode `notificationManager()` de l'objet intermédiaire, parce que le booléen `premier_` est à faux, et cette méthode va appeler la méthode `notification()` de l'objet `manager`. Et puisque les deux fils d'exécution utilisent le même objet `intermediaire`, si le fil d'exécution `employe1` s'exécute en premier, alors il va s'endormir en libérant l'objet `manager`, mais l'objet `intermediaire` restera toujours verrouillé, ce qui mettra tous les fils d'exécution qui utilisent l'objet `intermediaire` en attente, jusqu'à ce que le fil d'exécution `employe1` libère l'objet ; ce dernier, tant qu'il ne reçoit pas une notification, reste verrouillé, et tous les fils d'exécution qui génèrent la notification (`employe2`) sont bloqués sur lui.

3.3 Détection des interblocages

Dans cette partie du chapitre, nous allons voir les différentes méthodes de détection d'interblocage utilisées [FJ06], dont les types de détections suivants :

- Détection statique.
- Détection dynamique.
- Vérification du modèle (Model checking).
- Analyse de l'exécution.

3.3.1 Détection statique

La détection statique se base sur la détection des interblocages hors exécution du programme, ce qui signifie que la détection se base sur l'analyse du code source ou du code-octet d'un programme Java.

La détection statique se base sur la création d'une représentation abstraite du programme. Parmi les représentations on retrouve un graphe, où les sommets sont des points stratégiques dans le programme (par exemple l'appel d'une méthode, un saut, l'acquisition et la libération des verrous) et les arcs représentent les transitions d'un point vers un autre. Un graphe est construit pour chaque fil d'exécution.

Une des méthodes connues est de sauvegarder à chaque point stratégique l'ensemble des verrous libres avant et après la transition pour chaque fil d'exécution qui passe par cette transition.

La détection statique de l'interblocage commence par le sommet source de chaque graphe (où chaque fil d'exécution commence son exécution) et recherche les possibilités où deux ou plusieurs fils d'exécution peuvent causer un interblocage par création d'un cycle d'attente dans le graphe.

La détection statique consomme beaucoup de mémoire, parce que pour chaque possibilité d'exécution, un nouveau sous-graphe est construit. Le problème est aussi que

la détection statique n'est pas exacte, car elle produit beaucoup de possibilités qui ne représentent pas des interblocages.

RacerX [EA03] est un outil de détection statique des interblocages et de course aux données (*data race*) pour les programmes écrits en C. La méthode de détection est une méthode du haut vers le bas, sensible au contexte et interprocédurale, qui analyse l'ensemble des verrous. L'analyse se fait par une recherche du plus profond sommet jusqu'à le sommet source (*depth-first search* (*DFS*)) dans le graphe de contrôle des flux (*control flow graph* (*CFG*)). La recherche commence par le sommet source de chaque graphe (où le fil d'exécution commence son exécution) et construit l'ensemble des verrous à chaque sommet visité dans le *CFG*. L'outil *RacerX* détecte les interblocages en analysant le contenu de chaque ensemble des verrous pour chaque sommet dans le graphe construit.

3.3.2 Détection dynamique

La détection dynamique se base sur la détection des interblocages durant l'exécution du programme, ce qui signifie que la détection se base sur le fait de trouver l'interblocage au fur et à mesure que les fils d'exécution sont en train de s'exécuter.

La plupart des méthodes de détection dynamique utilisent le graphe ressource-allocation (Voir 3.2.2) pour détecter les interblocages, en cherchant s'il existe des cycles dans le graphe selon la quatrième condition nécessaire (Voir 3.2.1) pour provoquer un interblocage.

L'algorithme *Dreadlocks* (Voir 6.2) se base sur la détection dynamique des interblocages.

3.3.3 Vérification du modèle

La détection par vérification du modèle (*model checking*) se base sur la construction d'une représentation abstraite du programme ; cette représentation est appelée un modèle. Ce modèle vérifie des propriétés spécifiques.

Généralement, un modèle est représenté par un graphe orienté, où les noeuds représentent

des points stratégiques et les arcs représentent des transitions (Voir 3.3.1). Généralement, la détection par vérification du modèle utilise l'analyse statique pour construire un modèle du programme.

La détection de l'interblocage est de vérifier si le programme viole ce modèle.

Parmi les outils qui utilisent la vérification du modèle (*model checking*), on retrouve Java *PathFinder*.

3.3.4 Analyse de l'exécution

La détection par l'analyse de l'exécution consiste à surveiller une ou plusieurs exécutions d'un programme et à générer beaucoup d'informations. Ces informations générées permettent de trouver un ensemble de propriétés utiles selon plusieurs critères, comme par exemple l'ensemble des verrous. Ces informations sont utilisées pour prédire si un interblocage peut advenir dans une autre exécution du programme. La prédiction est faite en vérifiant si une ou plusieurs propriétés sont violées.

Le problème de ce type de détection est qu'il surveille seulement un ensemble limité d'exécutions, alors que le programme peut avoir des centaines de cas d'exécutions différentes.

Parmi les outils qui utilisent l'analyse de l'exécution, on retrouve *Eraser*. [HK00]

3.4 Conclusion

Dans ce chapitre, nous avons vu les conditions nécessaires pour provoquer un interblocage ; ensuite, nous avons parlé du graphe ressource-allocation qui est utilisé dans plusieurs méthodes de détection d'interblocage. Nous avons aussi parlé des types d'interblocages qui provoquent un interblocage dans un programme. À la fin, nous avons vu les méthodes de détection d'interblocage.

CHAPITRE IV

DÉTECTION IMMÉDIATE DES INTERBLOCAGES

Dans ce chapitre, nous parlerons de notre méthode de détection immédiate d'interblocage ; il s'agit d'une méthode de détection dynamique (voir 3.3.2). Notre méthode détecte seulement les interblocages causés par l'utilisation de l'exclusion mutuelle ; elle ne se préoccupe pas des interblocages causés par l'attente et la notification.

Ce chapitre est structuré comme suit : dans la section 4.1, nous présenterons une vue d'ensemble de notre algorithme de détection immédiate des interblocages. Dans la section 4.2, il s'agira de la construction d'une forêt d'attente pendant l'exécution des programmes Java. Dans la section 4.3, nous présenterons une vue d'ensemble de notre méthode de détection immédiate des interblocages. Dans la section 4.4, nous présenterons une version simplifiée de notre algorithme. Dans la section 4.5, nous exposerons les problèmes liés aux modifications concurrentes. Dans la section 4.6, nous présenterons la version complète de l'algorithme pour résoudre les problèmes de la recherche infinie, la fausse détection des interblocages et la non détection des interblocages dus aux modifications concurrentes dans les arbres d'attente pendant la recherche de la racine de l'arbre. Dans la section 4.7, nous exposerons le brisement de l'interblocage. Dans la section 4.8, nous présenterons le code de la version finale de notre algorithme. Finalement, dans la section 4.9, nous présenterons notre conclusion.

4.1 Vue d'ensemble

Voici, en bref, comment fonctionne notre algorithme de détection immédiate des interblocages. Pendant l'acquisition, le relâchement et l'attente des verrous par les fils d'exécution, un ou plusieurs arbres d'attente sont construits (forêt d'attente). Notre algorithme de détection immédiate d'interblocage se base sur la recherche de la racine de l'arbre d'attente courant. Si un fil d'exécution détecte qu'il va créer un interblocage en se mettant en attente d'un verrou, alors une exception est soulevée pour briser l'interblocage.

4.2 Construction de la forêt d'attente

Nous présenterons, dans cette section, la construction de la forêt d'attente dans la machine virtuelle. Dans la sous-section 4.2.1, nous présenterons la structure d'un arbre d'attente et, dans la sous-section 4.2.2, la structure d'une forêt d'attente. Ensuite, nous présenterons, dans les sous-sections 4.2.3, 4.2.4 et 4.2.5, la construction de la forêt d'attente. Dans la sous-section 4.2.6, nous exposerons la représentation d'un fil d'exécution actif dans un arbre d'attente. Dans la sous-section 4.2.7, nous présenterons un exemple de construction d'une forêt d'attente.

4.2.1 Arbre d'attente

L'arbre d'attente est une structure de données de type arbre. La figure 4.1 illustre cette structure. Les nœuds de l'arbre sont représentés par des carrés et des cercles. Les carrés représentent des fils d'exécution et les cercles représentent des verrous. Les arcs dirigés qui relient les nœuds représentent soit l'acquisition d'un verrou v par un fil d'exécution f (un arc relie le verrou v au fil d'exécution f), soit un fil d'exécution f qui se met en attente d'un verrou v (un arc relie le fil d'exécution f au verrou v). Un fil d'exécution qui n'a pas de parent est un fil d'exécution actif, et un fil d'exécution qui a un parent est un fil d'exécution en attente d'un verrou.

Notons qu'un verrou ne peut être acquis par plus d'un seul fil d'exécution à la fois et qu'un fil d'exécution ne peut être en attente de plus d'un seul verrou à la fois. Toutefois, un fil d'exécution peut posséder plusieurs verrous.

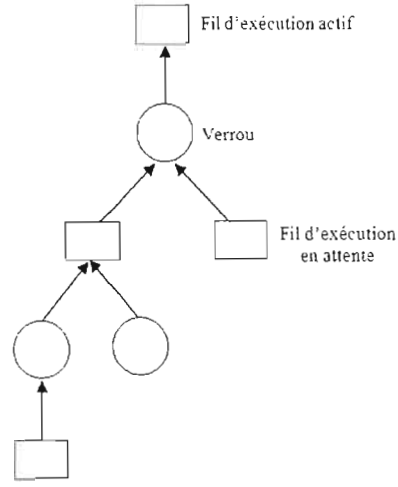


Figure 4.1 Arbre d'attente

4.2.2 Forêt d'attente

Dans la machine virtuelle, à un moment donné, nous trouvons un ou plusieurs arbres d'attente. La structure de données est donc une forêt. Le nombre d'arbres d'attente dont le sommet est un fil d'exécution correspond au nombre de fils d'exécution actifs. (voir figure 4.2)

Nous supposons l'absence d'interblocages. Un interblocage introduirait un cycle dans le graphe d'attente et la structure ne serait plus une forêt. Notre hypothèse se justifie par la détection immédiate des interblocages, qui nous permet d'éviter l'ajout de cycles dans la structure (voir section 4.3).

Notons que le version finale de l'algorithme pourra temporairement introduire des cycles pendant la détection des interblocages (voir section 4.6.3).

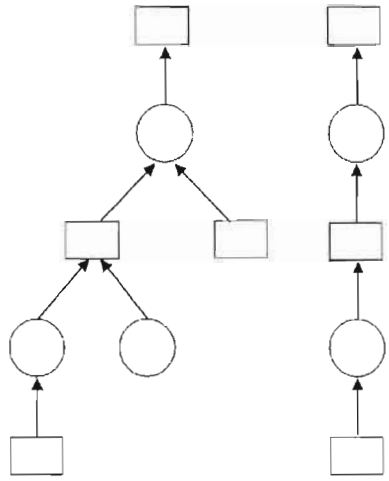


Figure 4.2 Forêt d'attente

4.2.3 Acquisition d'un verrou

Lorsqu'un fil d'exécution actif f acquiert un verrou libre v (le verrou n'a pas de fil d'exécution propriétaire), un arc partant du verrou v vers le fil d'exécution f ($v \rightarrow f$) est ajouté dans la forêt d'attente. Le fil d'exécution f devient le propriétaire du verrou v.

La procédure ACQUISITION-VERROU (f, v) permet de mettre à jour la forêt d'attente après l'acquisition du verrou v par le fil d'exécution f. Cette procédure (algorithme 1) prend en entrée le fil d'exécution actif f et le verrou libre v. Le champ *propriétaire* du verrou v indique le fil d'exécution propriétaire du verrou v. Il y a une condition préalable à l'exécution de la procédure ACQUISITION-VERROU : c'est que le propriétaire du verrou v soit *NIL* (ligne 2). La procédure affecte le fil d'exécution f au champ *propriétaire* du verrou v (ligne 3) pour indiquer que le fil d'exécution f a acquis le verrou v (le fil d'exécution f est devenu le propriétaire du verrou v).

La figure 4.3 représente l'acquisition d'un verrou v par un fil d'exécution actif f.

La procédure ACQUISITION-VERROU(f, v) s'exécute en $O(1)$.

```
1 ACQUISITION-VERROU(f, v)
2 assert v.propriétaire = NIL
3 v.propriétaire←f
```

Algorithme 1 : Acquisition d'un verrou

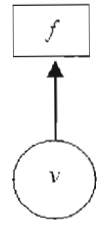


Figure 4.3 Acquisition d'un verrou

4.2.4 Attente d'un verrou

Lorsqu'un fil d'exécution actif f se met en attente d'un verrou v qui est déjà acquis par un autre fil d'exécution (le verrou v a déjà un fil d'exécution propriétaire autre que f), un arc partant du fil d'exécution f vers le verrou v ($f \rightarrow v$) est ajouté dans la forêt d'attente.

La procédure ATTENTE-VERROU (f, v) permet de mettre à jour la forêt d'attente avant que le fil d'exécution f se mette en attente du verrou v. Cette procédure (algorithme 2) prend en entrée le fil d'exécution actif f et le verrou v déjà acquis. Le champ *verrou_attente* du fil d'exécution f indique le verrou duquel f est en attente. Il y a une condition préalable à l'exécution de la procédure ATTENTE-VERROU : c'est que le propriétaire du verrou v soit différent du fil d'exécution f et ne soit pas *NIL* (ligne 2). La procédure affecte le verrou v au champ *verrou_attente* du fil d'exécution f (ligne 3) pour indiquer que f est en attente du verrou v.

```
1 ATTENTE-VERROU(f, v)
2 assert v.propriétaire ≠ NIL et v.propriétaire ≠f
3 f.verrou_attente←v
```

Algorithme 2 : Attente d'un verrou

La figure 4.4 représente l'attente d'un fil d'exécution f sur un verrou v.

Figure 4.4 Attente d'un verrou

La procédure ATTENTE-VERROU(f, v) s'exécute en *O(1)*.

4.2.5 Relâchement d'un verrou

Lorsqu'un fil d'exécution actif f relâche un verrou v (le fil d'exécution f est le propriétaire du verrou v), l'arc partant du verrou v vers le fil d'exécution f ($v \rightarrow f$) est supprimé de la forêt d'attente et le verrou v devient libre.

La procédure RELACHER-VERROU(f, v) permet de mettre à jour la forêt d'attente après que le fil d'exécution f ait relâché le verrou v. Cette procédure (algorithme 3) prend en entrée le fil d'exécution actif f et le verrou v. Il y a une condition préalable à l'exécution de la procédure RELACHER-VERROU : c'est que le fil d'exécution f soit le propriétaire du verrou v (ligne 2). La procédure affecte la valeur *NIL* au champ *propriétaire* du verrou v (ligne 3) pour indiquer que v n'a pas de propriétaire.

```
1 RELACHER-VERROU(f, v)
2 assert v.propriétaire = f
3 v.propriétaire←NIL
```

Algorithme 3 : Relâchement d'un verrou

La figure 4.5 représente le relâchement d'un verrou v par le fil d'exécution f.

La procédure RELACHER-VERROU(f, v) s'exécute en *O(1)*.

Remarque : Les opérations d'acquisition, attente et relâchement d'un verrou sont

Figure 4.5 Relâchement d'un verrou

exécutées en exclusion mutuelle de façon temporaire sur le verrou.

4.2.6 Fil d'exécution actif dans l'arbre d'attente

Un arbre d'attente (voir figure 4.1) contient un seul fil d'exécution actif, qui est le fil d'exécution au sommet de l'arbre d'attente. Tous les autres fils d'exécution du même arbre sont déjà en attente des verrous. Le fil d'exécution actif est le seul fil d'exécution de l'arbre qui peut acquérir d'autres verrous, se mettre en attente d'un verrou ou relâcher des verrous (acquis par lui).

4.2.7 Exemple de construction d'une forêt d'attente

Soient les fils d'exécution T_1, T_2, T_3 et T_4, et soient les objets A, B, C, D et E.

Le fil d'exécution T_1 exécutera le code suivant :

```
synchronized(A) {
    synchronized(B) {...}
}
```

Le fil d'exécution T_2 exécutera le code suivant :

```
synchronized(C) {
    synchronized(B) {...}
}
```

Le fil d'exécution T_3 exécutera le code suivant :

```
synchronized(D) {
    synchronized(E) {...}
}
```

Le fil d'exécution T_4 exécutera le code suivant :

```
synchronized(E) {
    synchronized(D) {...}
}
```

Le fil d'exécution T_1 exécute son code et acquiert les deux verrous A et B. En exécutant son code du bloc synchronisé, un temps d'exécution processeur est affecté à T_2 qui exécute son code et acquiert le verrou C puis se met en attente du verrou A. Un temps d'exécution processeur est affecté à T_3, après la mise en attente de T_2, qui exécute son code et acquiert les deux verrous D et E. En exécutant son code du bloc sychronisé, un temps d'exécution processeur est affecté à T_4 qui exécute son code et se met en attente du verrou E. Les arbres d'attentes construits après la mise en attente de T_4 sont schématisés dans la figure 4.6

4.3 Vue d'ensemble de l'algorithme de détection des interblocages

Lorsqu'un fil d'exécution actif f essaie d'acquérir un verrou v déjà acquis par un autre fil d'exécution, deux possibilités existent :

- le verrou v se trouve dans le même arbre d'attente que le fil d'exécution f, ou
- le verrou v se trouve dans un autre arbre d'attente.

Notre algorithme de détection immédiate des interblocages vérifie que le fil d'exécution *fil-sommet*, au sommet de l'arbre d'attente du verrou v, est le fil d'exécution actif f, et que le chemin du verrou v vers *fil-sommet* n'a pas changé pendant la recherche du sommet. Si *fil-sommet* est le même que le fil d'exécution actif f, alors un interblocage est détecté, et une exception est soulevée pour briser l'interblocage ; sinon aucun interblocage n'est détecté.

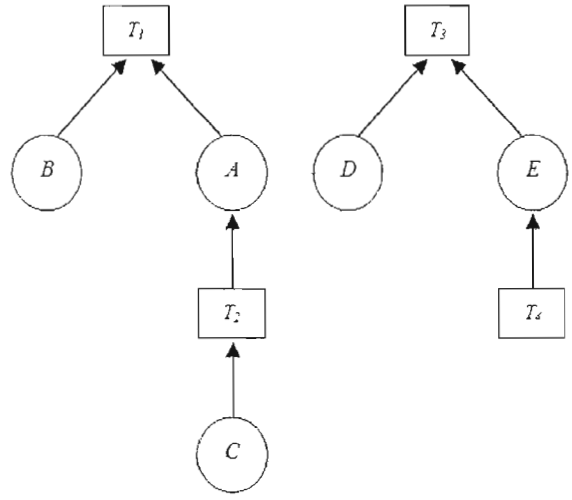


Figure 4.6 Coexistence de plusieurs arbres d'attentes

4.4 Vision simplifiée de l'algorithme de détection immédiate des interblocages

Dans cette version simplifiée de l'algorithme de détection immédiate des interblocages, nous supposons que la forêt d'attente ne change pas pendant le déroulement de l'algorithme ; cela signifie qu'aucun fil d'exécution ne peut faire des opérations concurrentes sur les verrous (acquisition, attente ou relâchement) et qu'aucun nouveau fil d'exécution ne peut être créé.

Cette version de l'algorithme consiste à trouver le sommet *fil-sommet* à partir du verrou v, en demande d'acquisition, par la recherche de la racine de l'arbre courant, et de le comparer avec le fil d'exécution actif f :

- si *fil-sommet* est f, alors un interblocage est détecté. f a demandé un verrou qui se trouve dans son sous-arbre. Si f se met en attente du verrou v (il créera un cycle dans son arbre), tous les fils d'exécution de son sous-arbre vont être en interblocage, parce qu'il est le seul fil d'exécution actif, sinon :
- le sommet *fil-sommet* est différent de f, alors il n'y aura pas d'interblocage si f se

met en attente du verrou v. f a demandé un verrou qui se trouve dans un autre arbre d'attente.

La procédure DETECTER-INTERBLOCAGE(f, v) (algorithme 4) prend en entrée le fil d'exécution actif f et le verrou v. La procédure retourne la valeur «vrai» si un interblocage est détecté, sinon elle retourne la valeur «faux». La procédure initialise la variable *verrou* par v, ensuite elle parcourt l'arbre du verrou v jusqu'au sommet (lignes 3 à 5). À chaque itération de la boucle tant que, la variable *verrou* reçoit la valeur *verrou.propriétaire.verrou_attente* (c'est-à-dire le verrou que le propriétaire du *verrou* est en attente). Les itérations s'arrêtent quand une des deux conditions est fausse :

- La condition *verrou.propriétaire* $\neq f$ est fausse. Cela signifie que le sommet de l'arbre est le fil d'exécution actif f. Dans ce cas, un interblocage est détecté et la procédure retourne la valeur «vrai» (ligne 6), ou :
- La condition *verrou* $\neq NIL$ est fausse. Cela signifie que nous avons atteint le sommet de l'arbre et qu'il est différent du fil d'exécution actif f. Dans ce cas, aucun interblocage n'est détecté, et la procédure retourne la valeur «faux» (ligne 6).

```
1 DETECTER-INTERBLOCAGE(f, v)
2 verrou←v
3 tant que verrou ≠ NIL et verrou.propriétaire ≠ f faire
4 |   verrou←verrou.propriétaire.verrou_attente
5 fin tant que
6 retourner verrou ≠ NIL et verrou.propriétaire = f
```

Algorithme 4 : Vision simplifiée de l'algorithme de détection immédiate des interblocages

4.4.1 Exemple d'un cas avec interblocage

Soit le fil d'exécution actif T_1, qui demande le verrou v_3 qui se trouve dans son sous-arbre. Le fil d'exécution actif T_1 appelle la procédure DETECTER-INTERBLOCAGE avec les paramètres T_1 et v_3 respectivement (voir la figure 4.7).

Le fonctionnement de notre algorithme simplifié dans l'exemple de la figure 4.7 va comme

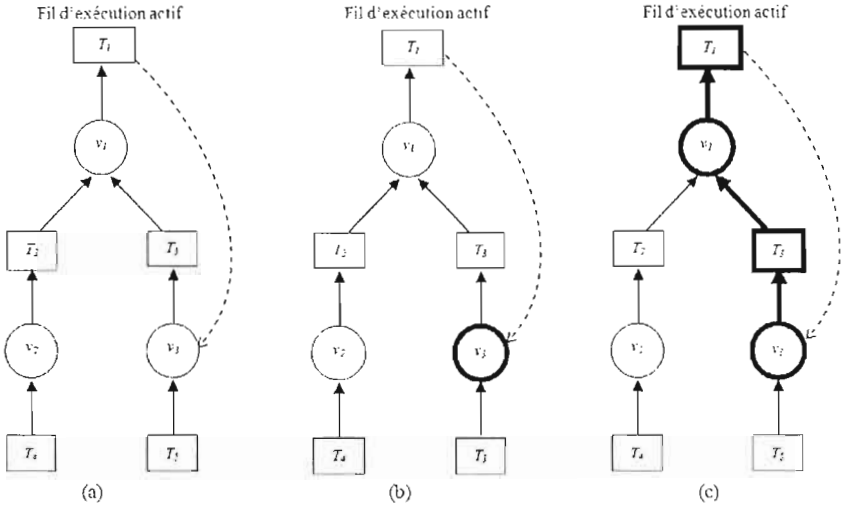


Figure 4.7 Détection d'un interblocage par l'algorithme simplifié.

suit :

a) T_1 demande le verrou v_3 qui est acquis par T_3.

b) La procédure initialise la variable *verrou* par v_3 (algorithme 4, ligne 2).

c) La procédure exécute les itérations suivantes de la boucle `tant que` :

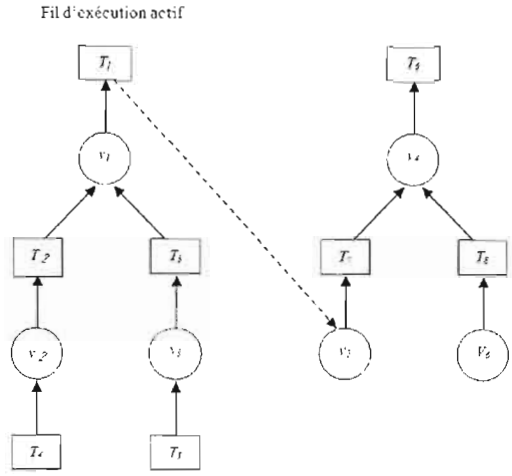
1- la condition «$v_3 \neq NIL$ **et** $v_3.propriétaire \neq f$» est vraie. *verrou* reçoit v_1 ($v_3.propriétaire.verrou\_attente$).

2- la condition «$v_1 \neq NIL$ **et** $v_1.propriétaire \neq f$» est fausse, parce que le propriétaire du verrou v_1 est T_1, c'est-à-dire que T_1 est le sommet de l'arbre.

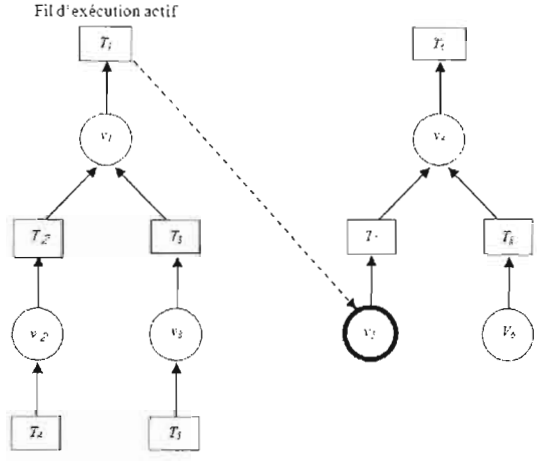
La procédure sort de la boucle `tant que`. La condition (ligne 6) «$v_1 \neq NIL$ **et** $v_1.propriétaire = f$» est vraie, et la procédure retourne la valeur «vrai».

4.4.2 Exemple d'un cas sans interblocage

Soit le fil d'exécution actif T_1, qui demande le verrou v_5 qui se trouve dans un autre arbre que lui. Le fil d'exécution actif T_1 appelle la procédure DETECTER-INTERBLOCAGE avec les paramètres T_1 et v_5 respectivement (voir la figure 4.8).



(a)



(b)

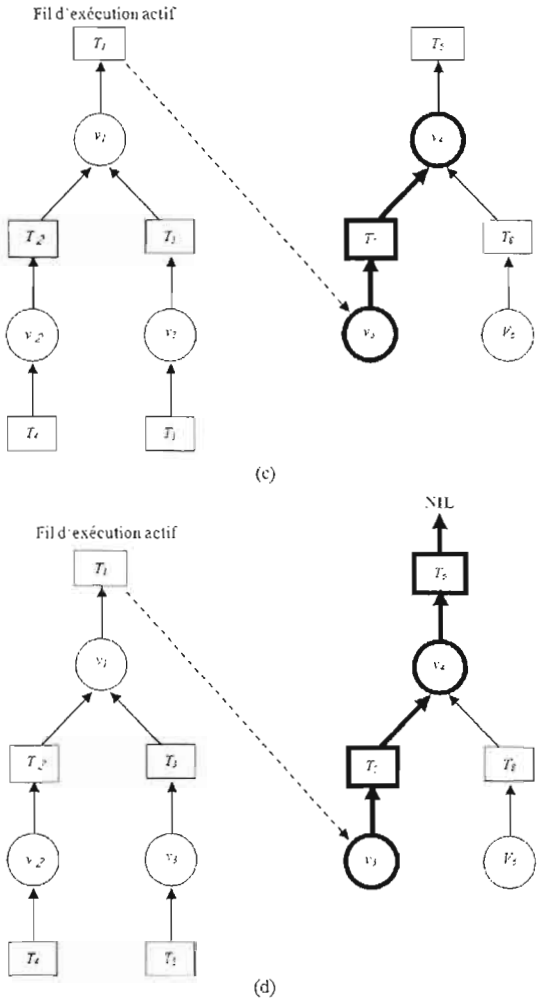


Figure 4.8 Détection de non existance d'un interblocage par l'algorithme simplifié.

Le fonctionnement de notre algorithme simplifié dans l'exemple de la figure 4.8 va comme suit :

a) T_1 demande le verrou v_5 qui est acquis par T_7.

b) La procédure initialise la variable *verrou* par v_5.

c) La condition «$v_5 \neq NIL$ **et** $v_5.propriétaire \neq f$» est vraie. *verrou* reçoit v_4 ($v_5.propriétaire.verrou\_attente$).

d) La condition «$v_4 \neq NIL$ **et** $v_4.propriétaire \neq f$» est vraie. *verrou* reçoit *NIL* ($v_4.propriétaire.verrou\_attente$). Cette fois la condition (ligne 3) «$verrou \neq NIL$» est fausse. La procédure sort de la boucle `tant que`. La condition (ligne 6) «$verrou \neq NIL$» est fausse et la procédure retourne la valeur «faux».

4.4.3 Analyse de l'algorithme

Dans cette version simplifiée de l'algorithme de détection immédiate des interblocages, il ne sera pas possible de faire une recherche de la racine dans un cycle, parce qu'à chaque fois qu'un fil d'exécution crée un cycle (interblocage), il est brisé immédiatement par notre algorithme. Pour cette raison, le cas le plus défavorable est que chaque fil d'exécution est visité une seule fois pendant la recherche de la racine.

Dans le cas le plus défavorable, où il existe n fils d'exécution dans la machine virtuelle, notre algorithme visite pendant la recherche de la racine :

- tous les n fils d'exécution, incluant le fil d'exécution actif dans le cas d'un interblocage.
- tous les *n-1* fils d'exécution, sauf le fil d'exécution actif dans le cas sans interblocage.

La boucle **tant que** des lignes 3-5 s'exécute en *O(n)*, et les lignes 2 et 6 s'exécutent en *O(1)*. Donc, dans le cas le plus défavorable, la procédure DETECTER-INTERBLOCAGE (algorithme 4) s'exécute en *O(n)*, où n est le nombre de fils d'exécution dans la machine virtuelle pendant l'exécution de la procédure.

4.4.4 Mise à jour de la procédure d'attente d'un verrou

Nous avons présenté la procédure ATTENTE-VERROU dans la section 4.2.4. Nous modifions la procédure (algorithme 5) pour briser l'interblocage si l'attente du verrou v par le fil d'exécution actif f crée un interblocage.

```
1 ATTENTE-VERROU(f, v)
2 assert v.propriétaire ≠ NIL et v.propriétaire ≠f
3 si DETECTER-INTERBLOCAGE(f, v) alors
4 |   soulever-exception
5 sinon
6 |   f.verrou_attente←v
7 fin si
```

Algorithme 5 : Attente d'un verrou modifiée - version simplifié de l'algorithme

La nouvelle procédure détecte si l'attente du v par f, en appelant la procédure DETECTER-INTERBLOCAGE (ligne 3), crée un interblocage. Si c'est le cas, alors une exception est soulevée (ligne 4) ; sinon, le fil d'exécution f entre en attente du verrou v (ligne 6).

L'appel de la procédure DETECTER-INTERBLOCAGE (ligne 3) s'exécute en $O(n)$ (voir la sous-section 4.4.3), où n est le nombre de fils d'exécution dans la machine virtuelle pendant l'exécution de la procédure, et le code de la ligne 4 ou de la ligne 6 s'exécute en $O(1)$. Donc, la procédure ATTENTE-VERROU s'exécute dans le cas le plus défavorable en $O(n)$.

4.5 Problématiques des modifications concurrentes

Nous avons supposé, dans la version simplifiée de l'algorithme, que la forêt d'attente ne change pas pendant la détection de l'interblocage. Mais, dans une machine virtuelle, les fils d'exécution libèrent, acquièrent ou se mettent en attente des verrous en concurrence. Ces opérations concurrentes causent des changements dans la forêt d'attente pendant la détection de l'interblocage. Dans certains cas, elles conduisent à :

- une recherche infinie ;
- la fausse détection des interblocages ;

- la non détection des interblocages.

Remarque : Empêcher les modifications concurrentes imposerait la sérialisation de toutes les opérations de synchronisation [1]. Ceci nécessiterait l'utilisation de synchronisation globale. L'impact d'une telle décision sur la performance du système pourrait être importante surtout dans un système multi-processeurs, car les autres processeurs qui désirent acquérir ou libérer un verrou doivent attendre après le processeur qui est en train de faire une détection d'interblocage. La conception du langage Java permet les opérations concurrentes d'acquisition et de libération des verrous ; nous avons donc décidé de les permettre dans notre algorithme.

4.5.1 Recherche infinie

Lorsque deux ou plusieurs arbres d'attente échangent des verrous ou des nouveaux arbres d'attente construits pendant que notre algorithme est en train de rechercher la racine, cette recherche continue infinement si l'algorithme court à la recherche de la racine d'un arbre à un autre sans qu'il finisse par trouver la racine de l'arbre.

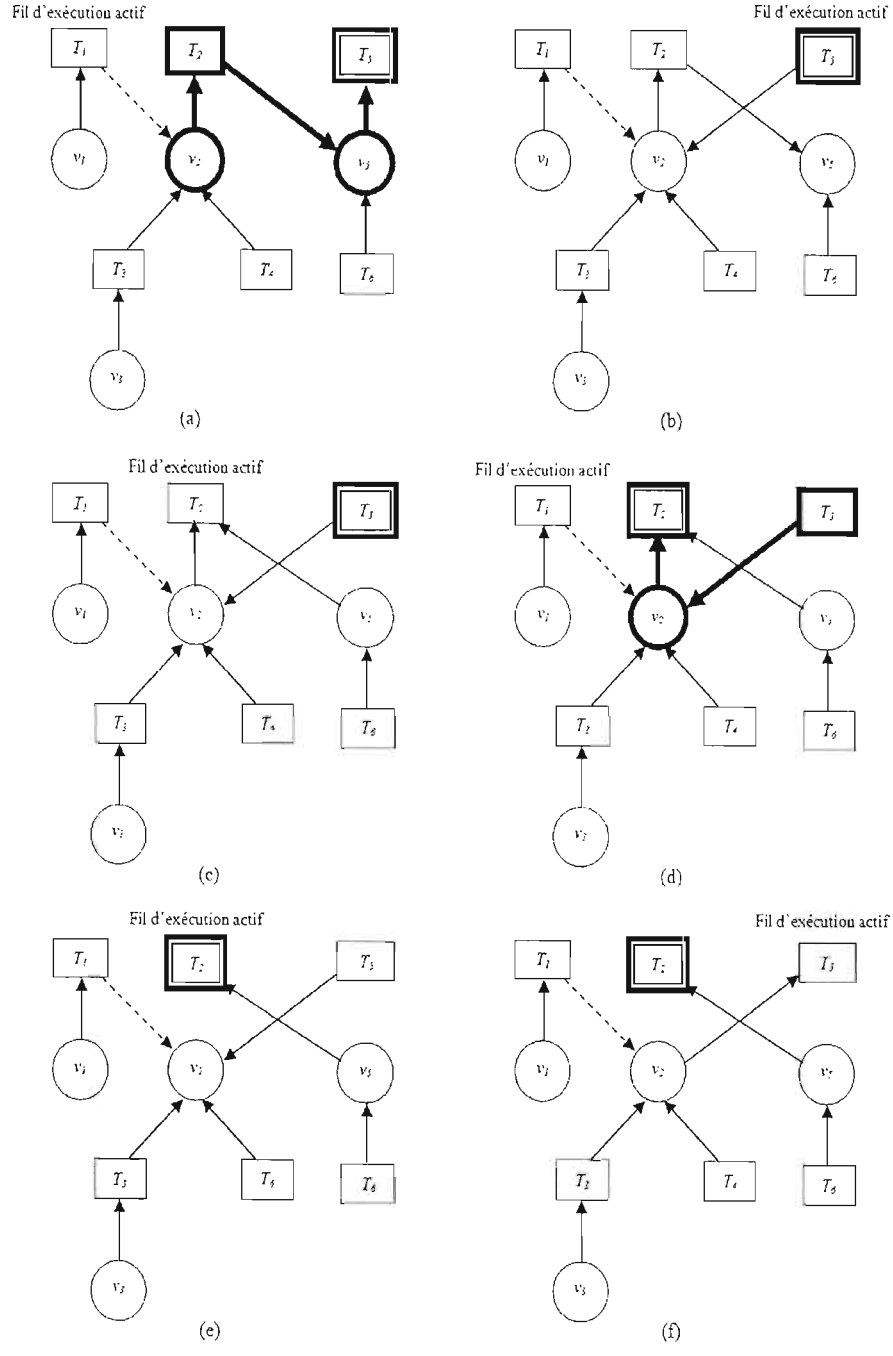
L'exemple de la figure 4.9 illustre un cas assez complexe d'une recherche infinie du sommet de l'arbre d'attente. Le fil d'exécution T_1 demande d'acquérir le verrou v_2, qui est acquis par le fil d'exécution T_2. T_1 exécute la procédure ATTENTE-VERROU(T_1, v_2) (algorithme 5), qui appelle la procédure DETECTER-INTERBLOCAGE(T_1, v_2) (algorithme 4).

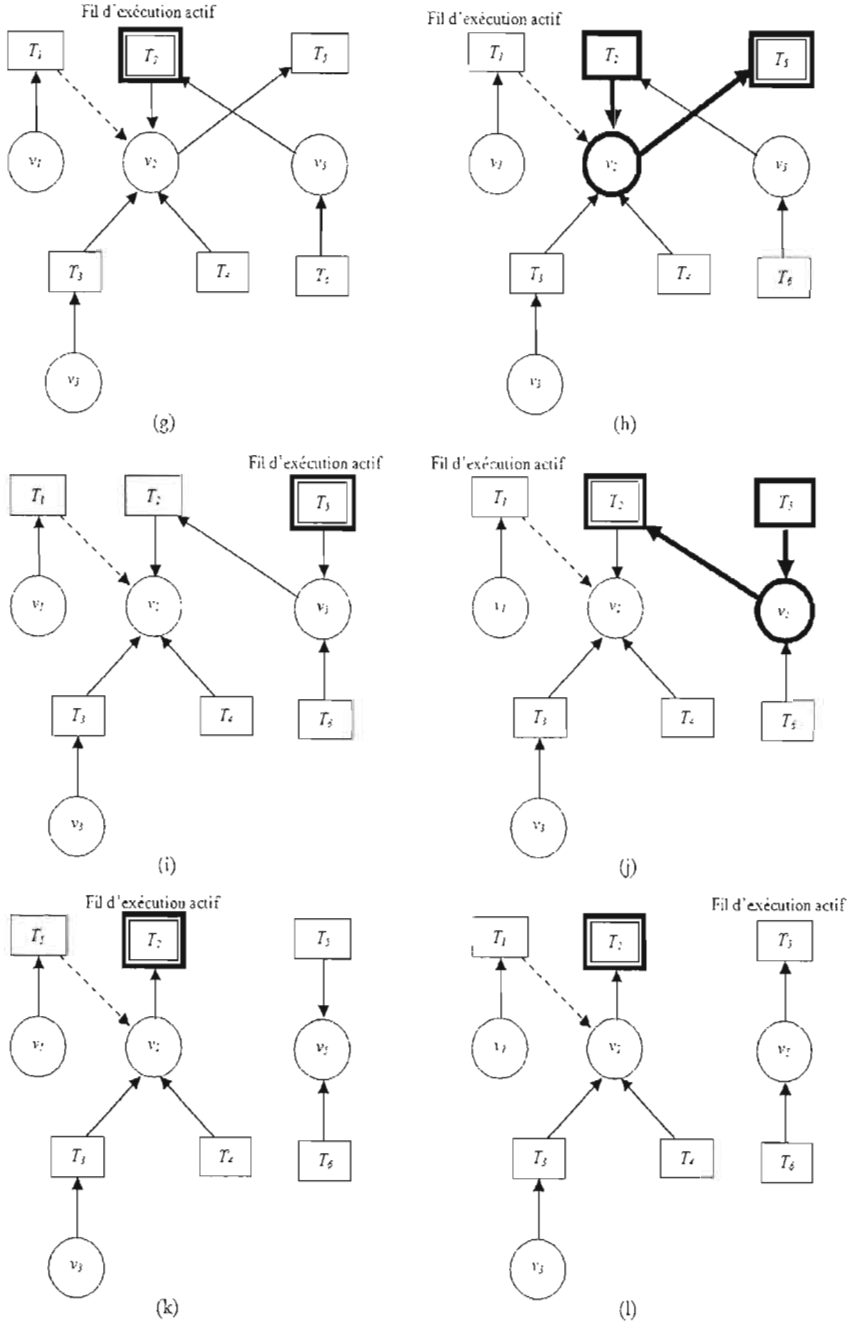
Le fonctionnement de notre algorithme va comme suit (voir figure 4.9) :

0→a) Le fil d'exécution T_1 demande d'acquérir v_2. La procédure DETECTER-INTERBLOCAGE recherche la racine de l'arbre à partir du verrou v_2 jusqu'à ce qu'il atteigne T_5.

a→b) Un temps d'exécution processeur est affecté à T_5, qui relâche le verrou v_5 et demande d'acquérir v_2. T_5 se met en attente. L'arbre d'attente de T_5 est devenu

1. En d'autres mots, les acquisitions et les libérations de verrous se feraient de façon séquentielle.


Fil d'exécution actif
(a)
Fil d'exécution actif
(b)
Fil d'exécution actif
(c)
Fil d'exécution actif
(d)
Fil d'exécution actif
(e)
Fil d'exécution actif
(f)


Fil d'exécution actif
(g)
Fil d'exécution actif
(h)
Fil d'exécution actif
(i)
Fil d'exécution actif
(j)
Fil d'exécution actif
(k)
Fil d'exécution actif
(l)

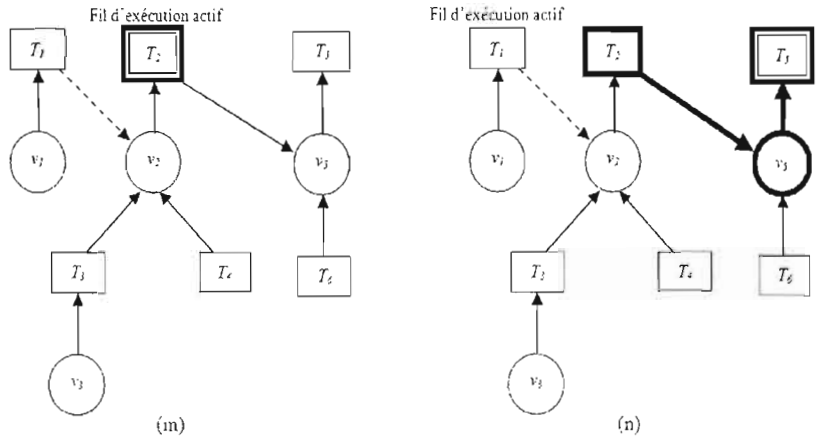


Figure 4.9 Modifications concurrentes - exemple d'une recherche infinie

un sous-arbre de l'arbre de T_2.

b→c) Un temps d'exécution processeur est affecté à T_2, qui acquiert le verrou v_5.

c→d) Un temps d'exécution processeur est affecté à T_1, qui continue sa recherche de la racine. La dernière location de la recherche était T_5. T_1 continue la recherche de la racine à partir de T_5 jusqu'à ce qu'il atteigne T_2.

d→e) Un temps d'exécution processeur est affecté à T_2, qui relâche v_2.

e→f) Un temps d'exécution processeur est affecté à T_5, qui acquiert v_2.

f→g) Un temps d'exécution processeur est affecté à T_2, qui demande d'acquérir v_2. T_2 se met en attente. L'arbre de T_2 est devenu un sous-arbre de l'arbre de T_5.

g→h) Un temps d'exécution processeur est affecté à T_1, qui continue sa recherche de la racine. La dernière location de la recherche était T_2. T_1 continue la recherche de la racine à partir de T_2, jusqu'à ce qu'il atteigne T_5.

h→i) Un temps d'exécution processeur est affecté à T_5, qui relâche v_2 et demande d'acquérir v_5. T_5 se met en attente. L'arbre de T_5 est devenu un sous-arbre de l'arbre de T_2.

i→j) Un temps d'exécution processeur est affecté à T_1, qui continue sa recherche de la

racine. La dernière location de la recherche était T_5. T_1 continue la recherche de la racine à partir de T_5 jusqu'à ce qu'il atteigne T_2.

j→k) Un temps d'exécution processeur est affecté à T_2, qui relâche v_5 et acquiert v_2.

k→l) Un temps d'exécution processeur est affecté à T_5, qui acquiert v_5.

l→m) Un temps d'exécution processeur est affecté à T_2, qui demande d'acquérir v_5. T_2 se met en attente. L'arbre de T_2 est devenu un sous-arbre de l'arbre de T_5.

m→n) Les deux figures 4.9-a et 4.9-n sont similaires. Les étapes de *a-m* se répètent indéfiniment.

Nous avons vu, dans cet exemple, une recherche infinie lorsque les arbres d'attente échangent des verrous, les arbres de T_2 et T_5 échangent v_2 et v_5, et l'algorithme continue la recherche de la racine d'un arbre à l'autre, de l'arbre de T_2 à l'arbre de T_5 et de l'arbre de T_5 à l'arbre de T_2, en créant un cycle infini de recherche.

4.5.2 Fausse détection des interblocages

Il peut arriver à notre algorithme de détecter qu'un cycle se créera si le fil d'exécution actif se met en attente d'un verrou, alors que dans les faits, le cycle était brisé à un noeud après que l'algorithme ait visité ce noeud. Ce brisement dans le cycle, sans que notre algorithme le détecte, conduit à une fausse détection de l'interblocage.

L'exemple de la figure 4.10 illustre un cas d'une fausse détection d'un interblocage. Le fil d'exécution T_1 demande d'acquérir le verrou v_2, qui est acquis par le fil d'exécution T_2. T_1 exécute la procédure ATTENTE-VERROU(T_1, v_2) (algorithme 5), qui appelle la procédure DETECTER-INTERBLOCAGE(T_1, v_2) (algorithme 4).

Le fonctionnement de notre algorithme va comme suit (voir la figure 4.10) :

0→a) Le fil d'exécution T_1 demande le verrou v_2. La procédure DETECTER-INTERBLOCAGE recherche la racine de l'arbre d'attente à partir du verrou v_2 jusqu'à ce qu'il atteingne T_2.

a→b) Un temps d'exécution processeur est affecté à T_2, qui relâche le verrou v_2 et

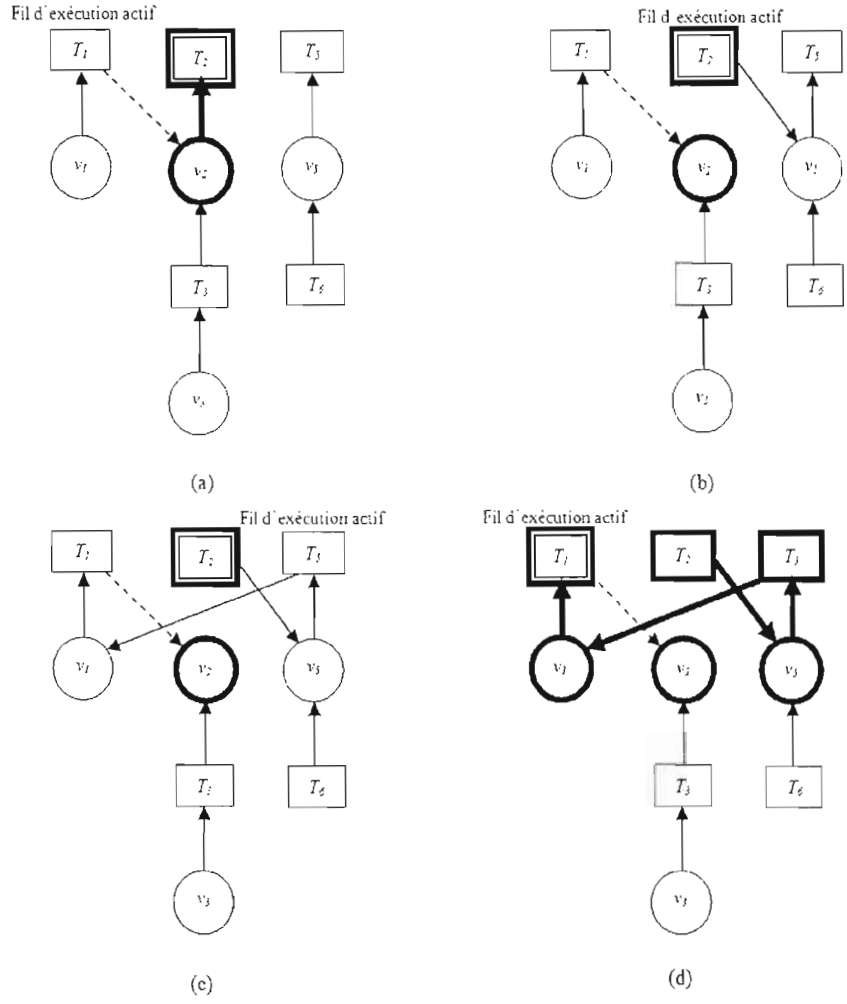


Figure 4.10 Modifications concurrentes - exemple d'une fausse détection d'un interblocage

demande d'acquérir v_5. T_2 se met en attente. L'arbre de T_2 est devenu un sous-arbre de l'arbre de T_5.

b→c) Un temps d'exécution processeur est affecté à T_5, qui demande le verrou v_1. T_5 se met en attente. L'arbre de T_5 est devenu un sous-arbre de l'arbre de T_1.

c→d) Un temps d'exécution processeur est affecté à T_1, qui continue sa recherche de la racine. La dernière location de la recherche était T_2. T_1 continue la recherche de la racine à partir de T_2, jusqu'à ce qu'il atteigne T_1. La procédure DETECTER-INTERBLOCAGE retourne la valeur «vrai» (ligne 6) pour indiquer qu'un interblocage est détecté (une fausse détection d'un interblocage). La procédure ATTENTE-VERROU soulève une exception (ligne 3).

4.5.3 Non détection des interblocages

Il peut arriver à deux ou plusieurs fils d'exécution de lancer l'algorithme de détection des interblocages concurremment, de sorte que chacun des fils d'exécution trouve qu'il ne crée pas d'interblocage et se mette en attente d'un verrou. Mais, dans les faits, les attentes créent un cycle et les fils d'exécution se retrouvent en interblocage.

L'exemple de la figure 4.11 illustre un cas de non détection d'un interblocage. Chacun des deux fils d'exécution T_1 et T_2 demande d'acquérir un verrou qui se trouve dans l'arbre de l'autre fil d'exécution. T_1 demande d'acquérir v_2, et T_2 demande d'acquérir v_1. Les deux fils d'exécution T_1 et T_2 exécutent ATTENTE-VERROU(T_1,v_2) et ATTENTE-VERROU(T_2, v_1) respectivement (algorithme 5), qui appellent la procédure DETECTER-INTERBLOCAGE (algorithme 4).

Le fonctionnement de notre algorithme va comme suit (voir la figure 4.11) :

0→a) Le fil d'exécution T_1 demande le verrou v_2, qui est déjà acquis par T_2.

a→b) Un temps d'exécution processeur est affecté à T_2, qui demande le verrou v_1 qui est déjà acquis par T_1.

b→c) Un temps d'exécution processeur est affecté à T_1, qui recherche la racine de l'arbre d'attente à partir du verrou v_2, jusqu'à ce qu'il soit confronté au fait que

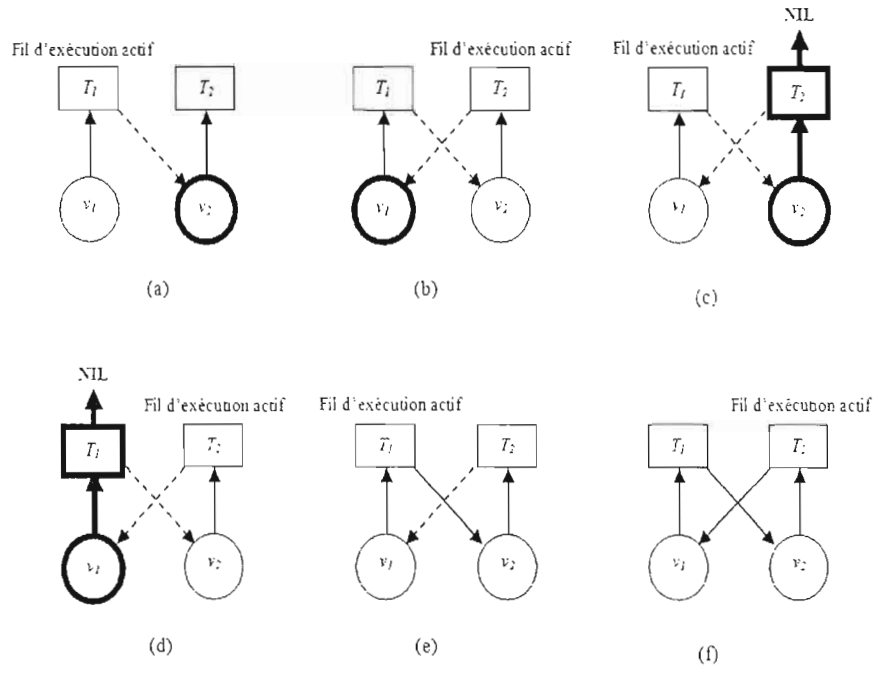


Figure 4.11 Modifications concurrentes - exemple de non détection des interblocages

la racine de l'arbre d'attente est différente de T_1. La procédure DETECTER-INTERBLOCAGE retourne la valeur «faux» (l'algorithme 4, ligne 6) pour indiquer qu'aucun interblocage n'est détecté.

c→d) Un temps d'exécution processeur est affecté à T_2, qui recherche la racine de l'arbre d'attente à partir du verrou v_1, jusqu'à ce qu'il soit confronté au fait que la racine de l'arbre d'attente est différente de T_2. La procédure DETECTER-INTERBLOCAGE retourne la valeur «faux» (l'algorithme 4, ligne 6) pour indiquer qu'aucun interblocage n'est détecté.

d→e) Un temps d'exécution processeur est affecté à T_1, qui se met en attente du verrou v_2.

e→f) Un temps d'exécution processeur est affecté à T_2, qui se met en attente du verrou v_1.

Les deux fils d'exécution T_1 et T_2 sont entrés en interblocage.

4.6 Vision complète de l'algorithme de détection immédiate des interblocages

Dans cette section, nous présenterons les versions incrémentales de la version complète de notre algorithme de détection immédiate des interblocages qui prend en considération les changements dans la forêt d'attente pendant l'exécution de l'algorithme.

Nous présenterons, dans la sous-section 4.6.1, la version de notre algorithme qui résout le problème de la recherche infinie. Dans la sous-section 4.6.2, nous exposerons la version de l'algorithme qui résout le problème de la fausse détection des interblocages. Finalement, nous présenterons, dans la sous-section 4.6.3, la version finale de l'algorithme qui résout le problème de la non détection des interblocages.

4.6.1 Recherche bornée

Nous avons vu, dans sous-section 4.5.1, que les modifications concurrentes dans la machine virtuelle peuvent nous conduire à une recherche infinie. Notre solution pour

éliminer la recherche infinie est de borner le nombre des fils d'exécution visités, pendant le parcours à partir du verrou demandé jusqu'à la racine, au nombre total des fils d'exécution *total-fils* existants dans la machine virtuelle avant le début de la recherche. Nous introduisons dans notre algorithme un compteur *visite* qui s'incrémente à chaque visite d'un fil d'exécution pendant la recherche de la racine.

La procédure DETECTER-INTERBLOCAGE (voir algorithme 6) initialise la variable *total-fils* en appelant la fonction TOTAL-FILS-EXECUTION(), qui retourne le nombre de fils d'exécution existants dans la machine virtuelle au moment de l'appel (ligne 2). La variable *visite* est initialisée à la valeur 1 (ligne 4). Cette variable est incrémentée à chaque visite d'un fil d'exécution (ligne 7). Une autre condition qui a été ajoutée à l'ensemble des tests d'arrêt de la recherche de la racine est $visite \leq total\text{-}fils$ (ligne 5). Cette condition permet d'arrêter la recherche si le nombre de fils d'exécution visités *visite* est supérieur au *total-fils*.

```
1 DETECTER-INTERBLOCAGE(f, v)
2 total-fils←TOTAL-FILS-EXECUTION()
3 verrou←v
4 visite←1
5 tant que visite ≤ total-fils et verrou ≠ NIL et verrou.propriétaire ≠ f faire
6 |   verrou←verrou.propriétaire.verrou_attente
7 |   visite←visite + 1
8 fin tant que
9 retourner visite ≤ total-fils et verrou ≠ NIL et verrou.propriétaire = f
```

Algorithme 6 : Recherche bornée

Dans un autre cas de recherche infinie, illustré à la figure 4.12, la recherche de la racine continue indéfiniment d'un arbre à un nouvel arbre. Ce nouvel arbre a été créé entre deux temps d'exécution processeur affectés au fil d'exécution actif. Avant que la recherche atteigne le sommet du dernier arbre, celui-ci se met en attente d'un verrou qui se trouve dans le nouvel arbre.

Le fonctionnement de la nouvelle version de notre algorithme dans l'exemple de la figure 4.12 va comme suit :

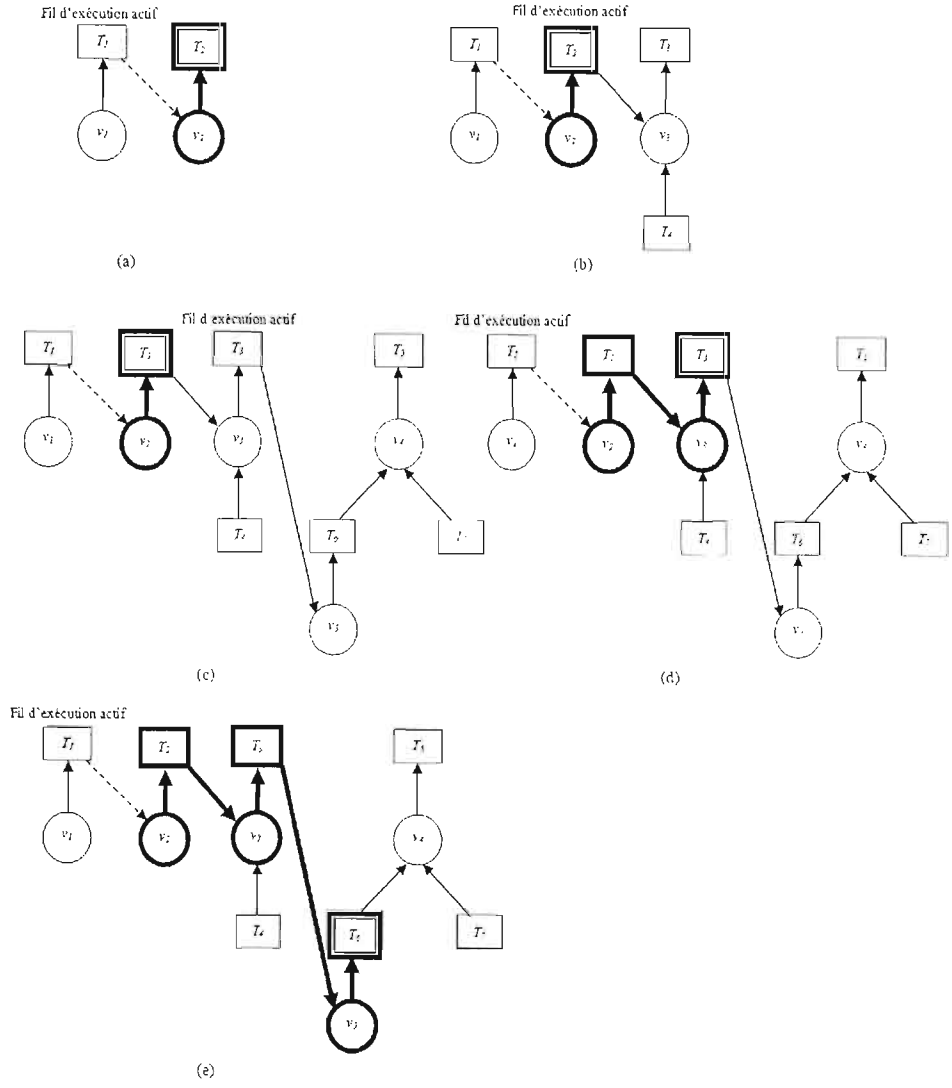


Figure 4.12 Exemple d'une recherche bornée

0→a) Le fil d'exécution T_1 demande d'acquérir le verrou v_2, qui est verrouillé par T_2. Notre algorithme (algorithme 6) recherche la racine de l'arbre à partir du verrou v_2. Il initialise la variable *total-fils* par le nombre de fils d'exécution existants avant le début de la recherche, et qui est 2 (ligne 2), ainsi que la variable *visite* par la valeur 1 (ligne 4).

a→b) Un temps d'exécution processeur est affecté à T_2, qui demande d'acquérir le verrou v_3 (l'arbre de T_3 a été créé avant l'affectation d'un temps d'exécution processeur à T_2). T_2 se met en attente. L'arbre de T_2 est devenu un sous-arbre de l'arbre T_3.

b→c) Un temps d'exécution processeur est affecté à T_3, qui demande le verrou v_5 (l'arbre de T_5 a été créé avant l'affectation du temps d'exécution processeur à T_3). T_3 se met en attente. L'arbre de T_3 est devenu un sous-arbre de l'arbre de T_5.

c→d) Un temps d'exécution processeur est affecté à T_1. La dernière location de la recherche était T_2 (*visite*=1, *total-fils*=2). T_1 continue sa recherche de la racine à partir de T_2 jusqu'à ce qu'il atteigne T_3 (le compteur incrémenté *visite*=2). La condition $visite \leq 2$ **et** $verrou \neq NIL$ **et** $v_2.propriétaire \neq f$ est vraie. Le compteur *visite* est incrémenté à 3 et *verrou* reçoit la valeur v_5.

d→e) T_1 exécute une autre itération de la boucle **tant que** (*visite*=3, *total-fils*=2). La condition $visite \leq 2$ est fausse, c'est-à-dire que la recherche a atteint le nombre maximum *total-fils* de fils d'exécution à visiter. La procédure DETECTER-INTERBLOCAGE retourne la valeur «faux» pour indiquer qu'aucun interblocage n'est détecté.

La recherche bornée permet d'arrêter la recherche infinie de la racine après que l'algorithme ait atteint le nombre de fils d'exécution existants dans la machine virtuelle avant le début de la recherche. Nous avons remarqué, dans l'exemple de la figure 4.12, que l'algorithme a arrêté la recherche après avoir visité les fils d'exécution T_2 et T_3.

4.6.1.1 Analyse de l'algorithme de la recherche bornée

Analysons notre nouvelle version de l'algorithme au niveau de la création ou non d'interblocages par le fil d'exécution actif :

1. Le fil d'exécution actif crée un interblocage. Ce qui donne lieu à deux cas :
   (a) Le compteur *visite* n'a pas dépassé *total-fils*. La procédure a trouvé que le sommet de l'arbre d'attente est le fil d'exécution actif. Celle-ci retourne la valeur «vrai» (ligne 9), ou :
   (b) Le compteur *visite* a dépassé *total-fils*. La procédure arrête la recherche et retourne la valeur «faux» (ligne 9). La procédure n'a pas détecté l'interblocage. Ce cas advient lorsque d'autres fils d'exécution ont été créés après le début de la détection de l'interblocage, et ces fils d'exécution ont créé un cycle vers le fil d'exécution actif. Le nombre total des fils d'exécution dans le cycle est supérieur à *total-fils*. C'est un cas de la non détection d'interblocage, qui sera détecté par notre version finale de l'algorithme (voir sous-section 4.6.3).
2. Ou le fil d'exécution actif ne crée pas d'interblocage. Deux cas sont possibles :
   (a) Le compteur *visite* n'a pas dépassé *total-fils*. La procédure a trouvé démontré que le sommet de l'arbre d'attente est différent du fil d'exécution actif. La procédure retourne la valeur «faux» (ligne 9), ou :
   (b) Le compteur *visite* a dépassé *total-fils*. Il s'agit du cas où le nombre de fils d'exécution à visiter pour trouver le sommet est supérieur à *total-fils* (la recherche infinie en fait partie). La procédure met fin à la recherche et retourne la valeur «faux» (ligne 9).

La boucle **tant que** des lignes 5-8 s'exécute en *O(total-fils)*, et les lignes 2-4 et 9 s'exécutent en *O(1)*. Donc, dans le cas le plus défavorable, la procédure DETECTER-INTERBLOCAGE (algorithme 6) s'exécute en *O(total-fils)*.

4.6.2 Recherche récursive

Nous avons vu dans la sous-section 4.5.2 que les modifications concurrentes dans la machine virtuelle peuvent nous conduire à une fausse détection des interblocages. Une détection d'un interblocage signifie que le fil d'exécution a demandé un verrou qui se trouve dans son sous-arbre d'attente, c'est-à-dire qu'un cycle se créera à partir du fil d'exécution actif vers lui-même, si le fil d'exécution actif se met en attente du verrou demandé [2]. La solution pour éviter la fausse détection des interblocages est de s'assurer que le cycle, non fermé, n'a pas changé pendant la recherche de la racine. Deux cas sont possibles :

- Soit c'est une détection d'interblocage, alors tous les fils d'exécution dans le cycle sont en attente de verrous ;
- Soit c'est une fausse détection d'interblocage, alors il n'y a pas un vrai cycle. Les fils d'exécution visités pendant la recherche ont subi des modifications qui ont brisé le cycle.

Nous avons ajouté dans notre algorithme (algorithme 7) une vérification pour nous assurer que le cycle n'a pas subi de modifications, en sauvegardant l'état de chaque fil d'exécution visité dans le chemin de la recherche de la racine (c'est-à-dire les deux verrous, le verrou acquis et le verrou en attente d'acquisition, qui appartiennent au chemin de la recherche de la racine). Si un interblocage est détecté, alors nous vérifions que les deux verrous sauvegardés pour chaque fil d'exécution visité dans le cycle n'ont pas changé par rapport aux valeurs actuelles des deux verrous dans la forêt d'attente. Lorsque la vérification échoue, le cycle a changé et c'est une fausse détection.

La procédure DETECTER-INTERBLOCAGE(f, v) (algorithme 7) initialise la variable *total-fils* en appelant la fonction TOTAL-FILS-EXECUTION(), qui retourne le nombre de fils d'exécution existants dans la machine virtuelle au moment de l'appel (ligne 1) et initialise la variable *visite* à 1 (ligne 2). La procédure appelle la procédure récursive

2. Notons que dans notre version finale, à la sous-section 4.6.3, le cycle se crée avant que le fil d'exécution actif rentre dans un interblocage.

DETECTER-PAR-RECURSION(*f*, *v*, *visite*, *total-fils*) (algorithme 8). La procédure DETECTER-INTERBLOCAGE retourne la valeur retournée par DETECTER-PAR-RECURSION.

```
1 DETECTER-INTERBLOCAGE(f, v)
2 total-fils←TOTAL-FILS-EXECUTION()
3 visite←1
4 retourner DETECTER-PAR-RECURSION (f, v, visite, total-fils)
```

Algorithme 7 : Recherche récursive

```
1 DETECTER-PAR-RECURSION(f, verrou, visite, total-fils)
2 si visite ≤ total-fils et verrou ≠ NIL et verrou.propriétaire ≠ f alors
3 |   visite←visite + 1
4 |   fil-propriétaire←verrou.propriétaire
5 |   verrou-attente←fil-propriétaire.vcrrou_attente
6 |   résultat←DETECTER-PAR-RECURSION(f, verrou-attente, visite,
  |   total-fils)
7 |   retourner résultat et fil-propriétaire=verrou.propriétaire et
  |   verrou-attente=fil-propriétaire.verrou_attente
8 fin si
9 retourner visite ≤ total-fils et verrou ≠ NIL et verrou.propriétaire = f
```

Algorithme 8 : Recherche par récursion

La procédure DETECTER-PAR-RECURSION(*f*, *verrou*, *visite*, *total-fils*) prend en entrée le fil d'exécution actif *f*, le verrou *verrou* en visite pendant la récursion actuelle, la variable *visite* qui est incrémentée à chaque visite d'un fil d'exécution pendant la recherche de la racine et la variable *total-fils*, qui contient le nombre de fils d'exécution passés par la procédure DETECTER-INTERBLOCAGE(*f*, *v*) et qui a toujours la même valeur dans tous les appels récursifs. Les étapes détaillées de la procédure sont :

1. La procédure vérifie si le nombre de fil d'exécution visités est inférieur ou égal à *total-fils* et si nous avons atteint le sommet de l'arbre ou non. Si nous n'avons pas encore dépassé *total-fils* et que nous n'avons pas atteint le sommet de l'arbre, alors nous incrémentons la variable *visite* (ligne 2) pour indiquer que nous avons visité un fil d'exécution en plus.
2. Ensuite, nous initialisons les variables *fil-propriétaire* et *verrou* (ligne 4 et 5) respectivement par le fil d'exécution propriétaire du verrou visité *verrou* et le verrou

que le fil d'exécution *fil-propriétaire* est en attente d'acquérir.

3. Après, nous appelons par récursion la procédure DETECTER-PAR-RECURSION(*f*, *verrou-attente*, *visite*, *total-fils*) avec les nouveaux paramètres initialisés dans les lignes 3 à 5. L'appel récursif de la procédure DETECTER-PAR-RECURSION a deux conditions d'arrêt :
   (a) soit que le compteur *visite* a dépassé *total-fils* de fils d'exécution (*visite* $\leq$ *total-fils* devient fausse). Dans ce cas, la procédure retourne la valeur «faux», ou
   (b) soit que la procédure a atteint le sommet de l'arbre d'attente (*verrou* $\neq NIL$ et *verrou.propriétaire* $\neq f$ devient fausse). Dans ce cas, si le fil d'exécution actif est la racine de l'arbre, alors la procédure retourne la valeur «vrai», sinon elle retourne la valeur «faux».
4. Au retour de l'appel récursif (ligne 6), la variable *résultat* contient la valeur «vrai» ou «faux» selon la condition d'arrêt :
   (a) Si le résultat est «vrai» et que les valeurs de *fil-propriétaire* et *verrou-attente* n'ont pas changé après l'appel récursif par rapport aux valeurs dans la forêt d'attente, alors la procédure retourne la valeur «vrai», sinon si ces un ou les deux valeurs sont changées, alors la procédure retourne la valeur «faux».
   (b) Si le résultat est «faux», alors la procédure retourne la valeur «faux».
5. À la fin des appels récursifs, nous arrivons à l'un des trois cas suivants :
   (a) Le sommet de l'arbre est le fil d'exécution actif, et le cycle dans l'arbre d'attente du verrou v vers le fil d'exécution f n'a pas changé, alors la procédure DETECTER-INTERBLOCAGE(f, v) retourne la valeur «vrai» pour indiquer qu'un interblocage est détecté, ou
   (b) le sommet de l'arbre est le fil d'exécution actif, mais le cycle dans l'arbre d'attente du verrou v vers le fil d'exécution f a changé, alors la procédure retourne la valeur «faux» pour indiquer que c'est une fausse détection d'interblocage, ou

(c) le sommet de l'arbre n'est pas le fil d'exécution actif, alors la procédure retourne la valeur «faux» pour indiquer qu'aucun interblocage n'est détecté.

Reprenons l'exemple 4.5.1. Le fonctionnement de notre nouvelle version de l'algorithme va comme suit :

0→a) Le fil d'exécution T_1 demande d'acquérir v_2. La procédure DETECTER-INTERBLOCAGE appelle la procédure récursive DETECTER-PAR-RECURSION.

a→b) T_1 recherche la racine de l'arbre d'attente à partir du verrou v_2 jusqu'à ce qu'il atteigne T_2. T_1 sauvegarde le propriétaire du verrou v_2, qui est T_2, dans une variable locale.

b→c) Un temps d'exécution processeur est affeté à T_2, qui relâche le verrou v_2 et demande d'acquérir le verrou v_5. T_2 se met en attente.

c→d) Un temps d'exécution processeur est affecté à T_1, qui continue sa recherche de la racine. La dernière location de la recherche était T_2. T_1 continue sa recherche à partir T_2, jusqu'à ce qu'il atteigne T_1 (l'algorithme a sauvegardé dans des variables locales tous les propriétaires des verrous visités et tous les verrous que les fils d'exécution visités sont en attente d'acquérir). La procédure récursive DETECTER-PAR-RECURSION a détecté un interblocage, parce que le fil d'exécution actif est la racine de l'arbre. La procédure DETECTER-PAR-RECURSION retourne la valeur «vrai» (ligne 9) pour indiquer qu'un interblocage est détecté.

d→e) T_1 vérifie en retournant des récursions de la procédure DETECTER-PAR-RECURSION que le cycle n'a pas changé pendant la recherche. Le premier retour de récursion retourne avec l'état *fil-propriétaire* égal T_5 et à *verrou* égal à v_5. L'algorithme vérifie (ligne 7) que la variable *résultat* retournée de l'étape (d) est vraie, ce qui est le cas, vérifie que le propriétaire du verrou v_5 est toujours T_5 (*fil-propriétaire=verrou.propriétaire*, la valeur sauvegardée de *fil-propriétaire* est T_5), ce qui est le cas, et vérifie que T_5 est toujours en attente du verrou v_1 (*verrou-attente=fil-propriétaire.verrou_attente*, la valeur sauvegardée de *verrou-attente* est v_1), ce qui est le cas. La procédure DETECTER-PAR-RECURSION retourne la

valeur «vrai» pour indiquer que le cycle détecté n'a pas changé pour ce retour de récursion.

e→f) T_1 continue de vérifier que le cycle détecté n'a pas changé. Le deuxième retour de récursion retourne avec l'état *fil-propriétaire* égal à T_2 et *verrou* égal à v_2. L'algorithme vérifie (ligne 7) que la variable *résultat* retournée de l'étape (e) est vraie, ce qui est le cas, vérifie que le propriétaire du verrou v_2 est toujours T_2 (*fil-propriétaire=verrou.propriétaire*, la valeur sauvegardée de *fil-propriétaire* est T_2), ce qui n'est pas le cas, parce que T_2 a relâché le verrou v_2 après que l'algorithme ait visité le fil d'exécution T_2. La procédure DETECTER-PAR-RECURSION retourne la valeur «faux» pour indiquer que le cycle détecté a changé pour ce retour de récursion. La procédure DETECTER-PAR-RECURSION retourne toujours la valeur «faux» pour les autres récursions, une fois qu'elle arrive au résultat que le cycle a changé.

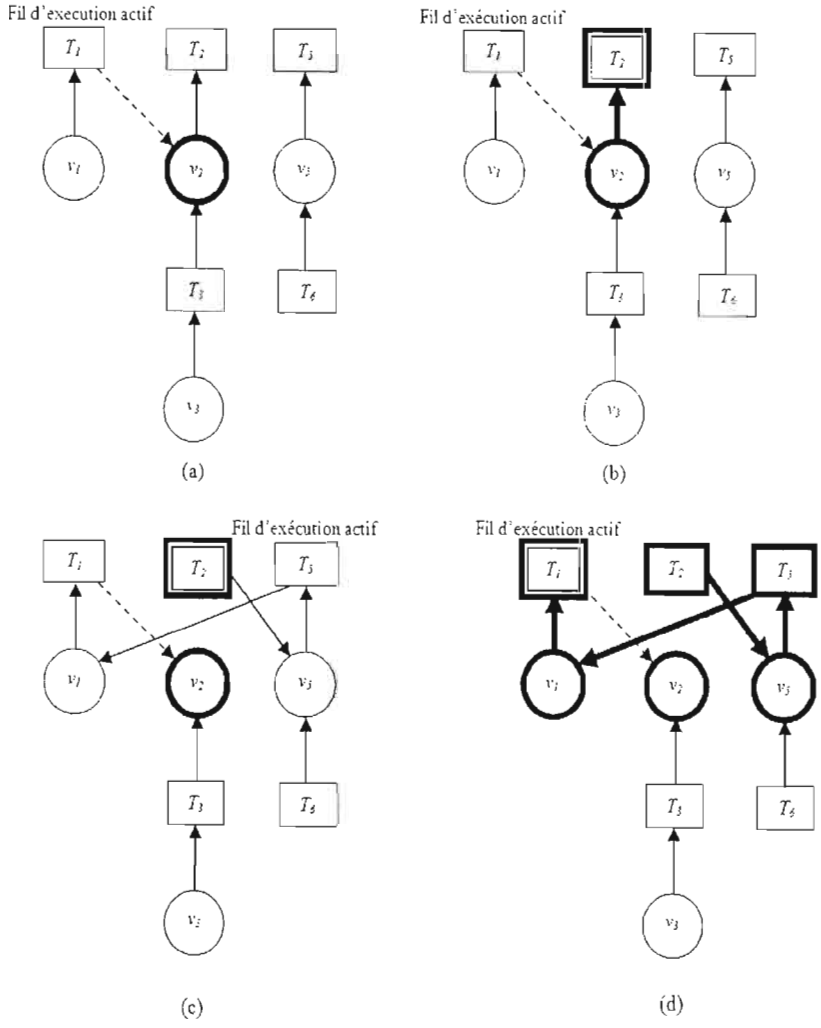
Le fil d'exécution T_1 arrive au résultat que le cycle détecté a changé, alors c'est une fausse détection d'interblocage. Le fil d'exécution T_1, après retour des récursions de la procédure DETECTER-PAR-RECURSION avec une valeur «faux», continue son exécution dans la procédure DETECTER-INTERBLOCAGE (algorithme 7, ligne 4), qui retourne la valeur «faux» pour indiquer qu'aucun interblocage n'est détecté.

4.6.2.1 Analyse de l'algorithme de la recherche par récursion

Les appels récursifs de DETECTER-PAR-RECURSION de la ligne 6 s'exécutent en *O(total-fils)*, et les lignes 2-5 et 7-9 s'exécutent en *O(1)*. Donc, dans le cas le plus défavorable, la procédure DETECTER-PAR-RECURSION (algorithme 8) s'exécute en *O(total-fils)*.

4.6.2.2 Preuves de rectitude de l'algorithme de la recherche par récursion

Nous devons démontrer que lorsque l'algorithme 7 détecte un interblocage, c'est qu'il y a effectivement un interblocage (donc, que la fausse détection est éliminée). Nous


Fil d'execution actif
Fil d'exécution actif
Fil d'exécution actif
Fil d'exécution actif
(a)
(b)
(c)
(d)

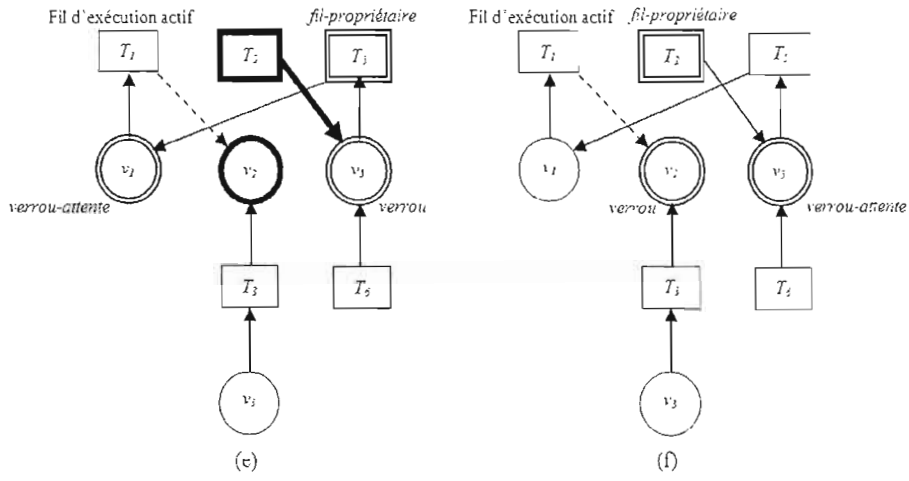


Figure 4.13 Recherche par récursion

commençons en prouvant la proposition suivante par induction :

Proposition : Après le $n^{\text{ième}}$ retour de récursion de DETECTER-PAR-RECURSION de l'algorithme 8, à la ligne 7, la partie du cycle formée par le verrou courant *verrou* et son propriétaire jusqu'à f ne peut plus changer.

Hypothèse : L'algorithme 8 retourne vrai.

Pré-condition : Assertion dans l'algorithme 5 (ligne 2).

Cas de base : L'appel récursif le plus imbriqué se termine à la ligne 9. À ce moment, le propriétaire du verrou courant est le fil d'exécution f. Après le premier retour de récursion, le propriétaire du verrou courant est le fil d'exécution f_1 qui est en attente d'un verrou acquis par f. Puisque f est le fil d'exécution qui est en train de détecter l'interblocage, c'est certain qu'il ne relâchera aucun verrou acquis par lui-même. Puisque f_1 est en attente d'un verrou qui ne sera pas relâché, il restera en attente et ne pourra relâcher aucun verrou acquis par lui-même, incluant le verrou courant. Donc, la partie du cycle formée par le verrou courant *verrou* et son propriétaire f_1 jusqu'à f ne peut

plus changer après un retour de récursion.

Hypothèse d'induction : Supposons que la partie du cycle formée par le verrou courant *verrou* et son propriétaire jusqu'à f ne peut plus changer après le $n^{\text{ième}}$ retour de récursion de DETECTER-PAR-RECURSION.

Preuve pour $n+1$: Après le $(n+1)^{\text{ième}}$ retour de récursion, le propriétaire du verrou courant *verrou* est le fil d'exécution f_{n+1}, qui est en attente d'un verrou acquis par f_n. L'hypothèse d'induction nous indique que le verrou acquis par f_n ne sera pas relâché. Puisque f_{n+1} est en attente d'un verrou qui ne sera pas relâché, il restera en attente et ne pourra relâcher aucun verrou acquis par lui-même, incluant le verrou courant. Donc, la partie du cycle formée par le verrou courant *verrou* et son propriétaire f_{n+1} jusqu'à f ne peut plus changer après $n+1$ retours de récursion.

Conclusion : La proposition est vraie pour tout n.

En supposant que l'algorithme 7 retourne vrai, on constate à la ligne 4, à l'aide de la proposition prouvée, que la partie du cycle formée par le verrou v jusqu'à f ne peut plus changer. Donc, si f se met en attente de v, il y aura nécessairement un interblocage.

4.6.3 Élimination de la non détection des interblocages

Nous avons vu dans l'exemple 4.5.3 que les modifications concurrentes dans la machine virtuelle peuvent nous conduire à la non détection des interblocages. Au moment même où le fil d'exécution T_1 fait sa recherche de la racine de l'arbre d'attente du verrou v_2, le fil d'exécution sommet T_2 du verrou v_2 est aussi en train de faire la recherche de la racine de l'arbre du verrou v_1 qui est acquis par T_1 sans que l'un ou l'autre des deux fils d'exécution sache que l'autre est en train d'essayer d'acquérir un verrou qui appartient à son arbre. Cette absence de connaissance conduit, avec des changements du temps d'exécution processeur d'un fil d'exécution vers l'autre, à un interblocage.

La solution à ce problème se trouve avant chaque début de détection d'interblocage :

- Le fil d'exécution actif indique qu'il est en attente du verrou demandé en initiali-

sant le champ *fil-d'exécution-actif.verrou_attente* ← *verrou-demandé* avant l'appel de la procédure DETECTER-INTERBLOCAGE.

- Deux cas sont possibles :
  - Si le fil d'exécution actif détecte un interblocage, alors il remet le *fil-d'exécution-actif.verrou_attente* à *NIL* pour indiquer qu'il n'est plus en attente, ensuite il soulève une exception, mais
  - s'il ne détecte pas un interblocage, alors l'initialisation de la case *fil-d'exécution-actif.verrou_attente* est déjà faite.

Si un autre fil d'exécution essaie d'acquérir un verrou dans l'arbre d'attente d'un fil d'exécution qui lui aussi essaie d'acquérir un verrou qui se trouve dans son arbre d'attente, alors quelles que soient les séquences d'exécution des deux fils d'exécution et avant qu'ils détectent qu'il existe un interblocage ou non, ils initialisent les cases *fil-d'exécution.verrou_attente* correspondant à eux avec les verrous qu'ils demandent. Quand ils lancent la détection d'interblocage, un fil d'exécution ou les deux vont finir par trouver un cycle, ce qui signifie une détection d'interblocage.

Nous avons modifié la procédure ATTENTE-VERROU(f, v) (algorithme 9), de sorte que l'algorithme initialise la case *f.verrou_attente*←v (ligne 3) pour indiquer que le fil d'exécution actif f est en attente du verrou v avant que l'algorithme appelle la procédure DETECTER-INTERBLOCAGE(f, v), qui va détecter s'il existe un interblocage ou non. Si un interblocage est détecté, la procédure DETECTER-INTERBLOCAGE retourne la valeur «vrai», et les deux lignes 5 et 6 vont être exécutées. La procédure ATTENTE-VERROU remet la case *f.verrou_attente*←NIL (ligne 5), ensuite elle soulève une exception (ligne 6). Dans l'autre cas, aucun interblocage n'est détecté ; la procédure ATTENTE-VERROU ne fait aucune action.

Reprenons l'exemple 4.5.3, mais cette fois nous utilisons notre nouvel algorithme ATTENTE-VERROU (voir algorithme 9). La figure 4.14 illustre l'exécution de notre algorithme.

Le fonctionnement de la nouvelle version de notre algorithme va comme suit :

0→a) Le fil d'exécution T_1 demande d'acquérir le verrou v_2, qui est déjà acquis par T_2.

```
1 ATTENTE-VERROU(f, v)
2 assert v.propriétaire ≠ NIL et v.propriétaire ≠f
3 f.verrou_attente←v
4 si DETECTER-INTERBLOCAGE(f, v) alors
5 |   f.verrou_attente←NIL
6 |   soulever-exception
7 fin si
```

Algorithme 9 : Attente d'un verrou modifiée - version complète de l'algorithme

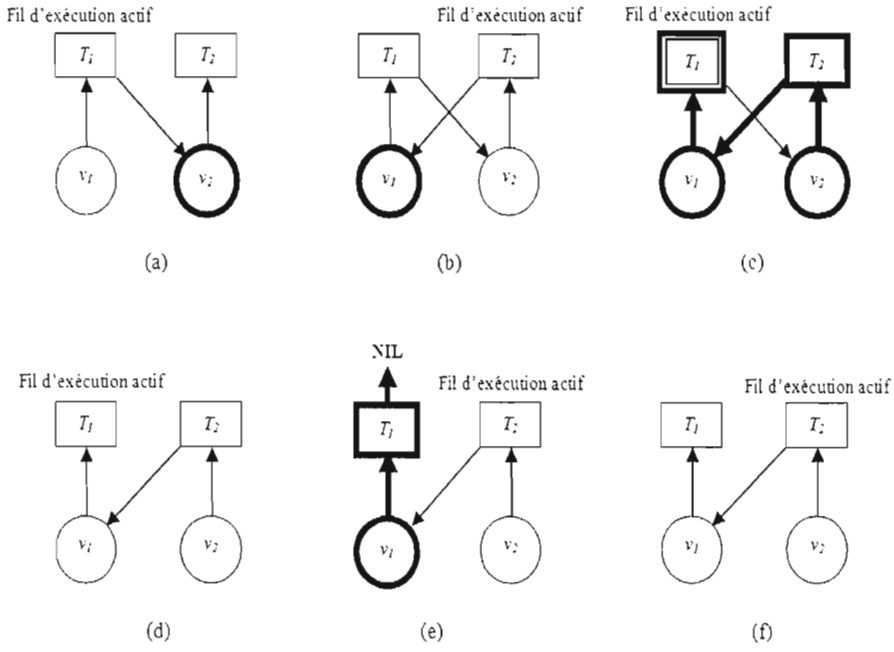


Figure 4.14 Élimination de la non détection des interblocages

T_1 appelle la procédure ATTENTE-VERROU, qui indique qu'il est en attente du verrou v_2 (initialise le champ $T_1.verrou\_attente \leftarrow v_2$).

a→b) Un temps d'exécution processeur est affecté à T_2, qui demande d'acquérir le verrou v_1, qui est déjà acquis par T_1. T_2 appelle la procédure ATTENTE-VERROU, qui indique qu'il est en attente du verrou v_2 (initialise le champ $T_2.verrou\_attente \leftarrow v_1$).

b→c) Un temps d'exécution processeur est affecté à T_1, qui recherche la racine de l'arbre d'attente à partir du verrou v_2, jusqu'à ce qu'il trouve que la racine de l'arbre d'attente est lui-même (détection d'un interblocage). La procédure DETECTER-INTERBLOCAGE retourne la valeur «vrai».

c→d) Le fil d'exécution T_1 continue son exécution dans la procédure ATTENTE-VERROU, il annule son attente du verrou v_2 ($T_1.verrou_attente \leftarrow$NIL), ensuite soulève une exception pour briser l'interblocage.

d→e) Un temps d'exécution processeur est affecté à T_2, qui recherche la racine de l'arbre d'attente à partir du verrou v_1 jusqu'à ce qu'il trouve que la racine de l'arbre d'attente est différente de T_2. La procédure DETECTER-INTERBLOCAGE retourne la valeur «faux».

e→f) Le fil d'exécution T_2 continue son exécution dans la procédure ATTENTE-VERROU sans aucune action, parce que il a déjà indiqué qu'il est en attente du verrou v_1.

4.6.3.1 Analyse de l'algorithme de l'élimination de la non détection des interblocages

Supposons que deux fils d'exécution f_1 et f_2 demandent d'acquérir deux verrous v_2 et v_1 respectivement, où v_1 est acquis par f_1 et v_2 est acquis par f_2. Le champ $v_2.verrou\_attente$ est devenu f_1 et $v_1.verrou\_attente$ est devenu f_2. Supposons que les deux fils d'exécution ont détecté l'existence d'un cycle dans la même période. Selon l'ordre d'exécution de la détection des modifications, il existe trois cas :

1. f_1 détecte que le cycle n'a pas changé, il remet $v_2.verrou\_attente$ à NIL et il brise l'interblocage. Ensuite, f_2 détecte que le cycle a changé et il acquiert le verrou v_1.

2. f_2 détecte que le cycle n'a pas changé, il remet $v_1.verrou\_attente$ à NIL et il brise l'interblocage. Ensuite, f_1 détecte que le cycle a changé et il acquiert le verrou v_2.
3. f_1 et f_2 détectent concurrement que le cycle n'a pas changé. Les deux fils d'exécution f_1 et f_2 brisent l'interblocage.

Dans les trois cas, notre nouvelle version a éliminé l'interblocage. Le troisième cas peut paraître surprenant, mais il est tout à fait légitime car il y avait une création concurrente d'un même cycle et notre algorithme a brisé l'interblocage lors de la fermeture des derniers liens du cycle.

Il est impossible que f_1 et f_2 détectent concurremment que le cycle a changé. Si f_1 détecte que le cycle a changé, alors f_1 a détecté que $v_1.verrou\_attente$ n'est plus f_2. Cela signifie que f_2 a déjà détecté que le cycle n'a pas changé et qu'il a remis $v_1.verrou\_attente$ à NIL. C'est une contradiction. Le même raisonnement s'applique dans le cas où f_2 détecte que le cycle a changé.

Le cas de deux fils d'exécution se généralise sur plusieurs fils d'exécution f_1,f_2,...,f_{n-1}, f_n demandent d'acquérir des verrous v_2, v_3,...,v_n,v_1 respectivement où v_1 est acquis par f_1 qui demande d'acquérir v_2, v_2 est acquis par f_2 qui demande d'acquérir v_3,...,v_n est acquis par f_n qui demande d'acquérir v_1. Dans ce cas, un ou plusieurs fils d'exécution détectent que le cycle n'a pas changé, et ils brisent l'interblocage dans la même période avant que les autres fils d'exécution détectent que le cycle a changé. C'est possible aussi que tous les fils d'exécution détectent que le cycle n'a pas changé dans la même période et qu'ils brisent l'interblocage.

Revenons sur l'analyse de la recherche bornée dans la section 4.6.1. Le cas où un interblocage aurait été créé après que le fil d'exécution a commencé sa détection de l'interblocage. La procédure DETECTER-INTERBLOCAGE ne détecte pas l'interblocage, parce que le compteur *visite* a dépassé *total-fils*. Après que le fil d'exécution actif a commencé la détection d'interblocages (la variable *total-fils* était initialisée), d'autres fils d'exécution ont été créés entre deux temps d'exécution processeur du fil d'exécution actif. Ces fils d'exécution créés ont créé un cycle du fil d'exécution actif vers lui-même.

Le nombre total des fils d'exécution à visiter pour trouver le cycle est devenu supérieur à *total-fils*. Nous avons mentionné que l'algorithme de la recherche bornée ne détecte pas ce cas d'interblocage. Notre version finale de l'algorithme (algorithme 9) détecte cet interblocage. C'est le fil d'exécution qui fermera le cycle, plutôt que le fil d'exécution actif, qui va détecter l'interblocage.

L'appel de la procédure DETECTER-INTERBLOCAGE (ligne 4) s'exécute en *O(total-fils)*, où *total-fils* est le nombre total des fils d'exécution existants dans la machine virtuelle avant le début de la recherche, et le code de la ligne 3 et 5-6 s'exécute en *O(1)*. Donc, la procédure ATTENTE-VERROU modifiée s'exécute dans le cas le plus défavorable en *O(total-fils)*.

4.7 Brisement de l'interblocage

Quand le fil d'exécution actif, qui demande d'acquérir un verrou, détecte un interblocage, il soulève une exception pour briser l'interblocage. Dans l'implémentation de notre algorithme dans la machine virtuelle Java *SableVM* (voir chapitre 5), nous brisons l'interblocage par le soulèvement de l'exception ***IllegalMonitorException*** (*Runtime Exception*) de Java.

Deux cas sont possibles pour le traitement de l'exception envoyée :

- La première est d'attraper l'exception par un bloc Java **try..catch**, qui permet de gérer l'exception. Le fil d'exécution actif demandeur du verrou continue son exécution. Le code Java suivant montre comment gérer l'exception :

```
try {
  ... // code demande du verrou par le fil d'exécution actif.
} catch(IllegalMonitorException ex){
 ... // gérer l'exception
}
```

- La deuxième est de ne pas attraper l'exception, ce qui va finir par mettre un terme à l'exécution du fil d'exécution actif demandeur du verrou.

4.8 Version finale de l'algorithme de détection immédiate des interblocages

```
1 ACQUISITION-VERROU(f, v)
2 assert v.propriétaire = NIL
3 v.propriétaire←f
```

```
1 ATTENTE-VERROU(f, v)
2 assert v.propriétaire ≠ NIL et v.propriétaire ≠f
3 f.verrou_attente←v
4 si DETECTER-INTERBLOCAGE(f, v) alors
5 |   f.verrou_attente←NIL
6 |   soulever-exception
7 fin si
```

```
1 RELACHER-VERROU(f, v)
2 assert v.propriétaire =f
3 v.propriétaire←NIL
```

```
1 DETECTER-INTERBLOCAGE(f, v)
2 total-fils←TOTAL-FILS-EXECUTION()
3 visite←1
4 retourner DETECTER-PAR-RECURSION (f, v, visite, total-fils)
```

```
1 DETECTER-PAR-RECURSION(f, verrou, visite, total-fils)
2 si visite ≤ total-fils et verrou ≠ NIL et verrou.propriétaire ≠ f alors
3 |   visite←visite + 1
4 |   fil-propriétaire←verrou.propriétaire
5 |   verrou-attente←fil-propriétaire.verrou_attente
6 |   résultat←DETECTER-PAR-RECURSION(f, verrou-attente, visite,
  |   total-fils)
7 |   retourner résultat et fil-propriétaire=verrou.propriétaire et
  |   verrou-attente=fil-propriétaire.verrou_attente
8 fin si
9 retourner visite ≤ total-fils et verrou ≠ NIL et verrou.propriétaire = f
```

4.9 Conclusion

Dans ce chapitre, nous avons présenté la construction de la forêt d'attente pendant l'exécution de la machine virtuelle. Nous avons présenté en premier, une version simplifiée de notre algorithme de détection immédiate des interblocages et nous avons exposé les problèmes liés aux modifications concurrentes. Nous avons présenté la version complète de notre algorithme qui résout les problèmes de la recherche infinie, de la fausse détection des interblocages et de la non détection des interblocages causés par les modifications concurrentes. Nous avons prouvé la rectitude de l'algorithme et analysé sa complexité asymptotique. Finalement, nous avons présenté le brisement des interblocages par le soulèvement d'exceptions.

CHAPITRE V

EXPÉRIMENTATION

Dans ce chapitre, nous présenterons le travail de mesure de performance de notre algorithme par rapport à une version qui ne contient pas notre algorithme de détection immédiate et de brisement de l'interblocage.

Notre algorithme était implémenté sur la machine virtuelle *SableVM* (*SableVM* est une machine virtuelle Java, gratuite, avec code source libre, disponible sous les termes de *GNU Lesser General Public License* (*LGPL*)), et nous avons comparé les résultats par rapport à la version 1.13 de *SableVM*.

En raison du temps serré pour faire les tests de performance, parce que nous avons trouvé le problème de détection des interblocages faux dans les derniers mois avant la date finale de dépôt de mémoire, la recherche récursive (Voir 4.6.2) n'est pas incluse dans l'algorithme pour les tests de performance.

Ce chapitre est structuré de la façon suivante : dans la section 5.1, nous parlerons de l'environnement de test. Ensuite, dans la section 5.2, nous discuterons des programmes utilisés pour la comparaison des performances. Nous donnerons, à la section 5.2, les résultats de nos mesures, et nous discuterons des différents résultats dans la section 5.4. Finalement, dans la section 5.5, nous présenterons notre conclusion.

5.1 Environnement de test

Tous nos tests ont été faits sur un ordinateur portable *Intel*® *Core*™, 2 processeurs *T5200*, 1.60 GHz de vitesse chacun, 2 Go de RAM, mémoire cache 2 Mo, disque dur avec 5 400 RPM. Le système d'exploitation est Linux Debian/Gnu Linux, avec une version 2.6.18 du noyau. Tous les processus démons étaient arrêtés pendant les tests.

Le calcul est fait par une moyenne de l'ensemble des temps d'exécution (5 exécutions) par programme. Les temps d'exécution sont basés sur le temps (système + utilisateur) retourné par la ligne de commande `time`.

5.2 Programmes pour étude comparée (*Benchmarks*)

Nos programmes de tests sont basés sur deux grands programmes :

- *Ashes* est une collection de programmes de test Java et des programmes pour étude comparée (*Benchmarks*) développés par le groupe de recherche *Sable* de l'université de McGill.
- Nos propres programmes développés pour les tests de performance des fils d'exécution et d'interblocage.

Ashes est constitué d'un ensemble de suites, et chaque suite est constituée d'un ensemble de programmes. La plupart des programmes n'utilisent pas les fils d'exécution, ce qui nous permet de bien comparer les performances sans que notre algorithme de détection et de brisement de l'interblocage soit appelé, parce qu'il n'y a pas de fils d'exécution en concurrence. Nous avons utilisé pour les tests les suites suivantes :

- `AshesEasyTestSuite`
- `AshesHardTestSuite`
- `jikesHpjTestSuite`
- `jikesPrTestSuite`

- `kaffeRegressionSuite`

Nous avons développé nos propres programmes pour comparer les performances entre la version 1.13 de *SableVM* et la même version, en incluant notre algorithme de détection immédiate et de brisement des interblocages. Nous avons quatre programmes :

- `ConcurrentSansConflit` est un programme qui crée un nombre de fils d'exécution, et chaque fil d'exécution verrouille un ensemble d'objets sans avoir de conflit de concurrence avec un autre fil d'exécution.
- `ConcurrentAvecConflit` est un programme qui crée un nombre de fils d'exécution, et chaque fil d'exécution verrouille un ensemble d'objets avec un conflit de concurrence avec d'autres fils d'exécution.
- `SimpleInterblocage` est un programme qui crée deux fils d'exécution et qui génère un interblocage entre eux.
- `ComplexeInterblocage` est un programme qui crée plusieurs fils d'exécution et qui génère un interblocage entre eux.

Nous avons conçu nos programmes (Voir Appendice A) de façon à utiliser les fils d'exécution, le verrouillage, la libération et la concurrence entre les fils d'exécution d'une manière intense pour arriver à montrer l'efficacité de notre travail, et le fait que nos programmes sont des programmes conçus pour mesurer les performances. Il est rare de trouver des vrais programmes qui utilisent ce genre d'intensité, parce qu'il serait inutile de faire se concurrencer des centaines de fils d'exécution sur des centaines d'objets pour réaliser une logique, sauf dans des programmes pour étude comparée (*Benchmarks*).

Les temps d'exécution des suites de *Ashes* et nos propres programmes sont calculés sur une moyenne de 5 temps d'exécution. Les méthodes `main` des programmes (les suites *Ashes*) ont été modifiées pour être exécutés dans des boucles allant de quelques centaines à des millions d'itérations pour augmenter leurs temps d'exécution afin qu'il dépassent une minute.

5.3 Résultats

Nous présenterons dans cette section les résultats de nos mesures de performance (Voir les tableaux de 5.1 à 5.6). Nous commencerons par présenter les suites de *Ashes*, ensuite nous présenterons nos propres programmes.

Nos programmes sont exécutés avec les paramètres suivants :

- `ConcurrentSansConflit` : lance 350 fils d'exécution qui verrouillent 350 objets sans conflit.
- `ConcurrentAvecConflit` : lance 350 fils d'exécution qui verrouillent 350 objets avec conflit.

Programme	SableVM 1.13 (min :sec.)	Notre algorithme (min :sec.)	Différence	
			(min :sec.)	(%)
`factorial`	02 :11.089	02 :11.329	+00 :00.240	+0.18
`fahrenheit`	02 :11.681	02 :11.603	-00 :00.078	-0.06
`life`	02 :38.548	02 :38.536	-00 :00.012	-0.01
`simple12`	01 :34.817	01 :33.888	-00 :00.929	-0.99
`simple58`	01 :35.710	01 :34.729	-00 :00.981	-1.04

Tableau 5.1 Comparaison de performance : ashesEasyTestSuite

Programme	SableVM 1.13 (min :sec.)	Notre algorithme (min :sec.)	Différence	
			(min :sec.)	(%)
`fft`	01 :24.494	01 :17.363	-00 :07.131	-9.22
`probe`	02 :26.284	02 :24.899	-00 :01.385	-0.96

Tableau 5.2 Comparaison de performance : ashesHardTestSuite

5.4 Discussion

Selon les résultats obtenus dans les tableaux 5.1 à 5.5, nous remarquons que la plupart des temps d'exécution pour les suites de *Ashes* sont similaires ou proches, sauf quelques-uns. Nous savons que les mesures ne sont pas exactes à 100 %, parce que les fils d'exécution sont non déterministes (c'est-à-dire que deux ou plusieurs exécutions du même programme se déroulent différemment). L'étude de Dayong, Verbrugge and Gagnon [VEE06] a démontré que le simple fait de modifier le code de la machine virtuelle peut générer une variation allant jusqu'à 9.5% du temps d'exécution des programmes.

Programme	SableVM 1.13 (min :sec.)	Notre algorithme (min :sec.)	Différence (min :sec.)	Différence (%)
absClassMain	03 :11.309	03 :08.411	-00 :02.898	-1.54
array1	02 :54.135	02 :51.310	-00 :02.825	-1.65
array4	02 :59.978	02 :57.816	-00 :02.162	-1.22
arraymethod	01 :59.581	01 :59.259	-00 :00.321	-0.27
bigComp	01 :15.979	01 :15.926	-00 :00.053	-0.07
bigi	01 :16.194	01 :15.775	-00 :00.419	-0.55
callmm	01 :45.974	01 :45.993	+00 :00.019	+0.02
checkarray	01 :08.687	01 :08.676	-00 :00.011	-0.02
checkcast1	01 :35.581	01 :35.551	-00 :00.030	-0.03
cmplx2	01 :35.240	01 :35.044	-00 :00.195	-0.21
cnvi2b_1	01 :34.751	01 :34.530	-00 :00.221	-0.23
while1	01 :34.783	01 :34.870	+00 :00.087	+0.09
while2	01 :40.621	01 :40.568	-00 :00.052	-0.05

Tableau 5.3 Comparaison de performance : jikesHpjTestSuite

Programme	SableVM 1.13 (min :sec.)	Notre algorithme (min :sec.)	Différence (min :sec.)	Différence (%)
pr102	01 :36.262	01 :36.048	-00 :00.214	-0.22
pr172	01 :19.511	01 :19.477	-00 :00.034	-0.04
pr174	01 :27.321	01 :25.765	-00 :01.556	-1.81
pr287	01 :04.168	01 :04.214	+00 :00.046	+0.07

Tableau 5.4 Comparaison de performance : jikesPrTestSuite

Programme	SableVM 1.13 (min :sec.)	Notre algorithme (min :sec.)	Différence (min :sec.)	Différence (%)
badFloatTest	01 :10.588	01 :10.600	+00 :00.012	+0.02
charCvt	01 :09.058	01 :08.774	-00 :00.284	-0.41

Tableau 5.5 Comparaison de performance : kaffeRegressionSuite

Programme	SableVM 1.13 (min :sec.)	Notre algorithme (min :sec.)	Différence (min :sec.)	Différence (%)
ConcurrentSansConflit	01 :59.481	01 :59.523	+00 :00.042	+0.04
ConcurrentAvecConflit	01 :26.239	01 :26.411	+00 :00.172	+0.20
SimpleInterblocage	infini	0.06		
ComplexeInterblocage	infini	0.16		

Tableau 5.6 Comparaison de performance : nos propres programmes

Le plus intéressant pour notre travail est le tableau 5.6 ; nous remarquons que la version de *SableVM*, qui inclut notre algorithme, n'était plus lente que d'un temps négligeable (0.2 % et 0.04 %) par rapport aux deux programmes `ConcurrentSansConflit` et `ConcurrentAvecConflit` avec 350 fils d'exécution et 350 objets à verrouiller, et il est rare de trouver des programmes qui utilisent ce nombre important de fils d'exécution, sauf pour les serveurs Web ou des programmes semblables. Alors que pour les deux autres programmes `SimpleInterblocage` et `ComplexeInterblocage`, il était impossible de comparer entre les deux versions, parce que ces deux programmes causent de l'interblocage ; et c'est l'objectif de notre travail de le détecter et ensuite de le briser immédiatement. Pour la version 1.13 de *SableVM*, le temps d'exécution était l'infini, parce que c'est un interblocage, alors que notre version a détecté et brisé les interblocages dans un temps négligeable par rapport à l'exécution du programme lui-même que nous ne pouvons pas mesurer.

5.5 Conclusion

Dans ce chapitre, nous avons présenté les mesures de performance entre une version standard de *SableVM* et celle où nous avions implémenté notre algorithme. Nous avons remarqué dans les résultats que notre algorithme était efficace dans l'exécution des programmes Java qui n'utilisent pas les fils d'exécution, et avec un coût variant entre 0.04 % et 0.2 % pour les programmes qui utilisent les fils d'exécution et ne causent pas des interblocages. Nous avons aussi vu que notre algorithme a détecté l'interblocage et qu'il l'a brisé immédiatement dans un temps minime par rapport au temps d'exécution total du programme.

CHAPITRE VI

TRAVAUX CONNEXES

6.1 Introduction

La plupart des algorithmes qui détectent les interblocages utilisent la théorie des graphes, ce qui est classique ; ainsi, les noeuds sont les fils d'exécution et les ressources et les arêtes sont, dans la plupart des algorithmes, le verrouillage et la libération des ressources par les fils d'exécution.

Dans ce chapitre, nous allons voir quelques travaux reliés à notre travail de détection de l'interblocage. Dans la section 6.2, nous discuterons de l'algorithme *DREADLOCKS*. Dans la section 6.3, nous allons voir l'algorithme *GoodLock*, et dans la section 6.4, nous parlerons de la généralisation de l'algorithme *GoodLock*.

6.2 Algorithme *DREADLOCKS*

Maurice Herlihy et Eric Koskinen [HK08] proposent un algorithme qui se rapproche de notre algorithme dans un sens. Il se base sur le principe du graphe d'attente (Voir section Graphe ressource-allocation 3.2.2). Au lieu de mettre à jour le graphe par la relation d'attente, l'algorithme utilise le prétraitement local de fil d'exécution (*thread-local digests*). Le prétraitement local de fil d'exécution se base sur :

- La relation *owner* : $R \rightarrow T$ (fil d'exécution T détient la ressource R).
- L'ensemble de prétraitement local de fil d'exécution D_T qui représente l'ensemble des autres fils d'exécution pour lesquels T est en train d'attendre directement ou

indirectement.

- Si T n'est pas en train d'essayer d'acquérir une ressource

  D_T={T}

- Si T essaie d'acquérir une ressource R

  D_T={T} $\cup$ $D_{owner(R)}$

Le fil d'exécution T détecte l'interblocage quand il essaie d'acquérir une ressource R et $T \in D_{owner(R)}$

La difficulté dans cet algorithme est la propagation des mises à jour des prétraitements locaux des fils d'exécution pour maintenir les ensembles.

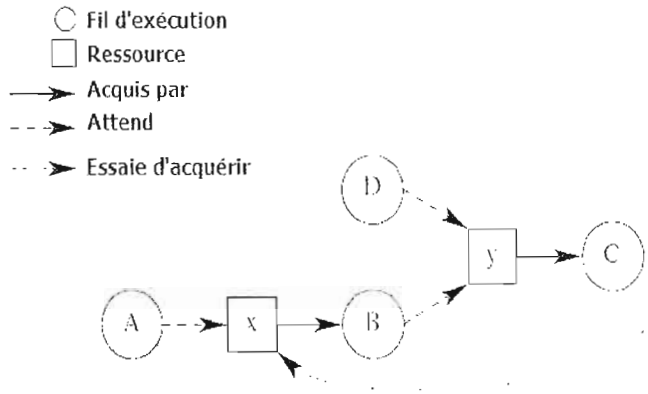


Figure 6.1 Exemple d'un graphe ressource-allocation [HK08]

Dans l'exemple de la figure 6.1 :

D_C = {C}

D_D = {C, D}

D_B = {C, B}

D_A = {C, B, A}

Si C essaie d'acquérir la ressource x qui est acquise par B, un interblocage est détecté par C parce que $C \in D_B$.

La différence avec notre algorithme est celle-ci :

- Notre algorithme se base sur la construction de l'arbre d'attente alors que l'algorithme *Dreadlocks* maintient les ensembles de prétraitements locaux des fils d'exécution.
- Notre algorithme détecte l'interblocage en cherchant le sommet de l'arbre et vérifie si le fil d'exécution courant est le sommet de l'arbre, alors que dans l'algorithme *Dreadlocks* il vérifie si le fil d'exécution courant appartient à l'ensemble de prétraitement local du fil d'exécution qui détient la ressource.

L'inconvénient de cet algorithme est la propagation des mises à jour qui coûte du temps si le nombre de fils d'exécution est important et le nombre d'opérations de verrouillage et de libération est considérable, alors que notre algorithme maintient seulement l'arbre d'attente qui est de l'ordre $O(1)$ par opération. Par contre, la détection de l'interblocage dans l'algorithme *Dreadlocks* se fait par une opération simple de vérification s'il fait partie d'un l'ensemble alors que notre algorithme fait une recherche de la racine de l'arbre d'attente qui est d'ordre $O(n)$.

6.3 Algorithme *GoodLock*

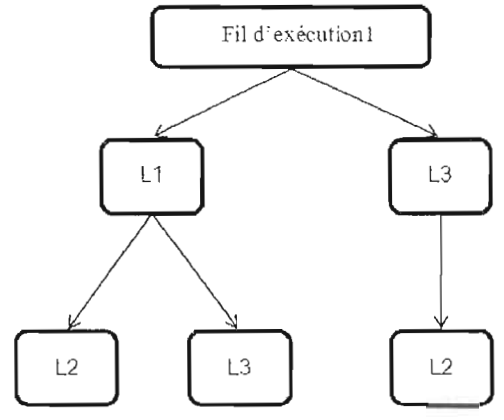
Cet algorithme proposé par Klaus Havelund [BH05] se base sur l'analyse de l'exécution d'un programme (Voir section Analyse de l'exécution 3.3.4 et modèle de vérification (*Model checking*) 3.3.3). Il est applicable sur deux fils d'exécution qui partagent deux verrous. Ces deux verrous sont acquis par les deux fils d'exécution dans un ordre inversé (Voir chapitre 3 pour en savoir plus sur les problèmes de synchronisation). Ces deux fils d'exécution n'ont pas en commun un verrou d'exclusion mutuelle *Gate Lock* (comme dans le tableau suivant, `C` est un verrou d'exclusion mutuelle pour A et B).

Sans verrou d'exclusion mutuelle		Avec verrou d'exclusion mutuelle	
fil d'exécution 1	**fil d'exécution 2**	**fil d'exécution 1**	**fil d'exécution 2**
		`lock(C) ;`	`lock(C) ;`
`lock(A) ;`	`lock(B) ;`	`lock(A) ;`	`lock(B) ;`
`lock(B) ;`	`lock(A) ;`	`lock(B) ;`	`lock(A) ;`
`unlock(B) ;`	`unlock(A) ;`	`unlock(B) ;`	`unlock(A) ;`
`unlock(A) ;`	`unlock(B) ;`	`unlock(A) ;`	`unlock(B) ;`
		`unlock(C) ;`	`unlock(C) ;`

L'algorithme se base sur la construction d'un arbre des verrous pour chaque fil d'exécution :

- Chaque noeud dans l'arbre est associé à un verrou, sauf le sommet de l'arbre, qui représente le fil d'exécution.
- Dans le verrouillage imbriqué, le fil d'exécution acquiert le verrou parent quand il acquiert le verrou enfant (comme dans l'exemple suivant L1 et L2, L1 et L3, L3 et L2).

Exécution	Arbre des verrous

`lock(L1)`
`lock(L2)`
`unlock(L2)`
`lock(L3)`
`unlock(L3)`
`lock(L2)`
`unlock(L2)`
`unlock(L1)`
`lock(L3)`
`lock(L2)`
`unlock(L2)`
`unlock(L3)`
	

Le pseudo-code de l'opération d'acquérir (`lock`) et l'opération libérer (`unlock`) :

```
lock(Thread thread , Lock lock) {
   if(fil d'exécution a déjà acquis le verrou) {
     if(verrou est un enfant du noeud courant) {
        noeud courant = ce verrou enfant;
     } else {
        ajouter le verrou comme un enfant du noeud courant;
        noeud courant = nouveau enfant;
     }
   }
}
```

```
unlock(Thread thread , Lock lock) {
        courant = parent du noeud courant;
}
```

Pour détecter les interblocages potentiels, l'algorithme compare les arbres de verrous pour chaque deux fils d'exécution :

- *nesting*(n) : l'ensemble des verrous dans le chemin du sommet du verrou n jusqu'à n.
- Pour chaque paire d'arbres (t1, t2) :
  → $\forall n_1 \in t_1$ et $\forall n_2 \in t_2$, vérifier que $n_1 \in$ *nesting*(n_2) :
    - Si $n_1 \in$ *nesting*(n_2), alors après vérification qu'il n'existe pas un verrou d'exclusion mutuelle (*Gate lock*), ajouter les deux verrous et deux fils d'exécution à la liste des interblocages potentiels.

Cet algorithme est limité aux interblocages causés par deux fils d'exécution seulement ; il ne détecte pas les interblocages potentiels causés par plus de deux fils d'exécution. Un algorithme de généralisation est nécessaire.

6.4 Algorithme *GoodLock* généralisé

Rahul Agarwal et Scott D. Stoller [AS06] ont généralisé l'algorithme de *GoodLock* pour qu'il détecte les interblocages potentiels causés par plus de deux fils d'exécution ainsi que les interblocages causés par l'utilisation des sémaphores et des variables de condition. Les arbres de verrous sont construits de la même façon que celui de *GoodLock*. Cet

algorithme construit un graphe orienté $G=(V, E)$ de sorte que :

- V contient tous les noeuds de tous les arbres.
- E contient :
  - Les arcs de chaque arbre (du parent à l'enfant) ;
  - Interarcs : un interarc est un arc bidirectionnel entre deux noeuds associés au même verrou et qui se trouvent dans des arbres différents.

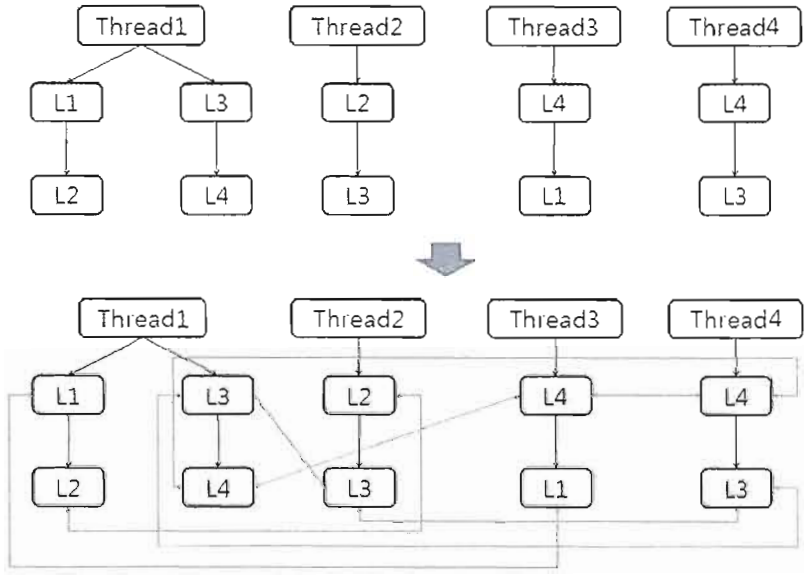


Figure 6.2 Exemple d'un graphe orienté de l'algorithme *GoodLock* généralisé [HAWS08]

Nous appelons :

- Pour un graphe G, un chemin valide est un chemin qui ne contient pas d'interarcs consécutifs, et où les noeuds de chaque arbre de verrouillage apparaîtraient tout au plus comme une sous-séquence consécutive dans le chemin.
- Un cycle valide est un cycle qui ne contient pas d'interarcs consécutifs et où les noeuds de chaque fil d'exécution apparaîtraient tout au plus comme une subséquence consécutive dans le cycle.

Exemple

$L3(thread1) \rightarrow L3(thread2) \rightarrow L3(thread4) \rightarrow L3(thread1)$ est un chemin invalide. $L3(thread1) \rightarrow L4(thread1) \rightarrow L4(thread4) \rightarrow L3(thread4) \rightarrow L3(thread1)$ est un chemin valide et un cycle valide.

Chaque cycle valide représente un interblocage potentiel. Pour détecter les interblocages potentiels, nous passons par deux étapes :

- Il faut traverser tous les chemins valides pour trouver les cycles valides.
- Chaque cycle valide est un interblocage potentiel, après vérification qu'il n'existe pas un verrou d'exclusion mutuelle (*Gate lock*).

L'inconvénient de cet algorithme est la complexité de mise à jour des Interarcs et le temps nécessaire pour traverser tous les chemins valides pour trouver les cycles valides qui est d'un ordre important.

6.5 Conclusion

Dans ce chapitre, nous avons présenté quelques travaux reliés à la détection des interblocages. Nous avons vu l'algorithme *Dreadlocks* et nous avons présenté aussi l'algorithme *GoodLock* et sa généralisation.

CONCLUSION

Dans ce mémoire, nous avons présenté notre algorithme de détection immédiate et de brisement de l'interblocage. L'objectif de notre travail était de détecter efficacement les interblocages causés par les fils d'exécution en concurrence.

Dans ce mémoire, nous avons parlé de la synchronisation dans les différentes étapes d'un programme Java, en commençant par le code écrit en Java, ensuite le code-octet. Nous avons vu aussi les problèmes causés par la synchronisation, et principalement le problème d'interblocage et les conditions nécessaires à son existence et après nous avons aussi parlé des méthodes de détection existantes. Nous avons vu la conception d'un algorithme efficace de détection immédiate des interblocages et nous avons vu la preuve de rectitude et l'analyse de la complexité de notre algorithme. Nous avons aussi vu le brisement de l'interblocage par le soulèvement d'une exception.

Nous avons expérimenté notre algorithme sur la machine virtuelle *SableVM*, en utilisant la suite de mesures (*benchmark*) *Ashes* et nos propres programmes de mesure de performance. Les résultats de cette expérimentation ont donné une comparaison de notre algorithme par rapport à la version 1.13 de *SableVM*. Ces résultats ont montré que notre détection immédiate a un coût nul dans une majorité de cas, et coûte une surcharge de temps d'exécution de 0,04 % à 0,2 % pour des programmes qui utilisent 350 fils d'exécution, où chaque fil d'exécution verrouille et libère 350 objets avec ou sans concurrence respectivement avec d'autres fils d'exécution.

APPENDICE A

Voici les programmes utilisés dans les tests de performance dans le chapitre 5.

A.1 ConcurrentSansConflit.java

```
class ConcurrentSansConflit extends Thread {
    private static int n_threads = 350;
    private static int n_locks = 350;
    private static Object[][] locks;
    private static ConcurrentSansConflit[] threads;
    private int id;

    private void SynchronisationSansConflit(int start_lock) {
        if (start_lock == 0) return;
        synchronized (locks[this.id][start_lock]) {
            SynchronisationSansConflit(start_lock-1);
        }
    }

    public void run() {
        SynchronisationSansConflit(n_locks-1);
    }

    public static void main(String[] args) {
        if (args.length == 2){
            n_threads=Integer.parseInt(args[0]);
            n_locks = Integer.parseInt(args[1]);
        } else if(args.length == 1) {
            n_threads = Integer.parseInt(args[0]);
        }

        threads = new ConcurrentSansConflit[n_threads];
        locks = new Object[n_threads][n_locks];
        for (int i = 0; i < n_threads; i++){
            threads[i] = new ConcurrentSansConflit();
            threads[i].id = i;
            for (int j = 0; j<n_locks; j++) {
                locks[i][j] = new Object();
            }
        }

        // Démarrer les fils d'exécution
```

```
        for (int i = 0; i < n_threads; i++) {
            threads[i].start();
        }
    }
}
```

A.2 ConcurrentAvecConflit.java

```
class ConcurrentAvecConflit extends Thread {
    private static int n_threads = 350;
    private static int n_locks = 350;
    private static Object[] locks;
    private static ConcurrentAvecConflit[] threads;
    private int id;

    private void SynchronisationAvecConflit(int start_lock) {
        if(start_lock < 0) return;
        synchronized (locks[start_lock]) {
            SynchronisationAvecConflit(start_lock-1);
        }
    }

    public void run() {
        SynchronisationAvecConflit(n_locks-1);
    }

    public static void main(String[] args) {
        if (args.length == 2) {
            n_threads = Integer.parseInt(args[0]);
            n_locks = Integer.parseInt(args[1]);
        } else if(args.length == 1) {
            n_threads = Integer.parseInt(args[0]);
        }

        threads = new ConcurrentAvecConflit[n_threads];
        locks = new Object[n_locks];
        for (int i=0; i < n_threads; i++) {
            threads[i] = new ConcurrentAvecConflit();
            threads[i].id = i;
            locks[i] = new Object();
        }

        // Démarrer les fils d'exécution
        for (int i = 0; i < n_threads; i++) {
            threads[i].start();
        }
    }
```

```
}
```

A.3 SimpleInterblocage.java

```
public class SimpleInterblocage {

    public static void main(String[] args) {
        final Object resource1 = "resource1";
        final Object resource2 = "resource2";
        Thread t1 = new Thread() {
            public void run() {
                synchronized(resource1) {
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) { }

                    synchronized(resource2) {}
                }
            }
        };

        Thread t2 = new Thread() {
            public void run() {
                synchronized(resource2) {
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) { }
                    synchronized(resource1) { }
                }
            }
        };

        t1.start();
        t2.start();
    }
}
```

A.4 ComplexeInterblocage.java

```
class ComplexeInterblocage {
    public static int MaxThreads = 110;
    public static int MaxLocks = 110;
    public static int Wait = 99;
    public static Object[] Lock;
    public static MyThread[] t;
```

```
public static int num_thread_started = 0;

public static void CallRecursive(int thread_index, int index) {
    if (index >= 1) {
        synchronized (Lock[index]) {
            CallRecursive(thread_index,index-1)
        }
    }
}

public static void CallRecursiveBack(int thread_index, int index) {
    if (index <= MaxLocks) {
        synchronized (Lock[index]) {
            CallRecursiveBack(thread_index, index+1);
        }
    }
}

public void RunDeadlock() {
    Lock=new Object[MaxLocks + 1];
    for (int i = 1; i <= MaxLocks; i++) {
        Lock[i] = new Object();
    }

    t = new MyThread[MaxThreads + 1];
    for (int i = 1; i <= MaxThreads; i++) {
        t[i] = new MyThread();
        t[i].setIndex(i);
    }

    for(int i = 1; i <= MaxThreads; i++) {
        t[i].start();
    }
}

public static void main(String[] args) {
    ComplexeInterblocage cd = new ComplexeInterblocage();
    cd.RunDeadlock();
}

public class MyThread extends Thread {
    public int index;

    public void setIndex(int i) {
        this.index = i;
    }

    public synchronized void ON() {
        num_thread_started++;
    }
```

```
        public synchronized void  OFF() {
            num_thread_started--;
        }

        public void run() {
            ON();
            if (this.index % 2 ==0) {
                try {
                    CallRecursive(this.index, MaxLocks);
                } catch(Exception e){
                    OFF();
                    return;
                }
            } else {
                try {
                    CallRecursiveBack(this.index, 1);
                } catch(Exception e) {
                    OFF();
                    return;
                }
            }

            OFF();
        }
    }
}
```

BIBLIOGRAPHIE

[AS06] Rahul Agarwal et Scott D. Stoller, 2006 « RunTime Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables », *In International Symposium on Software Testing and Analysis, Proceedings of the 2006 workshop on Parallel and distributed systems : testing and debugging*, Pages : 51-60. ACM Press.

[AT01] Andrew S. Tanenbaum, 2001 « Modern Operating Systems (2nd Edition) », *Édition Prentice Hall.* ISBN 0-130-31358-0.

[BD97] David R. Butenhof, 1997 « Programming with POSIX Threads », *Édition Addison-Wesley.* ISBN 0-201-63392-2.

[BH05] Saddek Bensalem et Klaus Havelund », 2005 « Scalable Dynamic Deadlock Analysis of MultiThreaded Programs », *In Parallel and Distributed Systems : Testing and Debugging (PADTAD - 3), IBM Verification Conference*

[BJ03] Jérome Bougeault, 2003 « Java, La maitrise », *Édition Eyrolles.* ISBN 2-212-11352-8.

[BKMS98] David F. Bacon et Ravi Konuru et Chet Murthy et Mauricio Serrano, 1998 « Thin Locks : Featherweight Synchronization for Java », *Conference on Programming Language Design and Implementation, Proceedings of the ACM SIGPLAN 1998 conference.* Pages : 258-268. ACM Press.

[CES71] E. G. Coffman, Jr. et M. J. Elphick et A. Shoshani, 1971 « System Deadlocks », *ACM Computing Surveys (CSUR), Vol. 3, No. 2.* Pages : 67-78. ACM Press.

[DI02] Ian F.Darwin, 2002 « Java en action », *Édition O'REILLY.* ISBN 2-84177-203-9.

[EA03] Dawson Engler et Ken Ashcraft, 2003 « RacerX : Effective, Static Detection of Race Conditions and Deadlocks », *In Operating Systems Principles, ACM SIGOPS Operating Systems Review(SOSP'03), Vol. 37, No. 5.* Pages : 237-252. ACM Press.

[FJ06] Jose E. Fadul, 2006 « Toward the Static Detection of Deadlock in Java Software », *Mémoire de maîtrise, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Air University*

[GE02] Etienne Gagnon, 2002 « A Portable Research Framework for the Execution of Java Bytecode », *School of Computer Science, McGill University, Montreal*

[HAWS08] Hong,Shin, 2008 « Potential Deadlock Detection By K. Havelund, R. Agarwal, L. Wang, S. D. Stoller »,

[HC04] Cay S.Hortsman et Gary Cornell, 2004-2005 « Au coeur de Java 2 (Volume 2) », *Édition CampusPress.* ISBN 2-7440-1833-3

[HK00] Klaus Havelund, 2000. « Using runtime analysis to guide model checking of java programs », *In SPIN Model Checking and Software Verification, Proceedings of the 7th International SPIN Workshop, volume 1885 of Lecture Notes In Computer Science*, Pages : 245-264. Springer

[HK08] Maurice Herlihy et Eric Koskinen, 2008 « DREADLOCKS : Efficient Deadlock Detection for STM », *In Parallel Algorithms and Architectures, Proceedings of the twentieth annual symposium.* Pages : 297-303. ACM Press.

[JC97] Java Code Conventions, 1997 « Sun Microsystems, Inc. », *Sun Microsystems, Inc.*

[LD00] Doug Lea, 2000 « Concurrent Programming in Java, Design Principles and Patterns », *Édition Addison-Wesley, second edition.* ISBN 0-201-31009-0

[LY99] Tim Lindholm et Frank Yellin, 1999 « The Java Virtual Machine Specification », *Édition Addison-Wesley, second edition.* ISBN 0-20-143294-3.

[NP96] Patrick Niemeyer et Joshua Peck, 1996 « Java par la pratique », *Édition O'REILLY.* ISBN 2-84177-022-2.

[SH00] Scott Oaks et Henry Wong, 2000 « JAVA Threads », *Édition O'REILLY, second edition.* ISBN 2-84177-079-6.

[VB98] Bill Venners, 1998 « Inside the Java virtual machine », *Édition McGraw-Hill.* ISBN 0-07-913248-0.

[VEE06] Dayong Gu et Clark Verbrugge et Etienne M. Gagnon, 2006 « Relative Factors in Performance Analysis of Java Virtual Machines », *VEE '06 : Proceedings of the 2nd international conference on Virtual execution environments.* Pages : 111-121. ACM Press.