Maîtrise d'Informatique
option Micro-Informatique Micro-Electronique

Réalisation d'une machine virtuelle Java sous le système d'exploitation Plan9

Laurent Bodet

Sous la direction de : Jean Méhat

Juillet 1997

Université Paris 8 - Département MIME

Remerciements

Je tiens à remercier tout particulièrement mon directeur de maîtrise Jean Méhat qui a toujours su se rendre disponible et attentif durant toute la réalisation de ce projet.

Je remercie chaleureusement les étudiants et anciens étudiants du département Informatique avec qui, par de nombreux échanges d'idées, de points de vue et d'encouragements, m'ont aidé à mener à bien ce projet.

Résumé

Nous présentons ici une implémentation nouvelle de la machine virtuelle Java que nous avons réalisée sous le système d'exploitation Plan 9. Cette implémentation représente environ 7000 lignes de C, plus 2000 lignes pour les librairies qui contiennent les méthodes natives qui assurent l'interface entre Java et le système.

Notre machine virtuelle s'articule autour d'un interpréteur de *byte code* Java qui décode et exécute les instructions machine générées par un compilateur. De par la structure orientée objet du langage Java, notre machine développe des mécanismes d'éditions de liens, de résolution dynamique à l'exécution et de gestion de la mémoire très particuliers.

Les erreurs des programmes Java qui s'exécutent dans notre machine sont pris en charge par un mécanisme de lancement et d'interception d'exceptions.

Le parallélisme des programmes Java est implémenté par un ordonnanceur de *threads*. Les primitives de synchronisation des *threads* sont réalisées à l'aide d'un moniteur.

Notre interprète supporte de gros programmes écrits en Java, notamment le compilateur Java de Sun lui-même que nous utilisons pour produire du *byte code* exécutable à partir du code source.

La vitesse d'exécution de notre machine est supérieure à celle de l'interprète de Sun pour les programmes de taille modeste ; pour les programmes de plus grande ampleur, il est sérieusement handicapé par l'absence, pour le moment, de mécanisme de récupération de la mémoire.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Nous présentons ici notre implémentation d'une machine virtuelle Java [1] pour le système d'exploitation Plan 9. Nous avons conçu et réalisé une machine logicielle qui est en mesure d'exécuter du code Java. Elle est compatible avec les spécifications du langage définies par Sun.

1.1 Motivations

Notre principale motivation était de travailler avec le système d'exploitation Plan 9. Ce système présente de nouveaux concepts et de nouveaux outils que nous voulions découvrir. A nos yeux, le développement d'applicatifs sur ce système est certainement le meilleur moyen d'y arriver. Il faut aussi ajouter que la jeune communauté des utilisateurs de Plan 9 encourage vivement le développement pour permettre de faire évoluer le système. Nous avons donc travaillé dans ce sens.

Le langage Java est devenu en peu de temps très populaire dans le monde de l'Internet. Poussé par Sun, les applications Java fleurissent sur le World Wide Web, parce qu'elles peuvent être téléchargées sur les navigateurs. Il était donc pour nous intéressant de se pencher d'un peu plus près sur ce nouveau langage.

Sun fournit par défaut des packages de développement Java[2] pour un nombre restreint d'architectures et il n'existe pas encore de tels outils destinés au système Plan 9. Beaucoup de personnes à travers le monde ont

[1] Notre machine est virtuelle parce qu'elle simule le comportement d'une machine hypothétique de manière logicielle. Java est un langage de programmation orienté objet conçu par Sun Microsystem. Java est une marque déposée.

[2] Le package standard de développement Java fournit par Sun s'appelle JDK pour *Java Developpement Kit*. Le JDK de Sun contient principalement le compilateur Java et la machine virtuelle Java.

décidé de porter le JDK de Sun sur divers systèmes et diverses plateformes[3]. Nous avons décidé de ne pas suivre la même voie et nous avons implémenté notre propre machine virtuelle.

Le concept de machine virtuelle choisi par Sun pour exécuter du code Java nous a intéressé. Les composantes de la machine virtuelle définissent un environnement complet d'exécution comme on le retrouve sur un ordinateur moderne.

1.2 La Machine virtuelle Java

La machine virtuelle Java est la plateforme pour laquelle tous les programmes Java sont compilés[4]. Cette machine est définie par un ensemble de composantes articulées autour d'un interpréteur.

La machine virtuelle Java est caractérisée par un jeu complet d'environ 160 instructions qui ressemblent aux instructions des microprocesseurs actuels. Elles sont codées par un octet de code opératoire et suivi par un ou plusieurs arguments ; on appelle le code machine Java du *byte code* Java. Les instructions peuvent être organisées en trois grands groupes :

- Les instructions qui opèrent sur les données,
- les instructions arithmétiques et de conversions de type,
- et les instructions qui contrôlent le flux d'exécution.

Ce sont les instructions machine Java qui sont interprétées dans la machine virtuelle.

Afin de contrôler l'exécution du *byte code*, la machine virtuelle possède un jeu de registres qui sont contenus en partie dans une *frame* d'exécution. Les *frames* sont des structures de données sur lesquelles la machine virtuelle exécute une méthode Java. Les différents registres sont :

- Un pointeur de programme qui indique à tout moment à la machine virtuelle quelle instruction elle doit exécuter.
- Une pile où les opérandes des instructions sont sauvegardées [5]. Cette structure de données est représentée par deux registres : le premier

[3] Les sources du JDK sont disponibles sous licence auprès de Sun. Le portage du JDK, qui est écrit en C ANSI, consiste à faire fonctionner la machine virtuelle de Sun sur la nouvelle architecture. Le compilateur écrit entièrement en Java aucun travail de portage n'est nécessaire.

[4] Ce mécanisme permet de rendre le code exécutable Java indépendant de toute architecture matérielle et système. Pour exécuter une application, il faut simplement disposer d'une machine virtuelle Java sur la plateforme où doit s'exécuter la machine virtuelle.

[5] Comme les opérandes sont placées sur la pile avant l'instruction, et qu'on y trouve le résultat de l'opération, la machine virtuelle est une machine à pile.

indique l'adresse du début de la pile et le second indiquant le sommet de la pile.
- Un registre qui contient l'adresse de la première variable locale de la *frame*.
- Un registre qui permet de connaître la *frame* courante. Ce registre n'est pas contenu dans la *frame*, il est global.

La gestion de la mémoire allouée par les objets Java dans la machine virtuelle est réalisée de façon automatique. Les objets alloués par une application sont pris en charge par la machine. Ainsi lorsque les objets ne sont plus utilisés par l'application, la machine virtuelle détruit ces objets et libère l'espace mémoire qu'ils occupaient.

1.3 Notre machine Virtuelle Java

La machine virtuelle Java que nous avons réalisée est capable d'exécuter des applications Java sur le système d'exploitation Plan 9. Elle est compatible avec les spécifications du langage Java fournies par Sun.

1.3.1 Lecture et représentation des fichiers compilés Java

Notre machine virtuelle peut lire et analyser les fichiers générés par le compilateur Java. Ces fichiers représentent la définition d'une et une seule classe Java. Ils contiennent une table des symboles, une table contenant la définition des variables de la classe et la définition des méthodes déclarées par la classe.

Les classes sont chargées dans la machine virtuelle par un chargeur de classe système. Son but est de retrouver, sur le disque local où s'exécute la machine virtuelle, le fichier qui représente une classe donnée et de construire en mémoire une représentation interne de cette classe.

Nous avons choisi de représenter les classes Java en utilisant la même structure de données qui est utilisée dans les spécifications Java. Toutes les classes chargées sont représentées avec la même structure de données et une classe ne possède qu'une seule représentation.

1.3.2 Le cœur de notre machine virtuelle

Notre machine virtuelle est basée sur un interpréteur de code machine Java. L'interpréteur pour exécuter du code Java utilise deux mécanismes importants qui sont l'édition de liens et la résolution dynamique.

L'interpréteur

Une *frame* représente dans notre machine virtuelle une structure de données qui correspond à l'environnement d'exécution des méthodes Java. Lorsque notre machine désire exécuter une méthode définie dans une classe Java, elle construit une *frame*. Cette *frame* se compose d'un pointeur de programme qui contient l'adresse de la prochaine instruction à exécuter, d'un espace mémoire pour stocker les variables locales de la méthode et d'une pile contenant les opérandes des instructions machine.

Notre interpréteur est une simple fonction C qui décode et exécute les instructions ; c'est à cet endroit que le code Java devient le code machine. Il possède une entrée par instruction définie dans les spécifications de Sun.

Pour réaliser l'appel de méthodes, notre machine virtuelle gère une pile de *frames*. Lors de l'appel d'une méthode, notre machine crée une *frame* et l'empile sur le sommet de la pile de *frames*. Lorsque que la méthode finit son exécution, la machine virtuelle dépile sa *frame* et poursuit son exécution sur la méthode qui reste sur le sommet de la pile de *frames*.

L'édition de liens

L'édition de liens est le mécanisme qui va permettre à une classe nouvellement chargée de transiter vers un état où elle pourra être utilisée par l'environnement d'exécution de notre machine virtuelle.

Ce mécanisme commence par la préparation de la classe. Il s'agit ici de construire les structures de données qui sont utilisées par l'interpréteur. Vient ensuite l'initialisation de la classe. Cette étape consiste à exécuter l'initialisateur statique d'une classe. Il initialise simplement les variables statiques d'une classe [6].

Notons que nous avons du réaliser les différentes étapes de l'édition de liens de manière récursive. Une classe ne peut être préparée et initialisée que si sa classe parente est préparée et initialisée.

La résolution dynamique

Les instructions machine qui composent les méthodes d'une classe Java font en permanence référence à la table des symboles. Le processus de résolution consiste à déterminer les valeurs concrètes des valeurs symboliques contenues dans la table des symboles. Il permet à l'interpréteur de retrouver les références réelles sur les données qu'il sait manipuler.

[6] Les variables statiques sont partagées par toutes les instances d'une même classe. Nous pouvons les considérer comme les variables globales du langage C, auquel s'ajoute la protection fournie par l'accès aux classes.

Les objets Java

Toutes les données manipulées par le langage sont des objets. Un objet représente généralement une instance d'une classe particulière.

Notre machine virtuelle représente un objet Java par une structure de données qui contient principalement la classe qui définit l'objet et ses variables d'instances. La hiérarchie de la classe qui définit l'objet est parcourue récursivement pour déterminer toutes les variables dont hérite l'objet de ses classes parentes.

1.3.3 La gestion des exceptions

Le mécanisme d'exception introduit dans le langage Java permet de gérer les erreurs et les événements exceptionnels qui peuvent intervenir durant l'exécution des programmes.

Deux types d'exceptions peuvent être lancées dans notre machine virtuelle.

1. Les exceptions lancées par la machine virtuelle regroupent toutes les erreurs qui peuvent être générées lors des phases de chargement, d'édition de liens, de résolution ou lorsque les ressources du système sont épuisées.
2. Les exceptions lancées par le développeur Java. Ce sont les exceptions souvent redéfinies qui sont lancées par le mot clé `throw` du langage.

Nous avons réalisé la gestion des exceptions dans notre machine virtuelle en implémentant l'instruction machine `athrow` pour gérer les deux types d'exceptions.

1.3.4 Les threads Java

Les concepteurs du langage Java ont introduit la création et la manipulation des *threads* aussi bien dans le langage que dans ses librairies. La création des *threads* se réalise par la création d'un objet de type `Thread`. Ils sont contrôlés par des méthodes définies dans cette classe. Toutes les méthodes de contrôles agissent sur l'exécutions des *threads*.

Pour ordonnancer les différents *threads* d'une application Java, nous avons créé notre propre ordonnanceur. Il travail sur des listes qui représentent les états dans lesquels les *threads* se trouvent.

Les mécanismes de synchronisation des *threads* sont basés sur l'utilisation d'un moniteur.

1.3.5 Le portage des librairies Java

Le langage Java est fourni par défaut avec une librairie de base. Cette librairie est écrite principalement en Java. Cependant, il existe des fonctions dites *natives* qui ne peuvent pas être écrites en Java, parce qu'elles font en général appel à des fonctionnalités du système d'exploitation.

Nos fonctions *natives* sont écrites en C et *linkées* à notre machine virtuelle. Elles représentent environ 2000 lignes de code C.

1.4 Plan de lecture

Dans le chapitre 2 nous commençons par donner un aperçu du langage Java en le replaçant dans son environnement d'exécution et en décrivant ses principales caractéristiques. Nous poursuivons par la description de la machine virtuelle des spécifications de Sun.

Nous présentons succinctement dans le chapitre 3 les principaux concepts mis en œuvre dans le système d'exploitation Plan 9. Nous présentons aussi le réseau Plan 9 sur lequel notre machine virtuelle à été développé. Enfin nous passons en revue les différents outils qui nous ont permis de développer sur ce système.

Le chapitre 4 décrit dans un premier temps le format des fichiers générés par le compilateur Java : les exécutables Java. Nous détaillons ensuite la représentation interne des fichiers Java et le manière dont ils sont chargés dans notre machine virtuelle. Nous décrivons aussi la lecture des fichiers Java à partir des archives *zippées*. Enfin, nous terminons ce chapitre par la présentation d'un désassembleur de fichiers compilés que nous avons réalisé.

Le chapitre 5 décrit les principaux mécanismes que nous avons implémentés dans notre machine virtuelle. Nous commençons par décrire son environnement d'exécution et l'implémentation de notre interpréteur. Nous continuons par la description des mécanismes d'édition de liens, d'initialisation et de résolution dont notre machine est dotée. Nous consacrons aussi dans ce chapitre une section à la manière dont les objets Java sont créés et manipulés par notre machine virtuelle. Nous poursuivons en montrant les différentes étapes qui sont réalisées par notre machine avant de commencer l'exécution d'une application et terminons pour synthétiser par la description de l'implémentation de quelques instructions machine Java.

Le chapitre 6 est entièrement consacré à l'implémentation du mécanisme de gestion des exceptions dans notre machine virtuelle. Nous commençons par décrire le concept des exceptions dans le langage Java. Nous abordons ensuite la manière dont nous lançons et interceptons les exceptions. Enfin, nous décrivons l'implémentation des méthodes natives associées aux exceptions.

Le chapitre 7 est dédié à l'implémentation des threads dans notre machine virtuelle. Nous commençons par décrire la manière dont Java introduit les threads dans le langage, ceci nous permet de mettre en évidence tous les mécanismes dont doit être doté notre machine. Nous analysons ensuite les différentes approches d'implémentation possibles des threads et justifions notre choix. Nous continuons par décrire le fonctionnement de notre ordonnanceur et décrivons tous les mécanismes de synchronisation des threads que nous avons implémentés. Nous terminons par un résumé qui done un diagramme de transitions d'états des threads et critiquons notre réalisation.

Le chapitre 8 montre la manière dont nous avons porté les librairies Java. Nous décrivons dans un premier temps comment est réalisé l'appel des méthodes natives dans notre machine virtuelle. Nous passons ensuite en revue les différentes classes des *packages* de la librairie Java qui ont nécessités un portage.

Nous commençons le chapitre 9 par décrire quelques applications que nous pouvons faire tourner avec notre machine virtuelle. Nous continuons par la description d'une série de test que nous avons effectué pour comparer la vitesse d'exécution de notre machine à celle développé par Sun.

Chapitre 2

Le langage Java et la machine virtuelle

Dans ce chapitre, nous abordons la description du langage Java et de la machine virtuelle qui lui est associée, pour lesquels nous avons développé un interprète décrit dans les chapitres 4 à 8.

Après un bref historique, nous replaçons le langage Java dans son environnement d'exécution et rappelons les principales caractéristiques du langage. Nous décrivons ensuite l'architecture de la machine virtuelle telle qu'elle est décrite par les spécifications de Sun.

2.1 Un survol de Java

Le langage Java fut conçu pour le développement de logiciels embarqués sur des machines électroniques grand public : des appareils tels que les grille-pain, les fours micro-ondes ou des cafetières électriques. Les logiciels embarqués sur de telles machines doivent respecter des contraintes bien particulières. Ils s'exécutent sur des puces électroniques qui évoluent très rapidement et pour des raisons de coût elles sont changées fréquemment par les constructeurs. Les logiciels doivent donc être capables de tourner sur ces nouvelles puces dès leur apparition sur le marché.

Lorsqu'une équipe de chez Sun dirigée par James Gosling travaille sur ce type de logiciels, ils s'apperçoivent rapidement que les langages existant comme C et C++ ne sont pas adaptés. En effet, les programmes écrit dans ces langages doivent être compilés pour une puce particulière. Lorsqu'une nouvelle puce sort, les programmes doivent être recompilés.

Gosling se lança donc en 1990 dans la conception d'un nouveau langage de programmation plus adapté à de tels logiciels. Ce langage, qui était connu sous le nom de Oak, était petit, fiable et indépendant de l'architecture. En

1993, alors de l'équipe de Sun continuait à travailler à la conception du langage Java, le World Wide Web commençait à s'imposer sur l'internet. L'équipe de Gosling réalisa alors qu'un langage comme Java, indépendant de toute architecture, serait idéal pour programmer sur Internet. Le développement de Java pris alors une nouvelle importance pour Sun.

Java est donc un langage de programmation orienté objet et conçu particulièrement pour évoluer sur l'internet . On distingue deux types particuliers de programmes Java.

Le premier, et certainement le plus répandu à l'heure actuelle, est le programme s'exécutant au sein des pages Web par l'intermédiaire d'un navigateur. Ce type de programme permet d'embellir et de rendre plus interactive les pages stockées sur les serveurs Web de l'internet. Ils utilisent les librairies graphiques fournies par Java et sont appelés des *Applets*.

Le second type de programme, que nous appellerons des *applications* n'utilisent pas de navigateur pour s'exécuter. Les *applications* sont lancées simplement sur la ligne de commande comme un programme écrit et compilé dans un autre langage. Il est nécessaire dans ce type d'application de posséder une machine virtuelle sur laquelle l'application s'exécute.

Remarquons tout de même que dans les deux types de programmes Java, le code compilé s'exécute sur une machine virtuelle. Les *Applets* qui s'exécutent dans un navigateur sont interprétées par une machine virtuelle. Dans ce cas la machine virtuelle est implémentée d'une manière un peu particulière dans le navigateur.

2.1.1 L'environnement d'exécution

Tout programme Java voulant être exécuté doit dans un premier temps passer entre les mains du compilateur Java. Le compilateur produit un fichier compilé pour la machine virtuelle.

L'exécution du code compilé incombe à l'interpréteur de la machine virtuelle. Le travail peut se diviser en trois étapes : le chargement de la classe, la vérification et l'exécution (voir figure 2.1).

Le chargement recherche le fichier compilé Java de la classe à exécuter et le charge en mémoire principale. Il charge aussi toutes les classes parentes.

Après que cette étape soit réalisée, tout le code est passé au vérificateur de code. Le vérificateur s'assure que la classe est bien formée et que la syntaxe et la sémantique des instructions sont vérifiées. Il doit aussi vérifier que les instructions qui implémentent les méthodes des classes ne violent pas l'intégrité du système.

Enfin le code est interprété. C'est à cet endroit que le code compilé Java ou *byte code* devient le code machine.

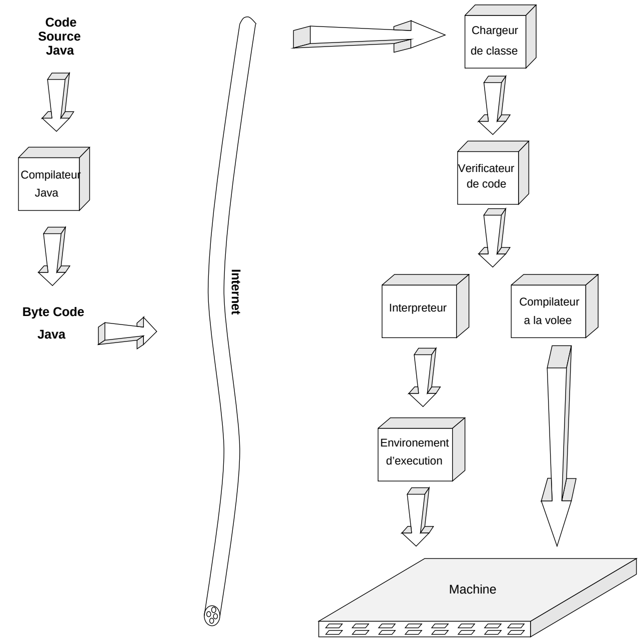

FIG. 2.1: L'environnement d'exécution Java

2.1.2 Principales caractéristiques de Java

Nous décrivons dans cette section les principales caractéristiques du langage Java décrites par les concepteurs dans [Gosling and McGilton, 1995]. Bien que la description qui en est faite soit racoleuse et purement publicitaire, nous nous efforçons de n'en retenir que les points importants.

Simple

Lors de la conception du langage, l'un des objectifs des concepteurs était de créer un langage assimilable rapidement par les programmeurs. Le langage ne contient donc qu'un nombre limité de constructions. Un autre objectif était de rendre familier le langage à une majorité de programmeurs afin de rendre plus rapide la migration. Le langage Java est donc basé sur le C et le C++.

Pour obtenir un langage simple les concepteurs ont supprimé beaucoup de fonctionnalités disponibles en C et C++ [Sun Microsystems, 1996]. Les fonctionnalités supprimées sont principalement celles qui encouragent des pratiques douteuses ou celles qui sont très peu utilisées. On ne retrouve pas d'instruction `goto` par exemple. Java n'utilise pas de fichiers d'en-tête et le préprocesseur a été éliminé. Le langage étant orienté objet, il ne possède ni structures ni unions. Java n'autorise pas non plus la surcharge d'opérateurs et l'héritage multiple.

La simplification la plus importante est sans aucun doute la suppression des pointeurs. Les pointeurs sont considérés comme étant la plus grande source de bogues dans les programmes C et C++.

Orienté objet

Java est orienté objet. Cela signifie que la conception de programme Java doit être pensée en terme de classes qui englobent des données et des méthodes qui les manipulent ; l'objet étant une instance d'une classe. Les données et les méthodes définissent respectivement l'état et le comportement d'un objet.

Les classes sont organisées en hiérarchie, dans laquelle une sous-classe hérite des propriétés et du comportement de sa classe parente. Une hiérarchie de classe à une classe racine dont le comportement est très général.

Interprété

Le compilateur Java génère du *byte code* et non du code machine. Pour exécuter du code Java, nous devons utiliser un interpréteur qui exécute le *byte code*. Java est donc en ce sens un langage interprété. Le *byte code* Java permet d'obtenir un format d'objet indépendant de toute architecture.

L'interpréteur et l'environnement dans lequel s'exécute une application Java sont appelés une machine virtuelle Java. Ce sont les différentes implémentations de la machine virtuelle sur différentes architectures qui rendent possible l'exécution d'applications Java.

Sécurisé

Java étant destiné à être utilisé dans un environnement distribué, un effort tout particulier a été réalisé par les concepteurs pour sécuriser l'exécution des applications. Pour fournir un environnement sûr, plusieurs mécanismes de sécurité sont mis en place lors de l'exécution d'application Java [Sun, 1996].

- Le langage Java étant fortement typé, le compilateur effectue un maximum de vérifications pour éviter toute erreur qui risquerait de compromettre l'intégrité du système.
- Le vérificateur de code intégré à la machine virtuelle tente de vérifier que le code qui vient d'être chargé dans la machine virtuelle respecte bien la syntaxe et la sémantique du *byte code* Java. A ce niveau la

machine virtuelle se protège contre le code généré par un compilateur ayant de mauvaises intentions.

- Les chargeurs de classes de la machine virtuelle peuvent être modifiés pour augmenter la sécurité des classes chargées. Les classes chargées par un chargeur de classe donné ne sont accessibles que par des classes qui ont été chargées par le même chargeur de classe.
- Une interface spécifique est créée pour l'utilisation des *applets* qui empêche ses dernières de manipuler les ressources locales du système où la machine virtuelle s'exécute.

Multi-Threads

Le langage Java supporte la création et la manipulation des *threads*. Les *threads* sont de simples flux d'exécution qui partagent un même espace d'adressage dans la machine virtuelle Java.

2.2 Architecture de la machine virtuelle

La machine virtuelle Java est définie par un ensemble d'instructions et d'unités fonctionnelles implémentées entièrement de manière logicielle. Elle est définie par les quatres spécifications suivantes, qui contrôlent l'implémentation et l'interprétation du code Java :

- le jeu d'instructions,
- le jeu de registres,
- la pile,
- l'allocation de mémoire.

2.2.1 Le jeu d'instructions

Le jeu d'instructions pour la machine virtuelle est équivalent à un jeu d'instructions d'un microprocesseur. Chaque instruction Java est constituée d'un code d'instruction (*opcode*) et suivie par zéro ou plusieurs opérandes.

On retrouve des instructions pour charger des entiers de la mémoire (*iload*), pour créer des tableaux (*anewarray*), des opérateurs logiques (*iand*, ET logique entre deux entiers). Chaque instruction est codée sur 8 bits, suivie par un nombre variable d'opérandes. Les opérandes fournissent les données nécessaires à l'exécution de l'instruction comme l'adresse à laquelle sauter ou la valeur à utiliser dans un calcul. Beaucoup d'instructions n'ont aucune opérande.

Il est courant sur les processeurs modernes d'aligner les codes d'instructions et les opérandes sur des mots de 32 ou 64 bits[1]. Du fait de la largeur de 8 bits des codes d'instructions et de la taille variable des opérandes, l'alignement sur plus de 8 bits aurait causé une perte d'espace mémoire (voir figure 2.2).

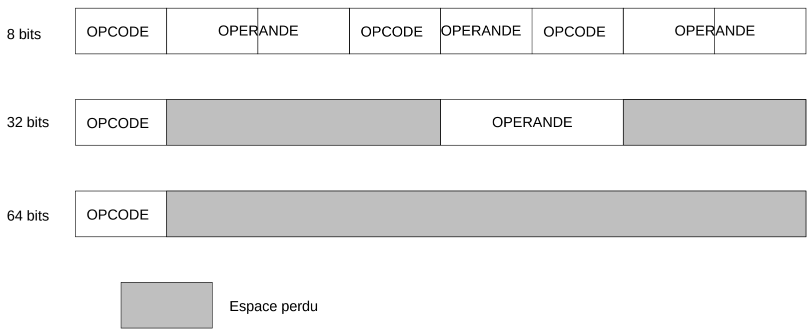

FIG. 2.2: Alignement sur 8, 32 et 64 bits

Ici, les concepteurs du système Java ont donc préféré réduire la taille du code généré par le compilateur plutôt que d'augmenter la vitesse du décodage des instructions.

2.2.2 Les registres et la pile

Tous les processeurs possèdent ce que l'on appelle des *registres*, qui contiennent des informations que le processeur utilise pour sauvegarder l'état du système.

Le jeu de registres qu'utilise la machine virtuelle Java sont les suivants :

pc le compteur de programme,
pt_stack le pointeur de sommet de pile,
frame le pointeur sur la frame d'exécution courante,
vars le pointeur sur la première variable locale de la méthode courante.

Tous ces registres ont une largeur de 32 bits et quelques un d'entre eux peuvent ne pas être utilisés dans certaines implémentations de la machine virtuelle.

Le compteur de programme pointe toujours sur l'instruction suivante que doit exécuter la machine virtuelle.

[1] Un format unique pour les instructions, alignées sur des adresses, permet de ne pas avoir à traiter le cas malencontreux d'un défaut de page au milieu de l'interprétation d'une instruction.

La machine virtuelle Java est une machine à pile. Les arguments des instructions doivent être empilés sur le sommet de la pile des opérandes avant que l'instruction soit exécutée [Floyd and Biegel, 1990]. En fin d'exécution le résultat de l'opération est empilé sur le sommet de pile. Le registre *pt_stack* pointe toujours sur le sommet de la pile des opérandes.

Avant qu'une méthode ne s'exécute, une *frame* est créée. Une *frame* contient les variables locales et la pile d'opérande.

Le registre *frame* pointe donc sur la frame en cours d'exécution et le pointeur *vars* référence les variables locales de cette dernière.

2.2.3 L'allocation mémoire

Les instances de classe, les objets, sont allouées dans la mémoire principale de la machine virtuelle. Après que cette mémoire ai été allouée, la machine virtuelle a la responsabilité de garder une trace de cette allocation, et, quand l'objet n'est plus utilisé, de libérer l'espace mémoire qui lui avait été attribué.

Les choix d'implémentation de ce mécanisme est laissé libre aux développeurs. Sur la machine virtuelle Java implémentée par Sun, ce mécanisme est réalisé par un *thread* d'exécution tournant en arrière plan.

Chapitre 3

Présentation de l'environnement de travail : Plan 9

Notre machine virtuelle est conçue pour interpréter du code Java sur le système s'exploitation Plan 9.

Nous présentons dans ce chapitre les concepts du système d'exploitation Plan 9 et donnons une description du réseau de machines Plan 9 que nous avons utilisé. Nous continuons par une description des outils de développement dont nous nous sommes servis et concluons en donnant notre avis personnel du système.

3.1 Le système d'exploitation Plan 9

Plan 9 est un système d'exploitation distribué conçu aux laboratoires Bell de AT&T [Pike et al., 1995]. Au milieu des années 1980, une équipe de chercheurs du Bell Labs[1] voulut créer un nouveau système d'exploitation à temps partagé en fondé sur de nouveaux concepts. Ce nouveau système devait être basé sur un réseau de machines accomplissant toutes des tâches distinctes ; de petites machine peu chères devaient servir de terminaux pour accéder à un large réseau de machines importantes et rapides, servant de serveurs de CPU et de systèmes de fichiers.

En 1989, le laboratoire Bell de AT&T utilise Plan 9 en interne. Ce nouveau système repose sur trois concepts fondamentaux :

- Toutes les ressources du système sont des systèmes de fichiers. Elles sont accessibles par un système de fichier hiérarchisé.

[1] La même équipe qui a développé Unix, C et C++

- Tous les accès aux fichiers sont articulés autour d'un protocole unique, le protocole 9P.
- Les différents services offerts par le système sont représentés par des fichiers. Ils sont dans un espace de nommage local modifiable.

Ces concepts permettent de distribuer toutes les ressources d'un système Plan 9 comme des fichiers. Les services qui exportent des hiérarchies de fichiers peuvent être des périphériques d'entrées-sorties, des services destinés à la sauvegarde régulière de système de fichier, le système de fenêtres, des interfaces réseaux. Les espaces de nommages des clients (les terminaux Plan 9) sont modifiables et fournissent une vue personnelle des ressources du système.

Les ressources qui sont donc représentées sous forme de fichiers se manipulent facilement à l'aide des appels systèmes `read` et `write`. Les fichiers peuvent exister localement ou ailleurs sur le réseau de manière transparente. Ils sont toujours manipulés par le protocole 9P.

Les machines d'un réseaux Plan 9

Un réseau de machines Plan 9 est constitué de trois types de machines bien distinctes.

1. Le serveur de fichier est une machine dédiée qui permet uniquement le stockage des fichiers. Un réseau Plan 9 peut posséder un ou plusieurs serveur de fichiers.
2. Le serveur de *cpu* est une machine puissante utilisée pour faire tourner des applications qui consomment beaucoup de mémoire et de temps *cpu*. Ces machines sont accessibles par des terminaux Plan 9 et permettent de pallier le manque de puissance des terminaux. Les serveurs de *cpu* servent aussi de serveurs d'authentification[2] ou de passerelles pour accéder à d'autres réseaux.
3. Les terminaux sont généralement des machines bon marché qui permettent d'accéder aux ressources du réseau Plan 9. Un utilisateur pour travailler *boote* Plan 9 sur un terminal.

Nous présentons sur la figure 3.1 la configuration du réseau de machines Plan 9 que nous avons utilisées pour développer notre machine virtuelle.

La machine `bootes` est le serveur de fichier du système Plan 9 de MIME. C'est un Pentium 100 doté de 16 Mo de mémoire vive. La machine `p9auth` est le serveur d'authentification. Il est chargé d'identifier les machines qui se connectent sur le réseaux Plan 9 ainsi que les utilisateurs du système. Cette machine est un PC 486 doté de 8 Mo de mémoire.

[2] Un serveur d'authentification est une machine spécialisée qui permet d'identifier une machine du réseau ou un utilisateur.

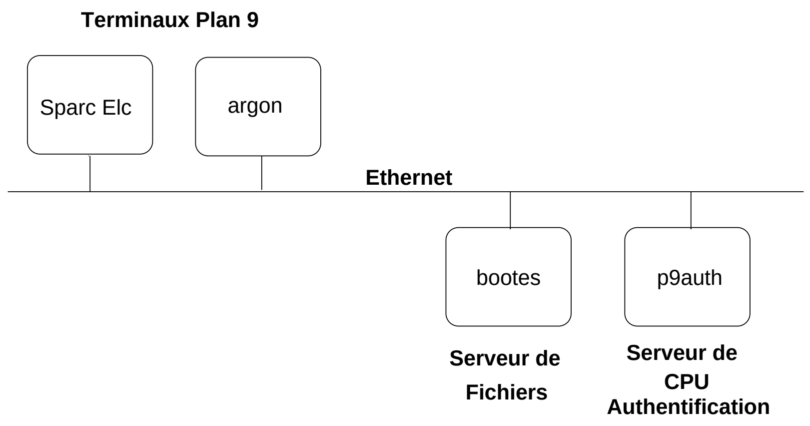

FIG. 3.1: Le réseau Plan 9 du département MIME

La machine `argon` est un terminal Plan 9. Elle est utilisée pour développer. C'est une Pentium 100 doté de 16 Mo de mémoire vive. Il existe de plus quelques terminaux qui peuvent être utilisés pour faire tourner Plan 9. Ce sont des stations Sun Sparc ELC.

3.2 Les outils de développement de Plan 9

Nous décrivons dans cette section les principaux outils que nous avons utilisé durant la réalisation de notre projet. Ces outils on été développés spécifiquement pour tourner sur Plan 9.

3.2.1 Le compilateur Plan 9

Le compilateur Plan 9 est un peu différent des autres compilateurs utilisés dans le monde Unix [Pike, 1995c]. C'est un ensemble de compilateurs pour les différentes architectures cibles sur lesquelle Plan 9 est susceptible de tourner[3].

Le langage accepté par les compilateurs est la base du C ANSI avec quelques extensions, un préprocesseur simplifié, une petite librairie pour les appels systèmes et une structure différente d'inclusion de fichiers.

Le compilateur est plus strict sur la déclaration des fonctions qui doivent utilisées le *nouveau style* définit par la norme ANSI. Le prototypage des fonctions est obligatoire.

[3]Les différentes architectures supportées sont les machines MIPS, SPARC, Motorola 6800 et Intel 386.

Un préprocesseur simplifié est intégré au compilateur. Il accepte uniquement les directives `#include`, `#define`, `#undef`, `#line`, `#ifdef`, `#ifndef` et `#endif`. Il ne supporte pas la directive `#if`.

Tous les programmes doivent obligatoirement commencer par

```
#include <u.h>
#include <libc.h>
```

Le fichier `u.h` contient tous les `typedefs` nécessaires aux autres fichiers à inclure. Le fichier `libc.h` contient toutes les définitions de la librairie standard.

Le compilateur est utilisé avec la commande `mk` [Flandrena, 1995], qui est une descendante de la commande `make` des systèmes Unix. Cette commande utilise notamment la variable d'environnement `objtype` pour déterminer quel compilateur utiliser lors de la génération d'un exécutable.

3.2.2 L'éditeur de fichier : Acme

Acme est l'éditeur de texte que nous avons employé lors de notre développement [Pike, 1995b]. Il est utilisable sur le système de fenêtrage $8\frac{1}{2}$ de Plan 9 [Pike, 1995a].

Acme est un éditeur de texte conçu tout spécialement pour le développement d'applications. Son interface est orienté texte. Elle s'inspire de l'éditeur d'Oberon [Reiser, 1991].

L'interface se présente par défaut en deux colonnes qui peuvent contenir des fenêtres. Ces dernières se décomposent en deux parties. La première partie d'une fenêtre contient une simple ligne de commandes qui peuvent s'appliquer à la fenêtre. La seconde partie, juste en dessous de la première, contient du texte qui représente soit des fichiers éditables, soit des résultats de commandes ou des programmes qui s'exécutent dans l'éditeur. Les fenêtre peuvent être déplacées dans un endroit quelconque de l'éditeur.

Dans Acme les trois boutons de la souris sont utilisés. Le premier, le bouton de gauche, permet de faire des sélections de texte. Le bouton du milieu permet d'exécuter une commande se trouvant sous le pointeur de la souris. Enfin, le bouton de droite permet de charger un fichier dans l'éditeur ou de faire des recherches rapides de chaînes de caractères dans un texte.

Pour éditer du texte, il suffit de déplacer le curseur de la fenêtre à l'endroit où l'édition doit être réalisée. Acme n'est pas *orienté curseur* et par conséquent tous les déplacements du curseur se font à l'aide de la souris. Nous pouvons éditer du texte n'importe où dans l'éditeur même dans la partie supérieure d'une fenêtre qui contient les commandes.

En plus de ces facultés d'édition et de manipulation des fichiers, Acme est programmable. Il possède ainsi par défaut un environnement spécial qui permet de lancer un shell, de lire son mail ou bien de lancer le débuggeur.

3.2.3 Le débugueur : Acid

Acid est un debuggeur symbolique pour le système Plan 9 entièrement programmable [Winterbottom, 1995a, Winterbottom, 1995b]. Il est basé sur un interpréteur avec des fonctions spécialisées pour le débuggage.

Acid est un outil très complet et pas très simple à utiliser au premier abord. Nous ne l'avons utilisé que pour faire des choses très simples qui nous permis d'avancer rapidement dans notre développement.

Chapitre 4

Lecture des fichiers compilés

Nous décrivons dans ce chapitre l'organisation des fichiers compilés Java et indiquons comment ils sont traités par notre machine virtuelle. Ils sont construits à partir du code source par le compilateur Java. Ces fichiers fournissent les exécutables de la machine virtuelle Java.

Nous commencons par décrire le format des fichiers générés par le compilateur Java. Puis nous montrons comment notre machine virtuelle réprésente de tels fichiers en mémoire et quels sont les mécanismes mis en place pour leurs chargements.

4.1 Le format des classes Java compilées

Un fichier Java compilé est constitué d'une suite d'octets. Toutes les entités 16 bits, 32 bits et 64 bits sont construites en lisant respectivement deux, quatre et huit octets consécutifs. Les valeurs entières (`short, int, long`) et valeurs flottantes (`float, double`) sont stockées dans un format *big-endian*[1].

4.1.1 Structure générale d'un fichier compilé

Un fichier compilé possède toujours l'extension `.class` et représente toujours une et une seule classe Java. La classe est représentée par la structure `ClassFile` suivante :

```
ClassFile {
    u4 magic;
    u2 minor_version;
```

[1]Les octets de poids forts sont stockés en premier, par opposition au format *little-endian*. Les systèmes basés sur des architectures 68000 ou Sparc représentent les données en format *big-endian* et les systèmes basés sur des architectures Intel en *little-endian*.

```
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count - 1];
        u2 access_flag;
        u2 this_class;
        u2 super_class;
        u2 interface_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attibutes_count];
    }
```

Notons qu'il n'y pas d'alignement mémoire réalisé dans la représentation que nous utilisons ici. Les fichiers sont considérés comme un simple flux d'octets. Nous définissons les types `u1` comme étant un octet non signé, `u2` un entier non signé codé sur 16 bits, `u4` un entier non signé codé sur 32 bits. Nous utiliserons cette notation dans toute la suite de ce chapitre.

Les différentes entrées de cette structure que nous appellerons items sont les suivants :

- `magic`
  L'item `magic` représente le nombre magique identifiant les fichiers Java compilés. Il doit avoir pour valeur `0xCAFEBABE`.
- `minor_version`
- `major_version`
  Les items `minor_version` et `major_version` donnent le numéro de version du compilateur qui a généré le fichier. Pour la version actuelle du JDK de Sun c'est à dire la version 1.0.2, la version du compilateur vaut 3.45 (3 pour l'item `major_version` et 45 pour l'item `minor_version`).
- `constant_pool_count`
  Cet item donne le nombre d'entrées contenues dans l'item `constant_pool`.
- `constant_pool[]`
  L'item `constant_pool` représente une table de structures de tailles variables. Nous y trouvons les différents symboles utilisés par le code contenus dans les méthodes. La première entrée `constant_pool[0]` est réservée et peut être utilisée par la machine virtuelle pour une utilisation interne. La première entrée valide est donc `constant_pool[1]`.
- `acces_flag`
  L'item `access_flag` est un masque de bits utilisé pour donner des modifications sur la déclaration de la classe suivant le tableau 4.1.

Nom	Valeur	Description
`ACC_PUBLIC`	0x0001	Visible par tout le monde
`ACC_FINAL`	0x0010	N'accepte pas les classes filles
`ACC_INTERFACE`	0x0200	Définit une interface
`ACC_ABSTRACT`	0x0400	Pas de définition fournie

TAB. 4.1: Modificateurs d'une classe

- `this_class`
  La valeur `this_class` doit être une entrée valide dans la table `constant_pool`. Elle représente une entrée de type `CONSTANT_Class` qui décrit la classe.
- `super_class`
  La valeur `super_class` doit être une entrée valide dans la table `constant_pool`. Elle représente une entrée de type `CONSTANT_Class` qui décrit la super classe (la classe mère).
- `interfaces_count`
  Cet item donne le nombre d'entrées contenues dans l'item `interfaces` suivant.
- `interfaces[]`
  L'item `interfaces` est un tableau contenant des entrées valides dans la table `constant_pool`. Chaque entrée représente une interface qu'implémente la classe.
- `fields_count`
  Cet item donne le nombre d'entrées contenues dans l'item `fields` suivant.
- `fields[]`
  L'item `fields` représente une table donnant la description complète des variables de la classe. Cette table contient exclusivement les variables de classe et d'instances déclarées par cette classe. Elle ne contient pas les variables héritées de ces classes parentes.
- `methods_count`
  Cet item donne le nombre d'entrées contenues dans l'item `methods` suivant.
- `methods[]`
  L'item `methods` représente une table donnant la description complète des méthodes de la classe. Chaque entrée fournit le nom, la signature, le code d'une méthode implémentée par cette classe. Au même titre que l'item `fields`, seules les méthodes déclarées explicitement par cette classe sont contenues dans la table.
- `attributes_count`
  Cet item donne le nombre d'entrées contenues dans l'item `attributes` suivant.

– `attributes[]`
  L'item `attributes` représente une table donnant une description des différents attributs associés à cette classe. Chaque classe peut avoir un nombre illimité d'attributs. Le seul attribut défini pour le moment par les spécifications de Sun est un attribut de type `SourceFile`. Si il existe d'autres attributs dans cette table il ne doivent en aucun cas modifier la sémantique du fichier compilé et ne sont présents que dans un but informatif.

4.1.2 Les signatures

Une signature est une chaîne de caractères définissant le type d'une variable ou les paramètres et le type de retour d'une méthode. Elles sont utilisées dans la table des symboles d'un fichier Java compilé dont nous parlerons dans la section suivante.

Les signature des champs de classes

La signature des champs de classes représente le type d'un argument d'une fonction ou le type d'une variable. La signature des champs de classes est définie par la grammaire suivante :

```
signature        := type
rtype            := type
type             := type_base | type_objet | type_tableau
type_base        := B | C | D | F | I | J | S | Z
type_objet       := L<nom de classe>;
type_tableau     := [ rtype
```

La valeur des types de base est la suivante :

`B`	`byte`	octet
`C`	`char`	caractère
`D`	`double`	flottant double précision
`F`	`float`	flottant simple précision
`I`	`int`	entier 32 bits
`J`	`long`	entier long 64 bits
`L<non de classe>;`	`...`	un objet de classe donnée
`S`	`short`	entier 16 bits
`Z`	`boolean`	vrai ou faux
`[`	`...`	une dimension pour un tableau

Ainsi une simple variable d'instance de type `int` est représenté par la signature `I`. La signature représentant une instance de la classe String serait `Ljava.lang.String;`. Enfin la signature pour une déclaration d'un tableau de `double` à trois dimensions en Java `double d[][][]`) donnerait `[[[D`.

Les signatures des méthodes

De façons à définir les signatures de méthode, il faut ajouter le fragment de grammaire suivant :

```
méthode_signature    := ( arguments_signature ) retour_signature
arguments_signature  := argument_signature *
argument_signature   := var_signature
retour_signature     := var_signature | V
```

Le caractère * signifie que le non-terminal `arguments_signature` est une suite non restreinte d'arguments [2]. Le caractère `V` indique que la méthode ne renvoie pas de valeur.

Ainsi la signature de la fonction Java :

```
String foo(int i, double d, Thread t)
```

serait

```
(IDLjava.lang.Thread;)Ljava.lang.String;
```

4.1.3 La table des symboles : la `constant_pool`

La `constant_pool` est la table des symboles des fichiers Java compilés. C'est une partie très importante du fichier Java compilé. Elle est en effet utilisée de façon continue par l'environnement d'exécution et par les instructions de l'assembleur Java.

Toutes les entrées de la table des symboles possèdent le format suivant :

```
cp_info {
    u1 tag;
    u1 info[];
}
```

Tous les items de la table des symboles doivent commencer par un octet, le champ `tag`, indiquant le type qu'il représente. Le contenu du champ `info` varie suivant le type du champ `tag`. Les différentes entrées de la `constant_pool` sont représentées dans le tableau 4.2.

Nous décrivons maintenant une à une les différentes entrées de la table des symboles.

`CONSTANT_Class`

Cette structure de la `constant_pool` représente la référence à une classe ou à une interface Java :

[2] En fait les spécifications Java limitent le nombre d'arguments d'une méthode à 255

Type	**Valeur**
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

TAB. 4.2: Les entrées de la table des symboles

```
CONSTANT_Class {
    u1 tag;
    u2 name_index;
}
```

Les champs de la structure `CONSTANT_Class` sont les suivants :

- `tag`
  Le champ `tag` doit avoir la valeur `CONSTANT_Class`.
- `name_index`
  La valeur du champ `name_index` doit être un index valide dans la `constant_pool`. L'entrée de la `constant_pool` à cet index doit être une entrée de type `CONSTANT_Utf8` qui représente le nom complet d'une classe Java.

CONSTANT_Fieldref,CONSTANT_Methodref, CONSTANT_InterfaceMethodref

Les variables de classes et d'instances, les méthodes et les méthodes d'interfaces sont représentées par des structures identiques :

```
CONSTANT_Fieldref {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}

CONSTANT_Methodref {
    u1 tag;
    u2 class_index;
```

```
    u2 name_and_type_index;
}

CONSTANT_InterfaceMethodref {
    u1 tag;
    u2 class_index;
    u2 name_and_type_index;
}
```

Les champs de ces structures sont les suivants :

- `tag`
  Le champ `tag` doit avoir la valeur `CONSTANT_Fieldref` pour une variable, la valeur `CONSTANT_Methodref` pour une méthode et la valeur `CONSTANT_InterfaceMethodref` pour méthode d'interface.
- `class_index`
  La valeur du champ `class_index` doit être un index valide dans la `constant_pool`. L'entrée de la `constant_pool` à cet index doit être une entrée de type `CONSTANT_Class` qui définit une classe Java pour les types `CONSTANT_Fieldref` et `CONSTANT_Methodref`. Pour le type `CONSTANT_InterfacMethoderef` l'entrée définie dans la `constant_pool` doit être une interface.
- `name_and_type_index`
  La valeur du champ `name_and_type_index` doit être un index valide dans la `constant_pool`. L'entrée de la `constant_pool` à cet index doit être de type `CONSTANT_NameAndType`. Cette entrée fournit le nom et la signature de la variable ou de la méthode.

`CONSTANT_String`

L'entrée de type `CONSTANT_String` représente un objet de type `java.lang.String` :

```
CONSTANT_String {
    u1 tag;
    u2 string_index;
}
```

Les champs de cette structure sont les suivants :

- `tag`
  Le champ `tag` doit avoir la valeur `CONSTANT_String`.
- `string_index`
  La valeur du champ `string_index` doit être un index valide dans la `constant_pool`. L'entrée de la `constant_pool` à cet index doit être une entrée de type `CONSTANT_Utf8` qui est une chaîne de caractères avec laquelle l'objet `java.lang.String` sera initialisé.

CONSTANT_Integer et CONSTANT_Float

Les entrées de type CONSTANT_Integer et CONSTANT_Float représentent des constantes entières et flottantes sur 32 bits :

```
CONSTANT_Integer {
    u1 tag;
    u4 bytes;
}

CONSTANT_Integer {
    u1 tag;
    u4 bytes;
}
```

Les champs de ces structures sont les suivants :

- tag
  Le champ tag doit avoir la valeur CONSTANT_Integer pour un entier et CONSTANT_Float pour un flottant.
- bytes
  Le champ bytes pour une entrée de type CONSTANT_Integer représente une valeur entière stockée dans l'ordre big-endian.
  Le champ bytes pour une entrée de type CONSTANT_Float représente une valeur flottante stockée dans l'ordre big-endian. Cette valeur numérique est codée suivant la norme IEEE 754.

CONSTANT_Long et CONSTANT_Double

Les entrées de type CONSTANT_Long et CONSTANT_Double représentent des constantes entières et flottantes double précision sur 64 bits :

```
CONSTANT_Long {
    u1 tag;
    u4 hight_bytes;
    u4 low_bytes;
}

CONSTANT_Integer {
    u1 tag;
    u4 hight_bytes;
    u4 low_bytes;
}
```

Les champs de ces structures sont les suivants :

- `tag`
  Le champ `tag` doit avoir la valeur `CONSTANT_Long` pour un entier et `CONSTANT_Double` pour un flottant double précision.
- `hight_bytes low_bytes`
  Les valeurs `hight_bytes` et `low_bytes` d'une structure `CONSTANT_Long` sont utilisées pour construire un entier sur 64 bits `(hight_bytes << 32) | low_bytes)`. Les valeurs `hight_bytes` et `low_bytes` sont stockées dans l'ordre big-endian.

  Les valeurs `hight_bytes` et `low_bytes` d'une structure `CONSTANT_Double` sont utilisées pour construire un flottant double précision utilisant la norme IEEE 754 `(hight_bytes << 32) | low_bytes`. Les valeurs `hight_bytes` et `low_bytes` sont stockées dans l'ordre big-endian.

CONSTANT_NameAndType

L'entrée de type `CONSTANT_NameAndType` définit le nom et la signature d'une variable ou d'une méthode :

```
CONSTANT_NameAndType {
    u1 tag;
    u2 name_index;
    u2 signature_index;
}
```

Les champs de cette structure sont les suivants :

- `tag`
  Le champ `tag` doit avoir la valeur `CONSTANT_NameAndType`.
- `name_index`
  La valeur du champ `name_index` doit être un index valide dans la `constant_pool`. L'entrée de la `constant_pool` à cet index doit être une entrée de type `CONSTANT_Utf8` qui représente le nom d'une variable ou d'une méthode d'une classe Java.
- `signature_index`
  La valeur du champ `signature_index` doit être un index valide dans la `constant_pool`. L'entrée de la `constant_pool` à cet index doit être une entrée de type `CONSTANT_Utf8` qui représente la signature d'une variable ou d'une méthode.

CONSTANT_Utf8

L'entrée de type `CONSTANT_Utf8` est utilisée pour une chaîne de caractère constante :

```
CONSTANT_Utf8 {
    u1 tag;
    u2 length;
    u1 bytes[length];
}
```

Les champs de cette structure sont les suivants :

- `tag`
  Le champ `tag` doit avoir la valeur `CONSTANT_Utf8`.
- `length`
  La valeur du champ `length` représente le nombre d'octets contenus dans le champ suivant.
- `bytes`
  Le champ `bytes` est un tableau contenant `length` caractère(s).

Les champs de classes

Les champs de classes sont les variables de classes[3] et les variables d'instances. Elles sont représentées dans un fichier compilé par la structure suivante :

```
field {
    u2 access_flag;
    u2 name_index;
    u2 signature_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Les champs de cette structure sont les suivants :

- `access_flag`
  L'item `access_flag` représente un champ de bits indiquant les droits d'accès et les propriétés de la variable (voir tableau 4.3).
- `name_index`
  La valeur de `name_index` doit être un index valide dans la *constant pool*. L'entrée dans la *constant pool* à cet index doit être de type `CONSTANT_Utf8` et représente le nom de cette variable.
- `signature_index`
  La valeur de `signature_index` doit être un index valide dans la *constant pool*. L'entrée dans la *constant pool*à cet index doit être de type `CONSTANT_Utf8` et représente la signature de cette variable.

[3]Les variables de classes sont les variables statiques des classes Java. Elles sont définies en utilisant le mot clé `static`.

Nom	Valeur	Description
`ACC_PUBLIC`	0x0001	Accessible par tout le monde
`ACC_PRIVATE`	0x0002	Accessible par la classe seulement
`ACC_PROTECTED`	0x0004	Accessible par les classes filles
`ACC_STATIC`	0x0002	Variable de classe
`ACC_FINAL`	0x0010	Valeur constante

TAB. 4.3: Modificateurs de variables

- `attributes_count`
  La valeur de `attributes_count` indique la taille du champ `attributes` suivant.
- `attributes[]`
  Le champ `attributes` représente une table contenant les différents attributs associés à cette variable.

  Le seul attribut reconnu pour un champ de classe est de type `ConstantValue`. Il contient une valeur numérique.

Les méthodes

Chaque méthode d'une classe Java est décrit par la structure suivante :

```
method {
    u2 access_flag;
    u2 name_index;
    u2 signature_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

Les champs de cette structure sont les suivants :

- `access_flag`
  L'item `access_flag` représente un champ de bits indiquant les droits d'accès et les propriétés de la méthode (voir tableau 4.4).
- `name_index`
  La valeur de `name_index` doit être un index valide dans la *constant pool*. L'entrée dans la *constant pool*à cet index doit être de type `CONSTANT_Utf8` et représente le nom de la méthode.
- `signature_index`
  La valeur de `signature_index` doit être un index valide dans la *constant pool*. L'entrée dans la *constant pool*à cet index doit être de type `CONSTANT_Utf8` et représente la signature de la méthode.

Nom	**Valeur**	**Description**
`ACC_PUBLIC`	0x0001	Accessible par tout le monde
`ACC_PRIVATE`	0x0002	Accessible par la classe seulement
`ACC_PROTECTED`	0x0004	Accessible par les classes filles
`ACC_STATIC`	0x0002	Méthode de classe
`ACC_FINAL`	0x0010	Pas de surchargement
`ACC_SYNChRONIZED`	0x0020	Accès synchronisé (monitor)
`ACC_NATIVE`	0x0100	Implémenté dans un autre langage
`ACC_ABSTRACT`	0x0400	Méthode abstraite, pas d'implémentation

TAB. 4.4: Modificateurs de méthodes

- `attributes_count`
  La valeur de `attributes_count` indique la taille du champ `attributes` suivant.
- `attributes[]`
  Le champ `attributes` représente une table contenant les différents attributs associés à cette méthode.

  Les seuls attributs reconnus pour une méthode sont de types `Code` et `Exception`.

  L'attribut `Code` contient le code Java de la méthode et des informations nécessaires à son exécution comme le nombre de variables locales et la taille de la pile des opérandes.

  L'attribut `Exception` contient la table des exceptions gérées par la méthode.

4.2 Le chargement et représentation interne

Avant que la machine virtuelle Java puisse exécuter un programme, la première chose qu'elle doit faire est de retrouver et de charger les fichiers compilés qui constituent le programme. Le chargement est le processus qui permet d'obtenir le flux d'octets qui représente une classe compilée Java pour construire un objet représentatif manipulable par notre machine virtuelle.

De façon générale une machine virtuelle Java, comme celle de Sun ou celle du navigateur Netscape, peuvent retrouver les fichiers compilés à partir du disque local, à partir du réseau ou en générant elle même la classe sous un forme de flux d'octets.

Notre machine virtuelle est capable de retrouver et charger les fichiers compilés sur le disque local où elle s'exécute. Elle n'est pas capable de retrouver les exécutables à partir du réseau[4].

[4]En fait ce type de chargement est utilisé surtout par les machines virtuelles s'exécutant

4.2.1 Représentation interne d'un fichier compilé

Le choix d'une bonne représentation des fichiers compilés pour notre machine virtuelle nous a paru longtemps difficile. Il s'est avéré que la structure même du fichier tel que nous l'avons vu dans la section précédente est bien adaptée à une représentation structurée que l'on utilise courament en langage C. C'est pour cela que nous avons retranscrit les différentes structures de données définies pas les spécifications Java pour construire notre représentation d'un fichier compilé [5].

La structure d'un fichier compilé représentant un classe Java est décrit par le structure `ClassFile` suivante :

```
struct ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;

    u2 constant_pool_count;
    cp_info *constant_pool;

    u2 access_flags;
    u2 this_class;
    u2 super_class;

    u2 interfaces_count;
    u2 *interfaces;

    u2 fields_count;
    field_info *fields;

    u2 methods_count;
    method_info *methods;

    u2 attributes_count;
    attribute_info *attributes;

   /*
    * runtime infomation
    */
```

sur les navigateurs et plus généralement toute machine virtuelle pouvant faire tourner des Applets Java.

[5]Nous ferons remarquer que le choix des structures d'exécution de la machine est bien plus important encore. Le fichier compilé Java ne sert qu'à construire ces structures d'exécution et se sont ces dernières qui jouent un rôle important tant au niveau de la rapidité d'exécution que sur la complexité du code qu'elles engendrent.

```
    char *r_name;
    char *r_file;
    int r_state;/* CF_LINKED, CF_INITIALIZED */
    ClassFile *r_super;
    Object *r_java_class;
    int *r_fields;
    int *r_resolved;
    MethodBlock *r_methods;
};
```

Les différents champs de cette structure sont construits de la même manière et ont les mêmes valeurs que ceux décrits dans la section précédente. Nous avons ajouté quelques informations qui sont nécessaires à la machine virtuelle pendant l'exécution. Ces informations que nous décrirons un peu plus tard sont principalement construites durant la phase d'édition de liens et de préparation de la classe (voir section 5.3).

Nous décrivons brièvement les structures importantes en commençant par la table des symboles.

```
struct cp_class { u2 name_index; };
struct cp_method { u2 class_index; u2 name_and_type_index; };
struct cp_string { u2 string_index; };
struct cp_int_float { u4 bytes; };
struct cp_long_double { u4 hight_bytes; u4 low_bytes; };
struct cp_nametype { u2 name_index; u2 signature_index; };
struct cp_asciz { u2 length; u1 *bytes; };

struct cp_info {
    u1 tag; /* CONSTANT_* */
    union {
        cp_class c;
        cp_method m;
        cp_string s;
        cp_int_float i;
        cp_long_double l;
        cp_nametype n;
        cp_asciz a;
    } u;
};
```

Comme nous l'avons vu précédemment la table de symboles d'un fichier compilé Java est une table contenant une suite de structures de tailles différentes. L'entrée `constant_pool` de la structure `ClassFile` est un pointeur sur une structure `cp_info`. Cette entrée est allouée à la lecture du champ `constant_pool_count` pour former un tableau de taille `constant_pool_count`.

La structure `cp_info` est constituée d'un indicateur (le champ `tag`) qui détermine le type du symbole et par conséquent le type valide pour l'union `u` de la structure `cp_info`. Chaque symbole est représenté par une structure personnelle le décrivant complètement. Ainsi une classe est représentée par une structure `cp_info` dont le champ `tag` vaut `CONSANT_Class` et l'union `u` représente une structure de type `cp_class`.

Un autre champ important de la structure `ClassFile` est celle décrivant les méthodes définies par cette classe. Le code suivant décrit sa structure :

```
struct method_info {
    u2 access_flag;
    u2 name_index;
    u2 signature_index;
    u2 attributes_count;
    struct method_attribute {
        u1 type; /*  ATTR_CODE or ATTR_EXCEPTION */
        union {
            Code_attribute *ca;
            Exception_attribute *ea;
        } u;
    } *attributes;
};
```

La structure `method_info` représente la définition d'une méthode. Le principe d'allocation de la table des méthodes se fait à l'identique de la table des symboles en utilisant la valeur du champ `methods_count` de la structure `ClassFile`.

Ici, dans le cas d'une méthode, il faut noter que la structure `method_info` ne contient pas directement le code de la méthode. Nous trouvons directement le nom et le signature de la méthode (les champs `name_index` et `signature_index`) ainsi que ces modificateurs d'accès (le champ `access_flag`). Les spécifications Java utilisent pour cela un attribut particulier, l'attribut de type `Code` définit par la structure suivante :

```
struct Code_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 *code;
    u2 exception_table_length;
    struct exception {
        u2 start_pc;
        u2 end_pc;
```

```
        u2 handler_pc;
        u2 catch_type;
    } *exception_table;
    u2 attribute_count;
    struct code_attribute {
        u1 type; /* ATTR_LINE_NUMBER or ATTR_LOCAL_VARIABLE */
        union {
            LineNumberTable_attribute la;
            LocalVariableTable_attribute va;
        } u;
    } *attribute;
};
```

La structure `Code_attribute` nous donne toutes les informations nécessaires à l'exécution de la méthode. On y retrouve la taille maximum de la pile (le champ `max_stack`) qu'est susceptible d'utiliser la méthode, le nombre maximum de variables locales (le champ `max_locals`) qu'utilise la méthode et enfin le code qui la définit (le champ `code` qui possède une taille de `code_length`).

L'attribut de code définit aussi la table des exceptions qui permet d'attraper et de détourner le déroulement normal d'une méthode. Nous décrirons la gestion des exceptions par notre machine virtuelle au chapitre 6.

Deux attributs supplémentaires sont définis par la structure `Code_attribute` :

- L'attribut de type `Line_Number_Table` qui donne une table de correspondance entre le compteur de programme et le code source qui l'a généré. Il est notamment utilisé dans la gestion des exceptions pour indiquer à quel endroit dans le code une exception s'est produite.
- L'attribut de type `Local_Variable_Table` qui donne une correspondance entre l'emplacement d'une variable dans la frame d'éxecution et son nom symbolique dans le code source[6].

En résumé, la structure interne que nous avons choisie pour représenter les fichiers compilés est semblable à celle que l'on retrouve dans les spécifications Java. Nous n'avons pas éprouvé le besoin de modifier ces structures car elles sont facilement représentables en langage C. Toutes les structures de données n'ont pas été présentées exhaustivement dans cette section mais ce que nous avons détaillé donne une idée concrète du principe qui a été utilisé.

[6]Cet attribut n'est pas utilisé par notre machine virtuelle actuellement. Il est principalement utilisé par les debugers Java.

4.2.2 Le chargement des fichiers compilés

Une machine virtuelle Java utilise pour le chargement des fichiers compilés deux mécanismes. Le premier, celui dont nous avons doté notre machine virtuelle, définit un chargeur de classe principale ou que l'on appelle chargeur système. Il a pour but de charger les classes des API Java et la classe qui se trouve sur la ligne de commande de l'interpréteur. Le second utilise le chargement de classe via une classe particulière de l'API la classe `ClassLoader`. Cette classe met en place une politique de chargement. Les programmeurs Java peuvent redéfinir une classe `ClassLoader` afin de modifier le comportement par défaut du chargeur de classe[7].

Comme nous l'avons déjà mentionné un peu plus tôt, notre machine n'utilise pas de chargeur de classe définissant une nouvelle instance de la classe `ClassLoader`[8]. Par contre elle possède un chargeur système. C'est ce dernier que nous décrivons.

Le but du chargeur système est de retrouver les fichiers compilés sur le système de fichier local quand l'environnement d'exécution le lui demande. Il prend en entrée le nom d'une classe et fournit en sortie la classe sous la forme d'une structure `ClassFile` (voir figure 4.1).

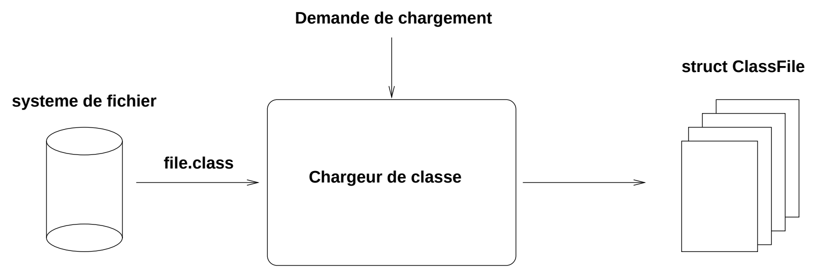

FIG. 4.1: Le chargeur de classe système

Recherche des fichiers compilés sur le système de fichier

La machine virtuelle afin de localiser les fichiers compilés sur le système de fichier utilise une variable interne qui contient une liste de répertoires.

[7]La machine virtuelle mémorise la façon dont une classe est chargée (i.e. avec quel `ClassLoader`). Quand une classe chargée par la machine virtuelle demande le chargement d'une autre classe, cette dernière est chargée en utilisant le même chargeur de classe que celle qui en a fait la demande. Le chargeur de classe crée son propre espace de nommage; ainsi les classes chargées par un chargeur de classes donné ne sont pas nécessairement accessibles par d'autres classes.

[8]Ce mécanisme est surtout utilisé par les machines virtuelles tournant dans des navigateurs Web

Cette liste de répertoires lui indique les endroits où elle est susceptible de trouver les fichiers compilés.

Il existe sur notre machine virtuelle deux manières de déterminer cette liste de répertoires.

1. La plus première solution est de positionner la variable d'environnement `java_class_path`. Par exemple sur le prompt du shell Plan 9 on tapera :

   ```
   $ java_class_path='. /lib/java/classes'
   ```

   Ceci indique que la recherche de fichiers compilés pourra se faire sur le répertoire courant où la machine virtuelle s'exécute et dans le répertoire `/lib/java/classes`. Cette méthode de lecture de la variable d'environnement est la méthode par défaut utilisé par notre machine virtuelle[9].
2. La seconde solution consiste à ignorer la variable d'environnement en utilisant l'option de `-classpath` de notre machine virtuelle lors du lancement d'une application Java sur la ligne de commande. Par exemple

   ```
   $ run -classpath '. /lib/java/classes' essai
   ```

   aura le même effet pour la recherche des fichiers que l'exemple que nous avons donner précédement dans le cas de la variable d'environnement.

Avant tout chargement et toute exécution la machine virtuelle détermine cette variable contenant les répertoires par l'une des deux méthodes que l'on vient de décrire et en construit la liste. Ainsi, lors de la demande de chargement d'une classe particulière le chargeur construit un nom de fichier absolu composé du premier répertoire de la liste suivi du nom de la classe auquel on ajoute l'extension `.class`. Si ce fichier n'existe pas, le chargeur construit un nouveau nom de fichier en prenant le second nom de repertoire dans la liste. Le processus recommence jusqu'à ce que le fichier soit trouvé ou que la liste des répertoires soit complètement parcourue. Dans ce dernier cas le chargeur de classe s'arrête en renvoyant un message d'erreur.

Chargement récursif des fichiers compilés

Chaque instance de classe Java possède une classe parente. Le sommet de la hiérarchie des objets Java est toujours un objet de la classe `java.lang.Object`. A ce titre quand le chargeur de fichiers reçoit l'ordre de charger une classe, il charge récurssivement la classe qu'on lui demande ainsi

[9] Le système Plan 9 utilise pour rechercher les commandes la variable d'environnement `path='. /bin'`. Lors de l'ouverture d'une session sur un terminal Plan 9, les répertoires contenant les exécutables correspondant à l'architecture du terminal sont *accrochés* au répertoire `/bin` à l'aide de la commande `bind`.

que toutes ces classes parentes[10]. La recursion s'arrête quand le chargeur tente de charger une classe déjà chargée ou qu'il remonte jusqu'à la classe `java.lang.Object`.

Pour éviter de charger plusieurs fois la même classe et tenir à jour une liste unique des classes déjà chargées, notre machine virtuelle tient à jour une table de *hash* contenant toutes les structures `ClassFile`[11].

Lecture des fichiers

Lorsque le fichier à charger est repéré par le chargeur, celui-ci peut l'ouvrir et commencer à le lire. Il crée une structure `ClassFile` qu'il remplit au fur et à mesure qu'il progresse sur le flux d'octets.

La lecture des éléments du fichier compilé se fait au moyen de primitives de lecture de un octet, de deux octets consécutifs formant un entier sur 16 bits et de quatre octets consécutifs formant un entier sur 32 bits. Comme nous l'avons vu les valeurs numériques stockées dans les fichiers compilés sont ecrites dans l'ordre *big-endian* ou *MSB-First*[12]. Or, notre machine virtuelle tourne sur le système Plan 9 qui lui s'exécute sur diverses plateformes matérielles[13]. Ainsi notre machine virtuelle doit savoir à tout moment sur quel type d'architecture matérielle elle s'exécute afin qu'elle représente les valeurs numériques qu'elle lit des fichiers compilés de manière correcte. Pour cela nous avons ajouté la constante `LITTLE_ENDIAN` qui quand elle est définie ajoute à notre machine virtuelle des fonctions de conversion du format *big-endian* vers le format *little-endian* pour les entiers. Cette constante doit être définie au moment de la compilation pour toutes les architectures *little-endian*[14]. Ces fonctions de conversion sont insérées dans les primitives de lectures ce qui permet à la machine de ne les exécuter une et une seule fois lors du chargement du fichier.

Dans un souci d'optimisation du temps de chargement des fichiers compilés les primitives de lecture sont écrites en utilisant une librairie qui *bufferise* les entrées sorties. Nous n'utilisons pas la classique librairie `stdio` mais une

[10] Le chargement réccurssif des classes vient du fait que pour initialiser une classe (processus qui intervient obligatoirement après le chargement et que nous décrirons au chapitre 5) il faut absolument que la classe parente soit elle aussi initialisée donc bien évidemment chargée.

[11] Nous utilisons une table où la clé de hash est constituée du nom de la classe. La foncion de calcul de la clé est tirée de [Aho et al., 1991]. Nous utilisons cette méthode afin de permettre à notre machine virtuelle de retrouver rapidement la définition d'une classe. Ce procédé permet d'effectuer r recherches de n noms en un temps proportionnel à $n(n+r)/m$ ou m est la taille de notre table.

[12] MSB-first (Most Signifcant Byte-first) signifie que les octets de poids fort sont stockés en tête de l'entier. Cet ordre est aussi appelé ordre réseau.

[13] Les différentes plateforme ou tourne Plan 9 actuellement sont : intel (386 et supérieur), mips, sparc, 68xxx et Power PC

[14] Par exemples les architectures Intel 386 et processeurs supérieurs

librairie spécifique Plan 9 qui s'appelle Bio.

4.3 Lecture des archives au format zip

Le chargeur système de notre machine virtuelle est capable de retrouver les fichiers dans des archives au format zip en plus de localiser et de lire les fichiers Java sur le disque local. Ces archives contiennent en général un ensemble de fichiers compilés qui représente un ou plusieurs packages Java[15].

Pour indiquer à notre machine virtuelle qu'elle doit chercher dans une archive au format `zip`, nous devons ajouter à la variable d'environnement `java_class_path` une entrée indiquant le nom complet de l'archive. Par exemple si la variable `java_class_path` est initialisé de la façon suivante :

```
$ java_class_path='. ../exemples /lib/java/classes.zip'
```

Le chargeur de classes saura qu'il doit chercher les classes sur le répertoire courant puis dans le répertoire `../exemples` et enfin dans une archive dont le nom absolu est `/lib/java/classes.zip`. Pour cela notre machine virtuelle ajoute à la liste de répertoires de recherche une entrée particulière pour les archives au format `zip`.

Une archive au format `zip` constitue un fichier unique contenant un ensemble de fichiers. La machine virtuelle lors de la création de la liste des répertoires et des archives crée pour toutes les archives une table de recherche contenant tous les noms des fichiers contenus dans l'archive, ainsi que l'emplacement et la taille exacte des fichiers dans l'archive[16]. Cette table permet à tout moment de savoir si un fichier donné appartient à l'archive et de le localiser. Cette table de recherche est triée et la recherche de fichier se fait de manière dichotomique.

L'utilisation de telles archives pour notre machine virtuelle nous a permis de régler le problème des noms de fichiers longs sur le système de fichier de Plan 9. Le système de fichier Plan 9 n'autorise pas la création d'un fichier dont le nom possède plus de 27 caractères. Cette limite semble convenable dans la plupart des cas. Un utilisateur créant un fichier peut toujours s'arranger pour créer un nom de fichier se pliant à cette limite.

Le problème est que le nom d'un fichier Java doit porter le nom de la classe qu'il définit. Ainsi les API Java (qui respectent bien sur cette règle)

[15] Java définit dans ses spécifications la notion de package. Un package regroupe un ensemble de classes Java compilées. Il peut être comparé à une librairie C. Les packages possèdent des règles très strictes d'accessibilité pour les classes qui y sont définies.

[16] Il faut noter que les archives utilisées par les machines virtuelles ne sont pas des archives compressées. Les archives au format `zip` sont très populaires dans le monde *dos* et *windows* et sont surtout utilisées pour compresser un ensemble de fichiers ou de répertoires.

définissent des classes portant des noms de plus de 27 caractères. Par exemple, la classe

`java.lang.ArrayIndexOutOfBoundsException`

qui est utilisée pour le lancer une exception lors d'un débordement de tableau, est compilée dans un fichier portant le nom

`ArrayIndexOutOfBoundsException.class`

qui est composé de plus de 27 caractères et donc impossible à représenter sur le système de fichier Plan 9.

Si de tels fichiers ne sont pas représentables comme un fichier Plan 9, ils peuvent l'être dans une archive au format *zip*. Ainsi notre machine virtuelle peut accéder simplement aux classes systèmes. Pour ce qui est des classes crées par les utilisateurs sur Plan 9, ils n'ont pas d'autres choix que de créer des noms de classes contenant moins de 21 caractères[17].

4.4 Le désassembleur de classe

Après la réalisation du chargeur de classe nous avons pensé l'utiliser afin de réaliser un désassembleur de fichier compilé.

Notre désassembleur prend donc en entrée un fichier Java compilé et affiche en sortie tout le contenu du fichier sous une forme lisible. Il est capable d'afficher le contenu de la table des symboles et le contenu des structures définissant les méthodes[18].

Exemple d'utilisation

Soit le fichier source Java suivant :

```
public class Essai {
    public int entier;
    static int poids;

    Essai(){    //constructeur
        poids = -1;
    }
    public static void main( String[] args ){
        int i = 10;
```

[17] 21 caractères pour le nom plus 6 caractères d'extension `.class` ajoutés par le compilateur font bien les 27 caractères qu' impose le système de fichier

[18] Notons qu'un désassembleur de classe existe. Il est fournit par défaut dans le JDK de Sun et s'appelle `javap`. Remarquons aussi qu'il donne simplement le code des méthodes et pas la table des symboles.

```
            poids *= i;
        }
        int retournePoids(){
            return poids * entier;
        }
}
```

En tapant sur la ligne de commande

```
$ dasm Essai.class
```

nous obtenons sur la sortie standard le code suivant :

```
...

 name = <init>
 max_stack = 1
 max_locals = 1
 code_length = 9
    0 aload_0
    1 invokenonvirtual #10 <Method java/lang/Object.<init>()V>
    4 iconst_m1
    5 putstatic #14 <Field Essai.poidsI>
    8 return

 name = main
 max_stack = 3
 max_locals = 2
 code_length = 10
    0 bipush 10
    2 istore_1
    3 getstatic #14 <Field Essai.poidsI>
    6 iload_1
    7 imul
    8 putstatic #14 <Field Essai.poidsI>
    9 return

 name = retournePoids
 max_stack = 2
 max_locals = 1
 code_length = 9
    0 getstatic #14 <Field Essai.poidsI>
    3 aload_0
    4 getfield #15 <Field Essai.entierI>
    7 imul
    8 ireturn
```

Nous ne montrons ici que la partie définissant les méthodes de la classe désassemblé. Nous retrouvons bien les différentes entrées dont nous avons déjà parlées et qui décrivent les frames d'exécution des méthodes (`max_stack` et `max_locals`). Nous trouvons aussi les instructions définissant les méthodes. Pour cela nous avons dû ajouter derrière le chargeur de classe un analyseur de code qui décode les instructions et leurs arguments. Nous verrons plus tard (ref) comment sont représentés les instructions et comment nous les décodons.

Motivations

Nous avons réalisé le désassembleur pour plusieurs raisons.

- Cela nous a permis de comprendre comment est organisé le code machine Java des méthodes.
- On peut ainsi mieux comprendre la sémantique des instructions Java. Nous pouvons notamment repérer plus facilement les références faites à la *constant pool*.
- Le désassembleur nous a aidé lors de la réalisation de l'interpréteur. Il nous a permis de suivre pas à pas l'interpréteur lors de son exécution.

Chapitre 5

Notre implémentation de la Machine Virtuelle

Ce chapitre est consacré à l'étude et à la description des mécanismes que nous avons mis en œuvre pour l'implémentation de notre machine virtuelle Java sur le système d'exploitation Plan 9. Le but principal d'une telle machine est l'exécution de programmes Java. Son moteur d'exécution est basé sur un interpréteur qui décode et exécute le code machine contenu dans les méthodes de toutes applications Java.

Pour cela nous décrivons dans un premier temps l'environnement d'exécution de notre machine virtuelle afin de pouvoir dans un deuxième temps décrire le fonctionnement de l'interpréteur. Nous décrivons par la suite les mécanismes de l'édition de liens et de résolution qui sont réalisés afin de respecter la sémantique objet du code Java à l'exécution. Nous terminons par la description des objets Java et la façon dont le démarrage de notre machine est réalisé.

5.1 L'environnement d'exécution

L'interpréteur Java repose sur des structures de données dynamiques que l'on appelle des *frames* pour exécuter les méthodes. Ces *frames* contiennent des cellules qui jouent le rôle des registres (pointer de pile, compteur ordinal, tableau de variables locales) dans un microprocesseur classique.

A chaque exécution d'une méthode par l'interpréteur une nouvelle *frame* est créée. C'est sur cette *frame* que l'interpréteur opère.

5.1.1 La structure des *frames*

Une *frame* est composée de plusieurs éléments comme le montre la figure 5.1.

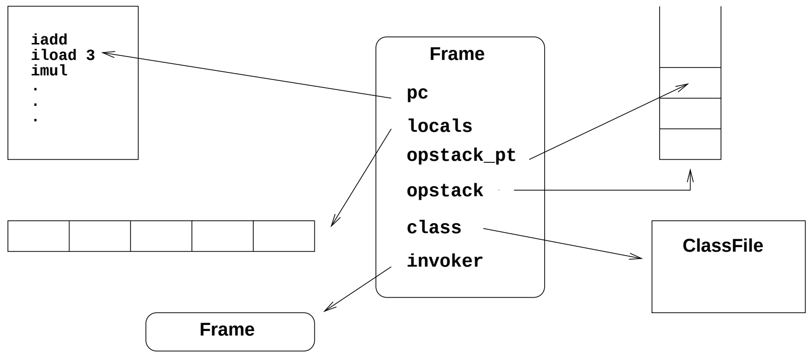

FIG. 5.1: Structure d'une frame d'exécution

La frame contient un pointeur de programme (le champ `pc`) qui pointe toujours sur l'instruction en cours d'exécution. Ce pointeur se déplace sur l'instruction suivante à chaque fois que l'exécution de l'instruction courante est achevée. Les instructions qui définissent les sauts dans un programme modifient directement le pointeur de programme de la *frame* courante pour aller pointer une instruction autre que la suivante. Lors de la création d'une *frame* le pointer de programme pointe sur la première instruction de la méthode. Le début du code définissant une méthode est retrouvé à l'aide de la structure `ClassFile` construite au chargement de la classe.

La machine virtuelle ne possède pas de pile où les arguments et les variables locales des fonctions sont empilés. Pour cela les variables locales d'une méthode sont stockées dans le tableau contenu dans la *frame* d'exécution (le champ `locals` de la figure 5.1). Ce tableau est alloué à la création de la *frame* et sa taille est déterminée par un champ se trouvant dans l'attribut de code de la méthode (voir la section 4.1.3). Notre machine virtuelle ne possède que deux primitives qui opèrent sur les variables locales.

1. Mettre une valeur dans une variable locale.
2. Récupérer le contenu d'une variable locale.

Notre machine possède un jeu de primitives de manipulation des variables locales par type de variable qu'elle manipule. Les types `int` sur 32 bits, `float` valeur flottante simple précision, `vlong` entier sur 64 bits et `double` valeurs flottantes double précision sont les types manipulés par l'ensemble des composantes de notre machine virtuelle. Il existe donc un jeu de primitives pour chacun des types précédents.

- `setlocal()`, `setlocalfloat()`, `setlocallong()`, `setlocaldouble()` qui prennent en argument un tableau de variables locales, un emplacement dans ce tableau, une valeur et la stocke à l'emplacement prévu.
- `getlocal()`, `getlocalfloat()`, `getlocallong()`, `getlocaldouble()` qui prennent en argument un tableau de variables locales, un emplacement dans ce tableau et retourne la valeur de la variable locale logée à l'emplacement demandé.

Nous noterons que les valeurs stockées dans les variables locales sont stockées dans l'ordre natif.

La machine virtuelle Java est une machine à pile. En ce sens les instructions Java opèrent sur une structure de pile. Ce sont les *frames*d'exécutions qui contiennent les piles. Une pile est définie de manière unique pour chaque méthode exécutée. Les instructions récupèrent leurs opérandes sur la pile et empilent leurs résultat sur le sommet. Les *frames* de notre machine virtuelle définissent un tableau `opstack` et un pointeur de pile `opstack_pt` implémentant le mécanisme de pile. Lors de la création d'une *frame* notre machine utilise une valeur se trouvant dans l'attribut de code de la méthode pour construire la pile et initialise le pointeur de pile `opstack_pt` à -1.

Au même titre que les variables locales, les manipulations de la pile des opérandes se font à travers deux primitives : la première empile une valeur sur le sommet de pile et la seconde dépile la valeur du sommet de pile. Il existe aussi quatre jeux de primitives, un par type de données manipulé par notre machine.

- `push()`, `pushFloat()`, `pushLong()`, `pushDouble()` qui empilent une valeur passée en argument sur le sommet de la pile des opérandes.
- `push()`, `pushFloat()`, `pushLong()`, `pushDouble()` qui dépilent une valeur du sommet de la pile des opérandes.

Nous retrouvons aussi dans la *frame* une référence sur la classe de la méthode. Ce pointeur sur une structure `ClassFile` nous permet en cours d'exécution de retrouver notamment la *constant pool*et les valeurs des variables de classes[1].

Pour pouvoir garder la trace des appels de fonctions la *frame* contient aussi un pointeur sur la *frame* qui en a fait la création. Les *frames* d'exécution

[1]Beaucoup d'instructions machine Java font référence à la table des symboles, il est donc important de pouvoir la retrouver très rapidement en cours d'exécution et donc d'en sauvegarder une référence par l'intermédiaire de la structure `ClassFile` dans la *frame* courante.

sont ainsi chaînées. Ce mécanisme a été utile pour l'implémentation des fonctions natives Java qui permettent de gérer les exceptions que nous décrirons au chapitre 6.

5.1.2 Gestion de la pile de frame

Un programme en général fait appel à plusieurs fonctions qui à leur tour appellent d'autres fonctions. L'interpréteur gère la création de ces différentes *frames* qui leurs sont associées par un mécanisme de pile. Toutes les *frames* construites sont empilées sur une pile de *frames*.

A chaque appel de fonction une *frame* est créée et initialisée. Elle est ensuite poussée sur la pile de *frames*. Lorsque la fonction termine son exécution la *frame* est dépilée de la pile et l'espace mémoire qu'elle occupait est rendu au système.

Ce mécanisme permet d'isoler correctement l'exécution du code par la pile de *frames*. Ceci est particulièrement appréciable pour la gestion des *threads*. Chaque *thread* possède sa propre pile de *frames*qui constitue son environnement personnel d'exécution.

De plus ce mécanisme permet de résoudre le problème des fonctions récursives ne se terminant pas. Une fonction récursive s'appelle elle même et crée donc une *frame* à chaque fois qu'elle s'appelle. Si toutefois cette fonction ne s'arretait pas, il y aurait sur la pile de *frames* un empilement infini de *frames*. Or en contrôlant la taille de la pile de *frames* la machine virtuelle peut décider de stopper l'exécution du programme en lançant une exception. Ceci permet au développeur Java de déceler ce type d'erreur plus facilement.

La taille maximum de la pile de *frames* de notre machine virtuelle est contrôlée par une option de lancement. L'option `-stacksize` placée sur la ligne de commande, prend en argument un entier qui indique la taille maximum de la pile de *frames* en nombre d'octets. Par défaut notre machine vituelle accepte une pile de 16 Ko par *threads*.

Ce mécanisme est implémenté en associant à chaque pile de *frames* une variable contenant la taille en nombre d'octets qu'elle occupe. A chaque empilement d'une nouvelle *frame*, cette variable se voit augmentée de l'espace mémoire occupé par la pile des opérande et les variables locales. Lorsqu'il y a dépilement de la *frame* se trouvant sur le sommet de pile, la variable est réajustée pour refléter l'état actuel de la pile.

5.2 Structure interne de l'interpréteur

L'interpréteur de notre machine virtuelle est l'endroit où le code machine Java devient le langage machine. Il est chargé principalement du décodage des instructions machines et de leur exécution. Toutes les instructions machines Java sont supportées sans exception ; l'interpréteur représente environs 2000 lignes de code C.

5.2.1 Format des instructions

Une instruction Java est constituée d'un code opératoire qui spécifie la valeur sémantique de l'instruction et d'un suite de paramètres [2]. Le code opératoire est codé sur un octet. Ceci permet de définir un jeu de 256 instructions. Les instructions possèdent un nombre variable de paramètres. Certaines sont composées seulement d'un code opératoire et ne possèdent donc pas d'arguments, tandis que d'autres possèdent plusieurs paramètres. Le nombre de paramètres est défini en fonction de l'instruction et peut être représenté par un ou plusieurs octets consécutifs.

Les spécifications Java définissent un jeu d'instructions d'environ 160 instructions. Ceci peut paraître beaucoup mais un bon nombre d'instructions diffèrent seulement par le type des opérandes qu'elles empilent ou dépilent dans la pile des opérandes. Ainsi les instructions `iload` et `fload` prennent toutes les deux la valeur d'une variable locale et l'empilent sur le sommet de pile. La seule différence réside dans le type de la variable qu'elles manipulent[3]. La première `iload` empile un entier et la seconde `fload` empile un flottant.

Les différentes instructions machine peuvent s'organiser en différents groupes que nous décrivons brièvement (pour plus de détails sur les instructions, on se reportera à l'annexe D).

- Les instructions opérant sur les données :
  1. empiler des constantes sur la pile : `bipush`, `ldc`.
  2. manipulation de la pile des opérandes : `pop`, `dup`.
  3. empiler des variables locales sur la pile : `iload`, `fload`.
  4. dépiler des valeurs de la pile pour les stocker dans des variables locales : `istore`, `fstore`.
  5. création de tableaux : `newarray`, `anewarray`.
  6. retrouver les valeurs stockées dans un tableau : `iaload`, `faload`.
  7. ranger des valeurs dans un tableau : `iastore`, `fastore`.
  8. créer des objets : `new`.

[2] Les spécifications Java appellent les paramètres des instructions des opérandes. Nous préférons utiliser le terme de paramètre car le terme d'opérande est aussi utilisé pour faire référence à la pile des opérandes où les instructions vont chercher des valeurs.

[3] On dit du jeu d'instructions qu'il n'est pas orthogonal. Les instructions d'un jeu orthogonal fournies pour un type donné doivent être nécessairement fournies pour les autres types.

9. manipuler les champs des objets : `putstatic`, `getfield`.

– Les instructions arithmétiques est de conversions de type :
  1. opérations arithmétiques : `iadd`, `isub`, `fmul`, `fdiv`.
  2. opérations logiques : `ishl`, `iand`.
  3. opérations de conversions : `i2f`, `i2d`.
– Les instructions contrôlant le flux d'exécution :
  1. branchements conditionnels : `ifeq`, `ifgt`.
  2. opérations de comparaisons : `lcmp`, `dcmpl`.
  3. branchements inconditionnels : `goto`, `jsr`.
  4. table de sauts : `lookupswith`, `tableswitch`.
  5. appels de méthodes : `invokestatic`, `invokevirtual`.
  6. retours de méthodes : `ireturn`, `return`.
– Les autres instructions :
  1. gestion des exceptions : `athrow`.
  2. synchronisation sur les objets : `monitorenter`, `monitorexit`.

A la vue des différents groupes d'instructions, nous pouvons remarquer que le jeux des instructions machine Java ressemble de très près aux jeux des microprocesseurs actuels.

5.2.2 Implémentation de l'interpréteur

Le but de l'interpréteur et de pouvoir exécuter une méthode définie dans une classe Java. C'est le moteur de notre machine virtuelle. De ce fait son organisation et sa structure articule toutes les autres composantes de notre machine.

Nous avons implémenté notre interpréteur par une simple fonction C. Elle comporte une boucle infinie dans laquelle les instructions sont décodées puis exécutées. Cette fonction prend en argument la *frame* de la méthode à exécuter.

Pour toutes les instructions machines, il existe une unique entrée qui représente son implémentation dans notre machine virtuelle. Toutes les instructions soit environ 160 instructions sont implémentées.

La figure 5.2 représente la structure et le fonctionnement de notre interpréteur.

Décodage des instructions

Le décodage des instructions est le mécanisme qui permet à l'interpréteur de retrouver l'instruction qu'il doit exécuter. Chaque instruction possède un

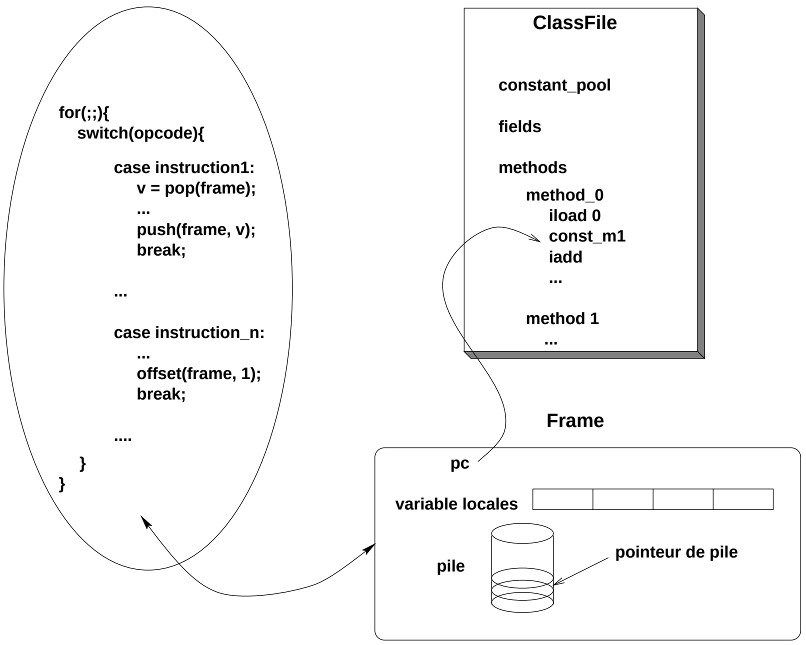

FIG. 5.2: Structure de l'interpréteur

code opératoire qui permet de l'identifier. Le pointeur de programme pointe toujours sur ce code. Ainsi la valeur contenue dans le pointeur de programme permet de placer l'interpréteur sur l'entrée unique de l'instruction dans notre interpréteur.

Le décodage de l'instruction se poursuit par le décodage des paramètres de l'instruction. Le nombre de paramètres est variable et dépend de l'instruction.

Pour réaliser ce décodage, notre interpréteur utilise l'instruction `switch` du langage C. Dans le corps de cette instruction apparaît une entrée par instruction comme le montre la figure 5.2.

Exécution des instructions

L'exécution des instructions intervient immédiatement après le décodage. Elle est bien sûre dépendante de l'instruction. De manière générale l'exécution d'une instruction modifie les données de la machine virtuelle. Une instruction peut modifier la *frame* en touchant par exemple aux variables locales ou bien en modifiant l'état de la pile. D'autres modifications comme la création d'objets ou la modification d'une variable de classe ou d'instance sont aussi possibles.

Après l'exécution de l'instruction courante le pointeur de programme est incrémenté pour pointer sur l'instruction suivante. Les instructions de branchement ne déplacent pas le pointeur de programme sur l'instruction suivante mais sur une autre instruction à l'intérieur de la même méthode. Le déplacement du pointeur de programme permet le bon déroulement de l'exécution de la méthode.

5.2.3 Appels et retours de fonctions

Pour commencer l'exécution d'une méthode, notre machine virtuelle crée une *frame* et la passe à notre interpréteur. La fin de l'exécution d'une méthode est réalisée quand l'interpréteur n'a plus d'instruction à exécuter.

Entre le début et la fin de l'exécution d'une méthode il n'y a pas d'autre exécution de l'interpréteur pour réaliser les appels de méthodes. *L'interpréteur est appelé une et une seule fois pour exécuter la fonction principale d'un programme Java.*

Appels de fonctions

Les appels de fonctions sont réalisés par l'exécution des instructions suivantes :

`invokestatic` Cette instruction lance l'exécution d'une méthode de classe (une méthode déclarée `static`). La référence à la méthode est retrouvée simplement à partir de la structure `ClassFile` décrivant la classe de cette méthode.

`invokevirtual` Cette instruction lance l'exécution d'une méthode appartenant à un objet se trouvant sur la pile des opérandes. La recherche de la méthode est un peu particulière. Elle est décrite complétement dans les sections 5.3 et 5.5.

`invokenonvirtual` Cette instruction lance l'exécution de l'initialisateur ou d'une méthode privée d'un objet se trouvant sur le sommet de la pile des opérandes. Elle fonctionne comme l'instruction `invokevirtual`.

`invokeinterface` Cette instruction lance l'exécution d'une méthode définie par une interface sur l'objet se trouvant sur la pile des opérandes. L'interpréteur doit rechercher dynamiquement la définition de cette méthode.

Après avoir retrouvé la définition de la méthode à exécuter, une *frame* est créé. Les arguments de la méthode se trouvent sur la pile des opérandes. Ils sont dépilés et placés dans l'ordre de dépilement dans les premières variables locales de la nouvelle *frame*.

Pour poursuive l'exécution sur la nouvelle méthode la nouvelle *frame* est placée sur le sommet de la pile de *frames*. Elle devient par conséquent la *frame* courante d'exécution.

Retour de fonctions

La fin d'exécution d'une méthode est réalisée de lors l'exécution de l'une des différentes formes de l'instruction `return`.

L'instruction `return` finit l'exécution de la méthode courante et continue à s'exécuter sur la *frame* de la méthode appelante. Les instructions `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn` se comportent comme l'instruction `return` mis à part qu'elles retournent une valeur à la méthode appelante (respectivement un entier, un long, un flottant, un double, une référence à un objet). Les valeurs renvoyées par ces instructions sont placées sur le sommet de la pile des opérandes de la méthode appelante.

L'interpréteur dépile ensuite la *frame* courante et libère l'espace mémoire qu'elle occupait. Ainsi le contrôle est passé à la *frame* appelante et l'exécution se poursuit sur l'instruction suivant l'appel de la fonction.

5.3 L'édition de liens

L'édition de liens pour notre machine virtuelle est réalisée juste après le chargement d'une classe[4]. Ce mécanisme permet de faire transiter une classe chargée vers un état ou elle pourra être directement utilisée par l'environnement d'exécution. Toutes les classes sans exceptions passent par cette étape.

L'édition de liens est constituée de deux étapes. La première est l'étape de vérification du code contenue dans les méthodes de la classe qui vient d'être chargée. Cette étape de vérification n'est pas encore implémentée dans notre machine virtuelle[5]. La seconde étape consiste à préparer la classe.

L'édition de lien au même titre que le chargement des classes se fait de manière récursive. Toutes les classes parentes d'une classe donnée doivent être auparavant passées par l'édition de liens. La demande d'édition de liens d'une classe nouvellement chargée engendre l'édition de liens pour toutes ces classes parentes si cela n'a pas déjà été fait.

[4]Les spécifications Java définissent le mécanisme de l'édition de liens à partir du chargement des classes alors que nous considérons ici que le réel travail d'édition de liens intervient après le chargement.

[5]La vérification de code a pour but de vérifier si le code des méthodes est syntaxiquement et sémantiquement correct. Dans la machine virtuelle de Sun (qui est notre référence) cette étape de vérification n'est pas effectuée sur les classes qui sont chargées par le chargeur système. Comme notre machine virtuelle ne possède qu'un chargeur de classe système nous n'avons pas resenti le besoin de développer cette partie. Elle est utilisée uniquement pour les classes chargées à travers le réseau.

5.3.1 La préparation de classe

La préparation est une étape qui permet de créer les structures de données spécifiques à la classe qui vont être utilisées par les instructions machines Java. Nous avons vu à la section 4.2 la représentation interne d'une classe [6] en laissant de coté quelques champs utilisés par le moteur d'exécution. Ce sont ces derniers qui sont construits par l'étape de préparation.

Allocation d'espace mémoire pour les variables statiques

Les variables statiques ou les variables de classes sont des variables qui sont partagées par toutes les instances d'une même classe. Lorsqu'une variable de classe est modifiée, cette modification est visible par l'ensemble des objets et des classes qui peuvent accéder à cette variable. Une variable de classe est donc unique dans notre machine virtuelle. Il nous a donc semblé cohérent de stocker les variables de classes dans notre structure `ClassFile` puisque elle aussi représente de manière unique une classe donnée.

Le champ `r_fields` de la structure `ClassFile` est un tableau qui contient toutes les variables statiques d'une classe. L'espace mémoire nécessaire au stockage des variables statiques est calculé en parcourant la table des variables (le champ `fields` de la structure `ClassFile`). Chaque entrée de cette table possédant le bit `ACC_static` positionné est une variable de classe. Les autres sont des variables d'instances et ne sont pas prise en compte.

Résolution des variables statiques

Après l'allocation des variables statiques, elle sont résolues. Le processus de résolution permet d'associer une entrée de la *constant pool*à un élément physique de la machine virtuelle. Ici, pour la résolution des variables statiques, il s'agit de faire correspondre l'entrée de la *constant pool*représentant une variable statique de la classe à son emplacement dans le tableau `r_fields` construit à l'étape précédente [7]

La table d'association est dans un premier temps allouée. Elle contient exactement autant d'entrées qu'il y en a dans la *constant pool.* C'est le champ `r_resolved` de notre structure `ClassFile`.

Pour chaque variable statique, nous cherchons son entrée dans la *constant pool.* Nous rangeons ensuite dans la table `r_resolved` l'adresse de la

[6] la structure `ClassFile`

[7] Le processus de résolution des variables de classes dans l'étape de préparation n'est pas obligatoire. Nous l'avons implémenté ici pour éviter à l'interpréteur de le faire pendant l'exécution des instructions (voir section 5.5). Les variables de classes sont généralement moins nombreuses que les variables d'instances par conséquent leurs résolutions durant l'étape de préparation n'est pas pénalisante.

variable statique se trouvant dans le tableaux `r_fields`. Pour plus de détails sur le mécanisme de résolution on se référera à la section 5.5 page 65.

Création du bloc de méthodes

Lors de la préparation d'une classe notre machine virtuelle analyse les différentes méthodes implémentées par cette classe. Elle construit un tableau regroupant toutes les informations nécessaires au moteur d'exécution pour la création des *frames*.

Chaque élément de ce tableau est une structure `MethodBlock` définie comme suit :

```
struct MethodBlock {
    ClassFile *class;
    char *name;
    int nargs;
    int access_flag;
    Code_attribute *ca;
};
```

Le tableau `r_methods` de notre structure `ClassFile` contient une entrée par méthode implémentée par la classe plus une entrée par méthode dont elle hérite de ces classses parentes.

- Le champ `class` contient une référence à laquelle la méthode appartient.
- Le champ `name` est le nom complet de la méthode tel qu'il est référencé dans la *constant pool*. Il est formé du nom de la méthode et de sa signature.
- Le champ `narg` contient le nombre d'arguments de la méthode.
- Le champ `access_flag` contient les modificateurs d'accès de la méthode.
- Le champ `ca` est une référence à l'attribut de code de la méthode. Nous retrouvons grâce à celui-ci l'adresse de la première instruction à exécuter.

Pour cela, lors de la préparation des méthodes, notre machine virtuelle commence par faire une copie de la table des méthodes de sa classe parente. Elle parcourt ensuite le tableau décrivant les méthodes qu'elle implémente pour y ajouter ces nouvelles méthodes. Si la nouvelle méthode à ajouter est déjà définie, c'est cet emplacement là dans le tableau des méthodes qui est pris. Sinon la nouvelle méthode est ajoutée à la fin de la table. Par ce mécanisme de contruction de la table des méthodes, nous assurons que toutes les méthodes d'une hiérarchie de classe possédant le même nom et la même signature se trouvent au même endroit dans la table des méthodes de chaque classe. Ceci aide au processus de résolution que nous détaillons à la section 5.5.

Nous pourrions penser qu'il n'est pas vraiment nécessaire de dupliquer ces informations dans le tableau `r_methods` puisque nous les trouvons toutes dans la structure `ClassFile`. Nous avons réalisé cela dans le but de créer plus rapidement les *frames* lors de l'appel d'une nouvelle fonction.

Création de la classe `Class`

Enfin la dernière chose que réalise la préparation est la création de la classe `java.lang.Class` associée à la classe en cours de préparation. En effet, les spécifications Java indiquent que pour chaque classe chargée par la machine virtuelle doit lui être associée une instance de la classe `java.lang.Class` la représentant[8].

Le champ `r_java_class` de notre structure `ClassFile` contient cette référence. C'est en fait un objet de type `java.lang.Class` qui contient dans sa première variable d'instance une référence sur la structure `ClassFile` en cours de préparation.

5.4 L'initialisation des classes

Après avoir été préparée une classe doit être initialisée. C'est une étape obligatoire et toutes les classes venant d'être chargées et préparées sont initialisées par notre machine virtuelle. L'initialisation d'une classe consiste à exécuter l'initialisateur statique de la classe. C'est une méthode directement construite par le compilateur et qui porte le nom réservé `<clinit>`. Cette méthode à pour but l'affectation des variables statiques de la classe.

Notre machine virtuelle pour réaliser l'initialisation d'une classe fraîchement préparée construit la *frame* d'exécution de la méthode `<clinit>`. Cette nouvelle *frame* est ensuite envoyée à l'interpréteur pour exécution. Lorsque l'interpréteur a fini son travail toutes les variables statiques de la méthode sont initialisées.

L'initialisation d'une méthode n'est autorisée que si toutes ses classes parentes sont elles aussi initialisées. Pour cela l'initialisation des classes est réalisée de manière récursive par notre machine virtuelle.

Un problème apparaît quand à l'utilisation de la récursion pour l'initialisation des classes. Prenons le code Java suivant comme exemple :

```
public class Truc {
```

[8]Cette classe de l'API Java permet en autre de retrouver à l'exécution le nom de la classe d'un objet et de créer des instances de classe à partir d'un nom de classe. Son utilisation reste tout de même marginale

```
        static Truc t1 = new Truc(1):
        int valeur;

        Truc(int i){ this.valeur = i };
}
```

La classe `Truc` après le chargement est préparée et initialisée. L'initialisation dans cet exemple consiste à affecter à la variable statique `t1` un objet de la même classe `Truc`. L'interpréteur lors de l'exécution de l'initialisateur va devoir créer un objet de type `Truc` dont la classe n'est pas encore initialisée. Il est normalement impossible à notre machine virtuelle d'utiliser une classe qui n'est pas initialisée.

Or, le code Java de notre exemple est tout a fait valide et s'exécute parfaitement bien sur la machine virtuelle de Sun.

Pour cela nous avons dû ajouter ce cas d'utilisation à notre procédure d'initialisation. A cet effet la structure `ClassFile` contient un champ de bits `r_state` qui indique dans quel état est la classe. Normalement, pour qu'une classe soit utilisable par l'environnement d'exécution les bits `CF_LINKED` et `CF_INITIALIZED` doivent être positionnés. `CF_LINKED` indiquant que la classe a été préparée et `CF_INITIALIZED` indiquant que la classe a été initialisée. Pour résoudre le problème que nous venons d'évoquer, nous avons ajouté le bit `CF_BEING_INITIALIZED` qui indique le cas où la classe est en cours d'initialisation. Quand ce bit est positionné la machine virtuelle s'autorise le droit d'utiliser la classe même si elle n'est pas complètement initialisée.

C'est le seul cas où nous autorisons une telle pratique à notre machine virtuelle.

5.5 Résolution dynamique à l'exécution

La résolution consiste à déterminer les valeurs concrètes des valeurs symboliques contenues dans la *constant pool*. Le code contenu dans les méthodes d'une classe fait en permanence référence à la *constant pool*. Les opérandes des instructions sont souvent des index dans la table de symboles. Pour cela notre machine est dotée d'un mécanisme de résolution de ces différentes références symboliques.

Par exemple l'instruction

```
getfield #7
```

place sur le sommet de la pile des opérandes le contenu de la variable d'instance dont la définition[9] est donnée par la septième entrée de la *constant pool*. Cette entrée doit bien sur définir une variable.

[9] On entend par définition le nom de la classe à laquelle elle appartient, son nom et sa signature.

La résolution d'une entrée n'est effectuée qu'une seule fois. Lorsque la machine virtuelle effectue une résolution, elle la sauvegarde pour pouvoir la réutiliser plus tard. Une résolution se réalisant dans le corps d'une boucle a beaucoup de chance d'être réutilisée. Ce mécanisme permet de gagner du temps à l'exécution.

Les valeurs déterminées à la résolution sont sauvegardées dans le tableau `r_resolved` de la structure `ClassFile`. Ce tableau est alloué à la préparation de la classe et possède le même nombre d'entrées que la *constant pool*. Ainsi après la résolution de la $n^{ième}$ entrée de la *constant pool*, l'entrée n du tableau `r_resolved` contiendra le résultat de la résolution. Si par la suite une autre instruction référence la $n^{ième}$ entrée de la *constant pool*notre machine virtuelle ne résoudra pas cette entrée et utilisera la $n^{ième}$ entrée du tableau `r_resolved`.

La résolution dépend du type de l'entrée située dans la *constant pool*. La résolution pour une classe ne va pas engendrer le même travail que pour une l'entrée représentant une méthode. Les différentes entrées que doit pouvoir résoudre notre machine virtuelle sont :

1. les entrées de type `CONSTANT_Class` qui représentent une classe Java,
2. les entrées de type `CONSTANT_Fieldref` pour les variables de classes et d'instances,
3. les entrées de type `CONSTANT_Methodref` pour les méthodes,
4. les entrées de type `CONSTANT_String` pour les chaînes.

5.5.1 Résolution des classes

La résolution d'une classe revient à retrouver la structure `ClassFile` de cette classe. Si la classe a déjà été chargée par notre machine virtuelle alors sa référence sera sauvegardée dans le tableau `r_resolved` directement. Dans le cas contraire la machine virtuelle demande au chargeur de classe de la charger. Elle est ensuite préparée et initialisée.

La demande de résolution d'une classe peut donc entraîner le chargement et l'initialisation de plusieurs autres classes.

5.5.2 Résolution des variables

La résolution des variables est divisée en deux parties. La première résout les variables de classes, la seconde résout les variables d'instances. La résolution des variables de classe est déclenchée par les instructions `getstatic` et `putstatic`. La résolution des variables d'instances est déclenchée par les instructions `getfield` et `putfield`.

variables statiques

Pour résoudre l'entrée d'une variable statique notre machine virtuelle demande la résolution de la classe à laquelle elle appartient. Elle récupère ainsi une référence sur sa structure `ClassFile`.

Ensuite notre machine virtuelle récupère la structure de type `name_and_type` de la *constant pool*qui donne le nom et la signature de la variable. Elle détermine ensuite son emplacement dans tableau `r_fields` en parcourant toutes les variables statiques déclarées par la classe.

La référence directe de la variable statique (son adresse dans le tableau `r_fields`) est sauvegardée dans le tableau `r_resolved`[10].

variables d'instances

La résolution des variables d'instances est similaire dans son principe à la résolution des variables statiques mis à part qu'elle doit s'effectuer à partir d'un objet. Un objet Java contient en plus des variables déclarées par sa classe toutes les variables dont il hérite de ces classes parentes. La résolution consiste donc à retrouver un déplacement dans le tableau qui contient toutes ces variables d'instances.

La recherche de la variable d'instance commence sur la classe à laquelle l'objet appartient. Si la variable ne se trouve pas déclarée dans cette première classe, la recherche se poursuit sur sa classe parente. De même si cette classe ne contient pas la déclaration de la variable recherchée, la recherche se poursuit sur la classe parente de cette dernière. Le processus de recherche continue ainsi récursivement de classe parente en classe parente. Au fûr et a mesure de la progression, un compteur est incrémenté indiquant l'emplacement de la variable dans l'objet. C'est ce compteur qui est sauvegardé dans le tableau `r_resolved` quand la variable est trouvée.

5.5.3 Résolution des méthodes

La résolution des méthodes consiste a retrouver l'emplacement de la méthode dans le bloc des méthodes construit lors de la préparation (le champ `r_method` de la structure `ClassFile`). Les instructions d'appels de méthode génèrent ce type de résolution.

[10] Les variables statiques d'une classe sont résolues lors de la préparation de la classe. Le mécanisme de résolution que nous venons de décrire n'est donc utilisé que pour les variables statiques n'appartenant pas à la classe courante.

Résolution des méthodes statiques

Pour la résolution des méthodes statiques notre machine virtuelle récupère de la *constant pool*le nom et la signature de méthode ainsi que le nom de la classe où elle est définie. Elle détermine ensuite son emplacement dans la tables des méthodes en parcourant le champ `r_methods` de sa classe. La référence directe à la méthode (son adresse dans le tableau `r_methods`) est sauvegardée dans le tableau `r_resolved`.

Résolution dynamique méthodes

La résolution dynamique des méthodes est réalisée pour toutes les méthodes qui ne sont pas déclarées statiques. Elle correspond à la résolution des méthodes qui sont appelées sur des objets Java.

La résolution dynamique se réalise de la même manière que la résolution des méthodes statiques. La différence entre ces deux méthodes de résolution se situe dans les valeurs sauvegardées dans le tableau `r_resolved`. La résolution dynamique sauvegarde l'emplacement de la méthode dans le tableau `r_methods` aux lieu de sauvegarder son addresse. Cela permet à l'instruction ayant demandé la résolution de retrouver la méthode à partir l'objet et non pas à partir du nom de la classe générée par le compilateur dans la table des symboles.

Ceci est dû à la façon dont le tableau des méthodes a été construit durant l'étape de préparation. Toutes les méthodes déclarées par une hiérarchie de classe possèdent le même emplacement quelles que soit leurs positions dans la hiérachie.

Nous avons du créer ce système de préparation et de résolution car le compilateur ne peut pas générer une référence correcte sur l'exécution d'une méthode dépendant d'un objet créé à l'exécution.

5.5.4 Résolution des chaînes

La résolution des chaînes consiste à construire une chaîne Java à partir de l'entrée de la *constant pool*. L'entrée de la *constant pool*contient les caractères formant le corps de la chaîne. Ils sont utilisés pour construire un objet de type `java.lang.String` (voir 5.6). La référence sur l'objet nouvellement construit est sauvegardé dans le tableau `r_resolved`.

5.6 Les objets

Toutes les données manipulées (mis à part les types primitifs) par la machine virtuelle Java sont des objets. Un objet peut donc être

1. une instance d'une classe particulière,
2. un tableau contenant des types primitifs et des instances de classes,
3. une chaîne de caractères.

Les objets ont une durée de vie dans la machine virtuelle. Ils commencent leur existence lors de leur création et la termine lorsqu'ils ne sont plus utilisés par la machine virtuelle. En fait il n'existe pas de moyen pour le programmeur Java de détruire explicitement un objet qui n'est plus utilisé[11]. La destruction des objets est prise en charge par la machine virtuelle. Ce mécanisme, connu sous le nom de *garbage collecting*, [12] n'est pas encore implémenté dans notre machine virtuelle.

5.6.1 Structure générique d'un objet

La définition des objets par notre machine virtuelle est unique. Tous les objets sont représentés par une structure unique quel que soit le type d'objets.

La représentation d'un objet est la suivante :

```
struct Object {
    ClassFile *class;
    int type;
    int *vars;
    int vars_size;
};
```

Pour différencier les différents types d'objets, nous utilisons les champs `class` et `type`. Ces deux champs vont, par exemple, permettre de différencier les objets de type tableau et de type chaîne (voir les sections suivantes pour une description détaillée). Les valeurs du champ `type` sont données par le tableau 5.1.

Dans les langages objets, les instances de classes héritent de toutes les variables des classes parentes et Java n'echappe pas à cette règle. Par exemple dans le code Java suivant :

```
public class Foo {
    int x; double d;
}

public class FooPlus extends Foo {
    float f;
```

[11] Il n'existe pas de fonction `free` ou `delete` comme en C ou C++ dans le langage Java.

[12] Cette expression vient du mot *garbage collector* que l'on peut taduire par ramasse-miettes. Un *garbage collector* est un mecanisme indépendant qui permet de retrouver les objets en mémoire qui ne sont plus référencés et donc inutilisés afin de libérer l'espace mémoire qu'ils occupent.

Type	Valeur	Description
`T_CLASS`	0	objet
`T_TAB`	1	tableau
`T_STRING`	2	objet chaîne
`T_VOID`	3	pas utilisé pour les objets
`T_BOOLEAN`	4	tableau de `boolean`
`T_CHAR`	5	tableau de `char`
`T_FLOAT`	6	tableau de `float`
`T_DOUBLE`	7	tableau de `double`
`T_BYTE`	8	tableau de `byte`
`T_SHORT`	9	tableau de `short`
`T_INT`	10	tableau de `int`
`T_LONG`	11	tableau de `long`

TAB. 5.1: Les différents types d'objets

```
}
```

la classe `FooPlus` hérite des variables de la classe `Foo`. Cela signifie que toutes les méthodes de la classe `FooPlus` pourront utiliser les variables `x` et `d` comme si elles avaient été déclarées à l'intérieur de la classe `FooPlus`.

Pour implémenter ce mécanisme notre machine virtuelle construit un tableau contenant toutes les variables d'instances que la classe déclare et y ajoute toutes celles dont elle hérite des classes parentes. Ces variables d'instances sont créées en parcourant toutes les classes parentes de manière récursive. Le tableau `vars` de notre structure `Object` contient ces variables et possède une taille donnée par le champ `vars_size`.

5.6.2 Les instances de classe

Les objets représentant de simples instances de classes sont créés par l'exécution de l'instruction `new`. Cette instruction possède en paramètre l'entrée dans la *constant pool* de la classe de l'objet à créer. Notre machine résout cette entrée et récupère une référence sur la structure `ClassFile` de cette classe.

Une structure `Object` est créée. Le champ `class` de cette structure contient la référence de la classe qui vient d'être résolue et le champ `type` contient la valeur `T_CLASS` du tableau 5.1.

La classe de l'objet ainsi que ces classes parentes sont parcourues afin de déterminer la taille du tableau `vars` qui contiendra toutes les variables. Le tableau `vars` est ensuite alloué et initialisé a zéro.

5.6.3 Les tableaux

Il existe trois types de tableaux en Java : les tableaux contenant des types primitifs, des tableaux contenant des instances de classes, des tableaux à plusieurs dimensions.

Les tableaux de types primitifs

Ce sont des tableaux contenant des valeurs numériques et des valeurs booléennes. Ils sont créés par l'exécution de l'instruction `newarray`. Cette instruction possède en paramètre le type du tableau. Notre machine virtuelle récupère sur la pile des opérandes le nombre d'éléments que doit contenir le nouveau tableau.

Une structure `Object` est créée et le champ `class` est initialisé à zéro. La valeur du champ `type` détermine le type des variables. C'est une des valeurs du tableau 5.1.

Les tableaux d'objets

Ce sont des tableaux contenant un ensemble d'instances de classes. Ils sont créés par l'exécution de l'instruction `anewarray`. Cette instruction possède en paramètre un indice dans la *constant pool*donnant la classe des objets et cette entrée est résolue.

Ensuite, la structure `Object` de ce tableau est créée. Le champ `class` contient la référence à la classe décrivant les objets du tableau. Le champ `type` contient la valeur `T_TAB`. Le nombre d'éléments du tableau est dépilé du sommet de la pile des opérandes et permet l'allocation du tableau `vars`.

Les tableaux à plusieurs dimensions

Ce type de tableaux est créé par l'instruction `amultianewarray`. Pour représenter les tableaux à plusieurs dimensions notre machine utilise des tableaux de tableaux. Ainsi pour créer le tableau à deux dimensions suivant

```
int[][] t = new int[2][3]
```

notre machine va créer dans un premier temps un tableau contenant la référence à deux autres tableaux. Les deux références sur des tableaux vont être initialisés par la création de deux tableaux de type primitif `int` et de taille trois (voir figure 5.3). Le même principe est utilisé pour la représentation des tableaux de plus de deux dimensions. Il y a $n-1$ niveaux d'indirections pour représenter un tableaux à n dimensions.

L'instruction `multianewarray` contient en paramètre le type du tableau. Il peut s'agir aussi d'une référence sur une classe. La dimension du tableau se

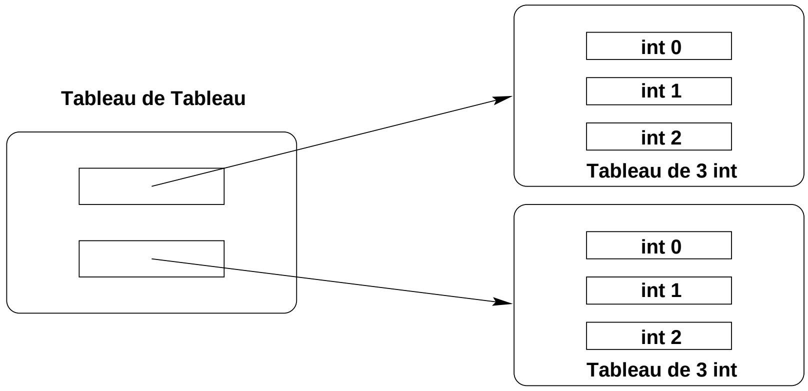

FIG. 5.3: Représentation des tableaux à plusieurs dimensions

trouve en deuxième paramètre de l'instruction `multianewarray` et permet de retrouver la taille de chacune des dimensions sur la pile des opérandes.

5.6.4 Les chaînes de caractères

Les chaînes de caractères en Java sont des objets à part entière. Ils sont des instances de la classe `java.lang.String`. Ces objets sont des objets immutables : après la création d'un objet de type `java.lang.String` la chaîne de caractères définissant l'objet ne peut être modifier[13].

Lorsque le programmeur Java crée dynamiquement une chaîne de caractères, notre machine utilise le même mécanisme que celui que nous avons décrit pour les instances de classe. C'est l'instruction `new` qui est utilisée.

Lorsque la machine résout une entrée de la *constant pool*qui est une référence sur une chaîne (une entrée de type `CONSTANT_String`), notre machine doit créer un objet de type `java.lang.String` et l'initialiser correctement avec la chaîne de caractère définie par l'entrée de la *constant pool*. Pour cela elle construit dans un premier temps un tableau de caractères Java contenant la chaîne et appelle elle même le constructeur `String([C)V`. Une fois ce constructeur exécuté, l'objet de type string est prêt à être utilisé par notre machine virtuelle.

[13]Pour modifier le contenu d'une chaîne de caractère les API Java définissent une classe de type `java.lang.StringBuffer` qui permet de les modifiée. Les primitives de manipulation comme la concaténation et le troncage sont implémentées par cette classe.

5.7 Démarrage de la Machine Virtuelle

Notre machine virtuelle est conçue pour exécuter des applications Java. Elle s'exécute sur le système Plan 9 comme une application traditionnelle et elle se lance à partir de la ligne de commande du shell `rc`. Par exemple

```
term% run circle -radius 10
```

provoque l'exécution de l'application `circle`. Les valeurs `-radius` et `10` forment les arguments de l'application `circle`.

Avant de pouvoir commencer l'exécution de l'application Java notre machine virtuelle exécute un certain nombre d'opérations qui permettent de mettre en place l'environnement d'exécution.

La première chose que réalise notre machine est la recherche et l'analyse des options de la ligne de commande. Ces options sont placées juste après le nom de la commande qui lance la machine et juste avant le nom de l'application qu'elle doit lancer. Seules trois options sont reconnues :

- l'option `-version` donne le numéro de la version de la machine virtuelle. Lorsque cette option est présente sur la ligne de commande la machine virtuelle affiche sur la sortie standard le numéro de version de la machine virtuelle et redonne immédiatement la main au shell. A ce jour le numéro de version vaut `0.1a`.
- l'option `-classpath` qui doit être suivit d'une liste de répertoires qui permet d'indiquer les répertoires où doivent se trouver les classes Java qu'elle va exécuter (voir la section 4.2.2).
- l'option `-stacksize` est suivie d'un entier indiquant la taille maximum en octets de la pile de *frames*.

Si toutefois les arguments de la machine virtuelle n'étaient pas corrects, la machine virtuelle afficherait un message d'erreur et terminerait son exécution.

Après la vérification des arguments de la ligne de commande, la machine virtuelle crée les listes de recherches des répertoires et les tables de recherches des archives au format `zip` (voir chapitre 4).

Après ces étapes préliminaires notre machine virtuelle Java commence réellement à préparer l'exécution de l'application. Les étapes de cette préparation sont les suivantes :

1. Le chargement, l'édition de liens et l'initialisation de la classe `Class` est réalisée pour vérifier que notre machine virtuelle est bien capable de retrouver les classes des API. En effet, si cette dernière classe n'est pas trouvée, c'est que la liste des répertoires et des archives n'est pas valide. La machine s'arrête immédiatement en affichant un message d'erreur.

2. La pile de *frames* est créée et initialisée.
3. La classe de l'application est chargée et passe par l'édition de liens et l'initialisation.
4. La création de la première *frame* d'exécution est réalisée. Pour cela notre machine virtuelle cherche la méthode `main` possédant la signature `([Ljava/lang/String;)V`. C'est cette méthode qui constitue le point d'entrée d'une application Java.
5. Les arguments qui se trouvent sur la ligne de commande de l'application sont fabriqués. Notre machine construit un objet de type chaîne pour chaque argument et le place dans un tableau de chaîne. C'est ce dernier tableau qui est passé en argument à la méthode `main` de l'application. Si l'application ne possède pas d'argument un tableau qui ne comprend aucune chaîne est passé à la méthode `main`.

L'exécution de la *frame* de la fonction `main` peut alors commencer.

5.8 Exemples d'implémentation de quelques instructions

Nous présentons dans cette section l'implémentation détaillée de trois instructions de notre interpréteur : l'instruction `ifeq` qui est une instruction de branchement, l'instruction `iastore` qui permet de ranger une valeur dans un tableau d'entiers et l'instruction `invokestatic` qui réalise l'appel d'une méthode de classe.

L'instruction `ifeq`

L'instruction `ifeq` est une instruction de branchement générée par le compilateur. Elle teste si la valeur entière qui se trouve sur le sommet de la pile des opérandes est nulle.

Le format de cette instruction est le suivant :

```
_ifeq    branch1    branch2
```

Elle est composée de trois octets : le premier indique le code opératoire et vaut 153, les deux derniers sont utilisés pour composer une valeur sur 16 bits `(branch1 << 8) | branch2` qui déplace le pointeur de programme si la valeur testée est nulle.

Dans notre interpréteur, l'instruction est représentée par le code C suivant :

```
case _ifeq: {
    int v = pop(f);
    if( !v )
        offset(f, pc2signedshort(pc));
```

```
        else
            offset(f, 3);
        break;
    }
```

La valeur à tester se trouve sur le sommet de la pile des opérandes. La macro `pop` permet de récupérer cette valeur et la place dans la variable `v`.

L'interpréteur test si la variable `v` est égale à zéro. Si c'est le cas, le pointer de programme doit être déplacé à l'adresse de branchement fournie par l'instruction `ifeq`. La macro `pc2signedshort` construit cette valeur sur 16 bits. La macro `offset` additionne cette dernière au pointeur de programme.

Si la valeur `v` testée est différente de zéro, le pointeur de programme est déplacé de trois octets qui correspondent à la taille de l'instruction.

L'instruction `iastore`

L'instruction `iastore` permet de ranger une valeur dans un tableau d'entiers.

Le code C qui suit, représente l'implémentation de l'instruction dans notre interpréteur :

```
case _iastore: {
    int v = pop(f);
    Object *array;

    index = pop(f);
    array = (Object *)pop(f);
    if( !array ){
        throwException(f, "java/lang/NullPointerException");
        break;
    }
    if( index >= (array->vars_size / sizeof(int)) ){
        throwException(f, "java/lang/ArrayIndexOutOfBoundsException");
        break;
    }
   *(array->vars + index) = v;
   offset(f, 1);
   break;
}
```

Avant l'exécution de l'instruction, la pile des oprérandes est constituée de :

```
..., ref_tableau, index, valeur
```

`valeur` l'entier à stocker dans le tableau `ref_tableau` à la position `index`.

Dans le code qui représente l'implémentation de l'instruction les variables `v`, `index` et `array` sont initialisées en récupérant les valeurs qui se trouvent sur la sommet de la pile des opérandes (respectivement, l'entier à stocker, l'index dans le tableau et le tableau).

Si la référence sur le tableau est nulle, l'interpréteur lance une exception de type `NullPointerExcepion` et si l'index dans le tableau est plus grand que la taille du tableau une exception de type `ArrayIndexOutOfBoundsException`. Dans les deux cas précédents l'instruction se termine immédiatement. Les exceptions sont traitées dans le chapitre 6.

Les tableaux sont de objets et les éléments du tableau sont accéssibles par le champ `vars` de la structure `Object`. L'interpréteur affecte donc la valeur `v` dans le tableau (la ligne `*(array->vars + index) = v`).

Le pointeur de programme est ensuite déplacé sur l'instruction suivante à l'aide de la macro `offset` et l'exécution continue.

L'instruction `invokesatic`

L'instruction `invokesatic` est générée par le compilateur Java pour réaliser l'appel à une fonction de classe.

L'implémentation de cette instruction est présentée par le code C suivant :

```
case _invokestatic:{
    Frame *new_frame;
    MethodBlock *mb;
    int i;

    index = (f->pc[1]<<8) | f->pc[2];

    if( !f->class->r_resolved[index] ){
        cp_resolution(f->class, index, 0);
        TEST_EXCEPTION();
    }

    mb = (MethodBlock *)f->class->r_resolved[index];

    if( mb->access_flag&ACC_ABSTRACT ){ /* si methode abstraite */
        throwException(f, "java/lang/AbstractMethodError");
        break;
    }

    if( mb->access_flag&ACC_NATIVE ){
        nativeCall(mb, 3, 1);
        break;
    }

    offset(f, 3);
    new_frame =  newFrame(mb, f);
    if( !new_frame )
```

```
        panic("can't create frame Invokevirtual()");

    for(i=0 ; i < mb->nargs ; i++)
        setlocal(new_frame, mb->nargs-i-1, pop(f->invoker));

    break;
}
```

L'instruction contient dans sa définition la référence dans la *constant pool* de la méthode à appeler. L'interpréteur place cet index dans la *constant pool* dans la variable `index`.

Cette entrée doit être résolue. Pour cela, l'interpréteur demande la résolution en appelant la fonction `cp_resolution`. Après la résolution de cette entrée, la variable `r_resolved` de la *constant pool* contient la référence sur la structure `MethodBlock` qui définie cette méthode.

Avant de préparer la *frame* d'exécution de cette méthode, l'interpréteur vérifie que la méthode n'est pas déclarée abstraite. Une méthode abstraite par définition ne contient pas de code et doit être implémentée par les classes dérivées. Si elle est déclarée abstraite la machine virtuelle ne peut pas l'appeler et lance une exception (voir le chapitre 6).

Si la méthode est déclarée *native*, elle représente une fonction de la librairie Java écrite en langage C. La fonction `nativeCall` réalise l'appel de la méthode. On se reportera au chapitre 8 pour plus d'information.

L'interpréteur construit ensuite la *frame* d'exécution et l'empile sur la pile de *frames* (la fonction `newFrame()`). Il dépile ensuite de l'ancienne pile des opérandes les arguments de la méthode et les place dans les variables locales de la nouvelle *frame* au moyen de la macro `setlocal`. L'instruction `invokestatic` se termine et l'interpréteur exécute la nouvelle méthode.

Chapitre 6

La gestion des exceptions et des erreurs Java

Le langage Java dispose d'un mécanisme d'exception pour gérer les erreurs et les événements exceptionnels pouvant survenir durant l'exécution des programmes. Notre machine virtuelle implémente le lancement et l'interception des exceptions ainsi que deux fonctions définies dans les librairies Java qui permettent au développeur de retrouver la trace des exceptions.

Nous commençons par décrire le concept des exceptions en Java. Pour continuer, nous montrons en détail comment notre machine virtuelle lance et intercepte les exceptions. Nous terminons par un détail de l'implémentation des fonctions de la librairie associées aux exceptions et un exemple d'utilisation des exceptions en Java.

6.1 Le concept d'exception en Java

Java introduit le concept d'exception pour gérer les erreurs et les événements exceptionnels qui peuvent intervenir durant l'exécution de programmes.

Lancer une exception en Java s'effectue à l'aide du mot réservé `throw`. On l'utilise pour signaler que quelque chose d'exceptionnel s'est produit. Capturer une exception se réalise à l'aide du mot réservé `catch`. Il sert à signaler que l'on va attraper une exception et faire tout ce qu'il faut pour gérer la crise.

Les exceptions se propagent en remontant les structures lexicales des méthodes puis la pile d'appel des méthodes. Si une exception n'est pas capturée par le bloc de code qui la lancé l'exception se propage vers le bloc directement englobant. Si ce dernier ne la capture pas, on réapplique le même procédé. Si, vraiment, aucun bloc englobant de la méthode ne veut la capturer, elle se propage vers la méthode appelante où elle continue à se propager. Si une exception n'est jamais capturée, elle se propage jusqu'à la méthode `main` où tout a commencé ; la machine virtuelle affiche un message d'erreur, une trace de la pile d'appel, et s'arrête.

Une exception, en Java, est un objet qui est une instance d'une sous-classe `java.lang.Throwable`. Les librairies Java définissent deux sous-classes `java.lang.Error` et `java.lang.Exception` qui scindent en deux ensembles les exceptions Java. Les exceptions descendant de la classe `java.lang.Error` représentent la gestion des erreurs qui interviennent durant la phase de chargement, l'édition de liens ou lors de l'épuisement des resources système comme la mémoire. Ces exceptions sont considérées comme irrécupérables et sont produites par la machine virtuelle. Les exceptions descendant de la classe `java.lang.Exception` sont générées par les librairies ou construites par l'utilisateur. Elles indiquent l'apparition d'événements exceptionnels qui doivent être capturés.

Les exceptions étant des objets, elles peuvent contenir des données et définir des méthodes. La classe `java.lang.Throwable`, au sommet de la hiérarchie des exceptions, définit une chaîne de type `java.lang.String` qui est héritée par toutes les classes d'exceptions. Cette chaîne est utilisée pour stocker un message lisible décrivant l'événement exceptionnel. Elle est positionnée lors de la création de l'objet exception en la passant au constructeur. Ce message peut être récupéré à l'aide de la méthode `getMessage()` hérité de la classe `java.lang.Throwable`.

Le mécanisme de gestion des exceptions Java est contenu dans les mots réservés du langage `try`, `catch` et `finally`. Le mot réservé `try` englobe un groupe lexical d'instructions Java susceptibles de lancer une exception. Le bloc `try` est nécessairement suivi de un ou plusieurs blocs de `catch` traitant les exceptions d'un type donné. L'exemple suivant montre la structure syntaxique des blocs `try` et `catch`.

```
try{
     // code Java
}catch(ArrayIndexOutOfBoundsException e){
    // gestion des exceptions de débordement de tableaux
}catch(ArithmeticException e)
   // gestion des exceptions sur des opérations arithmétiques
}
```

Dans le pseudo-code précédent deux types d'exceptions sont traités : les exceptions indiquant un débordement de tableau et les erreurs survenues lors d'opérations arithmétiques comme une division par zéro. Si une exception de ce type est lancée le code contenu dans le bloc `catch` associé à l'exception est exécuté. Dans le cas où l'exception n'est pas attrapée, l'exception se propage sur la méthode appelante.

Les clauses `catch` peuvent être suivies d'un bloc `finally` qui contient du code de *nettoyage*. Les instructions du bloc `finally` s'exécutent toujours, indépendamment de la façon dont on est sorti du bloc `try`[1]. La clause `finally` est généralement utilisée pour fermer les fichiers ou libérer des ressources.

6.2 Lancement des exceptions

Notre machine virtuelle implémente deux façons de lancer des exceptions. La première se charge de lancer toutes les exceptions qui sont générées par la machine elle-même, tandis que la seconde lance des exceptions générées par le code contenu dans les méthodes Java ; par exemple une exception n'existant pas dans les librairies Java peut être définie par le développeur pour son usage personnel.

6.2.1 Exceptions lancées par la machine virtuelle

Les exceptions lancées par la machine virtuelle regroupent toutes les exceptions qui peuvent être générées lors des phases de chargement, d'édition de liens, de résolution, durant l'exécution de certaines instructions ou quand les ressources système sont épuisées. Les principales exceptions lancées par notre machine virtuelle sont regroupées dans le tableau 6.1.

Type d'exception	**Evennement**
`NullPointerException`	référence sur un objet erronée
`ArrayIndexOutOfBoundsException`	débordement de tableau
`ArrayStoreException`	affectation en dehors d'un tableau
`ArithmeticException`	erreur de calcul (division par zéro)
`AbstractMethodError`	exécution d'une méthode abstraite
`NegativeArraySizeException`	dimension d'un tableau négative
`ClassCastException`	générée par l'instruction `checkcast`
`NoClassDefFoundError`	chargement d'une classe impossible
`NoSuchFieldError`	variable non trouvée
`IOException`	erreur d'une opération sur un fichier
`ClassNotFoundException`	classe non trouvée

TAB. 6.1: Exceptions lancées par notre machine virtuelle

Lorsqu'un événement exceptionnel intervient notre machine virtuelle doit créer un objet représentant le type de l'exception à lancer. Pour cela elle effectue le chargement et l'édition de liens de la classe de l'exception. Elle crée

[1]Les exceptions ne sont pas le seul moyen de sortir du bloc `try`. Les instructions `break`, `continue` et `return` permettent aussi de quitter un bloc `try` ; le bloc d'instructions de la clause `finally` est tout de même exécuté.

ensuite un objet représentant une instance de cette classe et l'initialise en appelant le constructeur `<init>()V`.

Pour que l'exception soit prise en compte par notre machine virtuelle, l'objet nouvellement créé est mis à la place du sommet de pile, écrasant ainsi ce dernier. Il n'est pas très important de perdre le sommet de pile puisque lorsqu'une exception se produit la méthode courante interrompt son exécution pour essayer d'attraper l'exception. De plus l'objet est stocké dans la *frame* d'exécution dans un pointeur qui vaut toujours zéro tant qu'une exception n'a pas été lancée. Ce mécanisme permet d'attraper et de gérer l'exception.

6.2.2 Exceptions lancées par un programme Java

Lancer une exception à partir d'un programme Java se fait à l'aide de l'instruction `throw`. Elle permet au developpeur de lancer des exceptions prédéfinies par les librairies ou de lancer ses propres exceptions qu'il aura redéfinies lui-même. La syntaxe employée pour lancer une exception serait

```
throw new ArithmeticException
```

Nous retrouvons ici le même principe que nous avons vu lorsque la machine virtuelle lance une exception. Le développeur doit avant tout créer un objet représentant l'exception qu'il désire lancer afin de le placer sur le sommet de la pile des opérandes.

L'instruction machine `athrow` est ensuite exécutée. C'est cette instruction qui retrouve le bloc d'instructions à exécuter pour gérer l'exception.

6.3 Interception des exceptions

Nous venons de voir dans la section précédente la façon dont les exceptions sont lancées. Dans le cas d'une exception lancée par la machine virtuelle la *frame* d'exécution contient un pointeur sur l'objet représentant l'exception ainsi que le sommet de la pile des opérandes. Dans le cas d'une exception lancée par logiciel, le sommet de la pile des opérandes contient l'objet représentant l'exception et le pointeur de programme pointe sur l'instruction `athrow`.

6.3.1 Détection des exceptions

Afin de faire réagir notre machine virtuelle de la même façon au deux lancements possibles d'une exception, nous avons ajouté dans l'interpréteur un test à chaque fin d'exécution d'une instruction. Ce test permet de détecter si la dernière instruction a déclenchée une exception interne. Si la dernière instruction déclenche une exception, la *frame* d'exécution contient une référence sur l'exception qui vient d'être lancée. Dans ce cas l'interpréteur détourne l'exécution normale de la méthode courante pour exécuter l'instruction machine `athrow`.

Par ce mécanisme, le lancement d'une exception interne par la machine virtuelle et le lancement d'une exception logicielle sont identiques et revient à exécuter l'instruction machine `athrow`, sachant que le sommet de pile contient une référence sur l'objet représentant l'exception qui vient d'être lancée.

6.3.2 Exécution de l'instruction `athrow`

L'exécution de l'instruction `athrow` permet à notre machine virtuelle de rechercher si la méthode courante intercepte l'exception se trouvant sur le sommet de la pile des opérandes. Pour cela nous retrouvons la table des exceptions construite par le compilateur dans l'attribut de code de la méthode (voir section 4.2). Le tableau 6.2 montre un exemple d'une table d'exception.

de	**à**	**cible**	**type**
0	19	20	`java.lang.ArithmeticException`
0	30	30	`java.lang.ArrayIndexOutOfBoundsException`

TAB. 6.2: Exemple d'une table d'exception

Les trois premières colonnes du tableau 6.2 représentent des adresses du pointeur de programme. Chaque ligne de ce tableau représente un type d'exception que peut intercepter la méthode. Les deux premières colonnes indiquent dans quelle plage d'adresses doit se trouver le pointeur de programme afin de pouvoir gérer l'exception déclarée par cette ligne de la table[2]. La colonne *cible* indique à quelle adresse commence la gestion de l'exception. Par exemple, pour la première ligne de la table, le pointeur de programme doit être compris entre l'adresse 0 et 19 afin de pouvoir exécuter le code se trouvant à l'adresse 20. La gestion est effective si et seulement si l'exception se trouvant sur le sommet de la pile des opérandes est une instance de la classe `ArithmeticException` ou une instance d'une sous-classe de cette dernière.

Lors de l'exécution de l'instruction `athrow` la table des exceptions est balayée afin de trouver la première entrée qui vérifie ses conditions. Si une entrée de cette table est valide alors le pointeur de programme est placé sur l'adresse où doit commencer la gestion de l'exception et l'exécution continue.

Si aucune entrée de la table des exceptions n'est valide, alors la *frame* courante est dépilée, on écrase le sommet de pile avec l'objet de type exception et la nouvelle *frame* d'exécution contient ce même objet. Ainsi après

[2] La plage d'adresses correspond au bloc `try`.

avoir passé le contrôle à la *frame* de la méthode appelante l'interpréteur trouve une exception non attrapée et exécute aussitôt l'instruction `athrow`. Le processus continue en remontant de méthode en méthode jusqu'à ce qu'une méthode gère l'exception ou jusqu'à ce qu'il n'y ai plus aucune méthode sur la pile. Dans ce cas la machine virtuelle affiche un message d'erreur et une trace de toute la pile de *frames* qui vient d'être parcourue.

6.4 Les méthodes natives associées aux exceptions

Les librairies Java définissent deux méthodes particulières qui facilitent la gestion des exceptions. Ces deux méthodes sont déclarées *natives* et ne possèdent pas d'implémentation Java. Elles sont écrites en langage C et sont *linkées* à notre machine virtuelle. L'appel et la définition des méthodes natives sont traités au chapitre 8, nous décrivons dans cette section uniquement l'implémentation de ces deux méthodes.

Les méthodes `fillInStackTrace` et `printStackTrace` appartiennent à la classe `java.lang.Throwable` dont toutes les exceptions héritent. La première méthode consiste à construire un objet représentant la pile de *frames* lors du lancement d'une exception. La seconde méthode affiche sous forme lisible l'empilement des méthodes sur la sortie standard.

6.4.1 Construction de la pile d'appel des méthodes

La méthode `fillInStackTrace` est définie par la classe `Throwable` et est appelée par le constructeur de cette classe. Lorsqu'une exception est lancée, un objet représentant une sous-classe de la classe `Throwable` est créé. Les différents constructeurs de la hiérarchie de classes sont exécutés et par conséquent le constructeur de la classe `Throwable`. Ainsi lors du lancement de toutes exceptions la méthode `fillInStackTrace` est exécutée.

Cette fonction, quand elle s'exécute, parcourt la pile de *frames* et sauvegarde deux types d'informations : la référence sur la méthode de la *frame* qu'elle examine et le numéro de la ligne dans le code source où se trouve le pointeur de programme.

Dans un premier temps la machine virtuelle doit compter le nombre de *frames* présentes sur la pile, pour allouer un tableau qui contiendra la paire : référence à la méthode, numéro de la ligne dans le code source. Lors du calcul du nombre de *frames* présentes sur la pile, notre machine virtuelle doit ignorer les *frames* d'exécution des constructeurs de l'objet exception.

Le tableau devant contenir les informations est alloué et notre machine parcourt la pile de *frames* pour le remplir. Ce tableau est représenté sur la figure 6.1.

La première information que sauvegarde notre machine lorsqu'elle par-

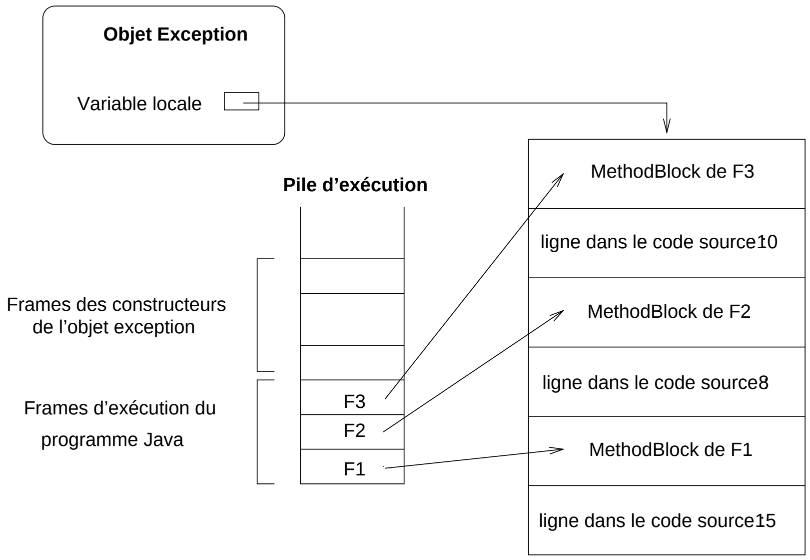

FIG. 6.1: Le tableau sauvegardé lors d'une exception

court la pile est la structure `MethodBlock` (voir la section 5.1) représentant la méthode. Elle permet de récupérer le nom de la méthode ainsi que le nom de la classe où elle est définie. La seconde information est le numéro de la ligne dans le code source du programme Java où pointe le pointeur de programme. Le numéro de la ligne est retrouvée dans une table qui associe adresse du pointeur de programme et numéro de ligne dans le code source. Notre machine récupère cette table à partir de l'attribut de code de la méthode (voir section 4.2).

Le tableau est stocké en dernier lieu dans la variable `backTrace` de la classe `Throwable`.

6.4.2 Affichage de la pile d'appel des méthodes

La fonction `printStackTrace` qui est définie par la classe `Throwable` affiche le contenu du tableau que notre machine virtuelle a préparée lors de l'appel de la fonction `fillInStackTrace`.

Notre machine virtuelle construit une chaîne Java dans laquelle elle place le nom de la classe, le nom de la méthode et le numéro de la ligne du code source pour toutes les entrées du tableau. Nous reconstituons ainsi l'environnement d'exécution exacte de la machine virtuelle lorsque l'exception c'est produite.

Cette chaîne est ensuite passée à la fonction `print` d'une classe représentant le flux de la sortie d'erreur standard. C'est cette dernière qui affiche les informations sur l'écran du terminal.

6.4.3 Exemple d'utilisation

Nous illustrons ici l'utilisation de la méthode `fillInStackTrace` et de la méthode `printStackTrace` par l'exemple suivant :

```
public class ex {
    public static void main(String av[]){
        try{
            t1();
        }catch(Exception e){
            System.out.println("exception catch");
            e.printStackTrace();
        }
    }
    public static void t1(){
        t2();
    }
    public static void t2(){
        int t[] = new int[5];

        for(int i = 0 ; i <= 5 ; i++) /* débordement */
            t[i] = i*i;
        System.out.println("Je ne passe pas ici.");
   }
}
```

Le programme Java précédent commence son exécution par la fonction `main`. Cette dernière déclare attraper les exceptions succeptibles d'être lancées par la fonction `t1`. Pour cela, l'appel de la fonction `t1` est contenu dans le bloc `try`.

Notre machine virtuelle effectue l'appel de la fonction `t1` qui elle appelle simplement la fonction `t2`. La fonction `t2` crée un tableau d'entiers qui est ensuite initialisé. Or la boucle `for` qui réalise l'initialisation est mal construite et va accéder au $6^{ième}$ élément d'un tableau qui n'en contient que 5. Notre machine virtuelle détecte l'erreur et lance une exception de type `ArrayIndexOutOfBoundsException`.

Pour cela, elle crée un objet de type `ArrayIndexOutOfBoundsException` qui lors de son initialisation exécute la fonction `fillInStackTrace` de la classe `Throwable`. Cette fonction parcourt la pile de *frames* et crée le tableau contenant les informations nécessaires à la fonction `printStackTrace`.

L'exécution de la méthode `t2` est immédiatement stoppée et notre machine virtuelle consulte la table des exceptions associée à cette méthode afin de trouver si elle intercepte cette exception. En l'occurrence la méthode `t2` ne possède pas de table d'exceptions et l'exécution se poursuit sur la méthode `t1`. La méthode `t1` ne possède pas de table d'exceptions. La machine virtuelle donne donc la main à la fonction `main` qui elle possède une table d'exception.

Notre machine recherche dans la table des exceptions si la fonction `main` intercepte les exceptions de type `ArrayIndexOutOfBoundsException`. Elle ne trouve pas directement ce type d'exception mais retrouve les exceptions de type `Exception`. Or la classe `Exception` est une classe parente de la classe `ArrayIndexOutOfBoundsException`. Ce sont donc les instructions associées au bloc `catch` de l'exception `Exception` qui sont exécutées, ce qui donne sur le terminal le résultat suivant :

```
exception catch
java.lang.ArrayIndexOutOfBoundsException:
        at ex.t2(ex.java:17)
        at ex.t1(ex.java:11)
        at ex.main(ex.java:4)
```

Lors de l'exécution du gestionnaire d'exception, la machine virtuelle affiche la chaîne `'exception catch'` et lance l'exécution de la fonction native `printStackFrame`. Cette dernière fait apparaître les fonctions et l'endroit dans le code source où s'est déroulé l'exception.

6.5 Réalisation

L'implémentation du mécanisme de gestion des exceptions a été réalisé simplement parce que nous avons pu regrouper l'interception des deux types d'exceptions qui peuvent être lancées, les exceptions lancées par la machine virtuelle et les exceptions logicielles. Cela représente environ 100 lignes de code C.

Chapitre 7

Réalisation du parallélisme dans notre machine virtuelle

Le langage Java donne la possibilité de créer et manipuler les *threads*. Les concepteurs de Java ont intégré ce concept aussi bien dans le langage que dans les librairies. Notre machine virtuelle supporte la création et la manipulation des *threads* Java en intégrant tous les mécanismes qui sont définis par les spécifications du langage.

Ce chapitre commence donc par la présentation de l'utilisation des *threads* par le langage Java. Cette introduction permet de mettre en évidence les mécanismes qui doivent être implémentés dans notre machine virtuelle. Nous continuons par une discussion sur les différentes approches possibles que nous avons étudiées pour doter notre machine virtuelle de *threads*. Nous donnons ensuite le détail de l'implémentation des composantes et des mécanismes que nous avons réalisés.

7.1 Les threads Java

Un *thread* est un simple flot d'exécution séquentiel dans un processus. Les *threads* possèdent un cycle de vie. La création d'un *thread* annonce le début de ce cycle et sa terminaison la mort du *thread*. Durant son cycle de vie, un *thread* exécute une séquence de code indépendamment des autres *threads* créés. Tous les *threads* partagent un espace de données commun et s'exécutent de manière concurrente.

Afin de fournir les *threads* les concepteurs du langage Java l'ont introduit directement dans le langage et dans la librairie Java[1]. Les classes `Thread`

[1] D'autres approches sont possibles. Sur les systèmes fournissant les *threads* au niveau du système d'exploitation, les différents langages utilisent des appels systèmes pour manipuler les *threads*. On retrouve ce mécanisme sur les systèmes Solaris 2.5 et Plan 9 par exemple.

et `ThreadGroup` permettent la création et la manipulation des *threads*. La classe `Thread` implémente les mécanismes de création, de synchronisation et de terminaison des *threads*, tandis que la classe `ThreadGroup` est chargée d'associer les *threads* dans des groupes[2].

7.1.1 Création des *threads*

La création d'un *thread* en Java repose sur la création d'un objet représentant une instance de la classe `Thread`. La classe `Thread` implémente l'interface `Runnable` qui déclare simplement une fonction `run()`. La fonction `run` est le cœur d'un *thread* ; c'est la fonction qui s'exécute après que le *thread* a commencé son exécution. Ainsi pour créer un *thread*, un objet de type `Runnable` doit être passé au constructeur de la classe `Thread` ou bien la classe `Thread` doit être sous-classée et doit définir sa propre méthode `run()`.

La première méthode de création d'un *thread* est donnée par le code Java suivant :

```
public class Premier implements Runnable {
...
    public void run() {
        // calcul des nombres premiers
    }
...
}
```

La classe `Premier` implémente l'interface `Runnable` et pour cela elle déclare une fonction `run()` qui contient le code qui sera exécuté par le *thread*. Pour créer et lancer le *thread* nous devons trouver du code Java ressemblant à

```
Premier p = new Premier();
new Thread(p).start();
```

L'objet de type `Premier` est créé et passé en argument au constructeur de la classe `Thread`. A ce moment le *thread* est créé par la machine virtuelle mais il ne commence son exécution qu'après l'appel de la méthode `start` de la classe `Thread`. La méthode `start` appelle simplement la méthode `run` de la classe `Premier` et le *thread* commence alors son exécution.

La seconde méthode de création d'un *thread* et certainement la plus utilisée peut être définie par le code Java suivant :

```
public class PremierThread extends Thread {
...
    public void run(){
```

Une autre approche consiste à utiliser une librairie implémentant les *threads* comme la librairie *pthread* qui, associée au langage C, permet de créer des applications *multi-threads* sous un système Unix.

[2] Un groupe de *threads* contient un ensemble de *threads* ou d'autres groupes de *threads*.

```
            // calcul des nombres premiers
        }
    ...
   }
```

La classe `PremierThread` descend de la classe `Thread` et réécrit la fonction `run`. La création et le lancement du *thread* devient alors quelque chose comme :

```
PremierThread p = new PremierThread();
p.start();
```

Le *thread* de type `PremierThread` est créé et on exécute la fonction `start` de cet objet qui va lancer l'exécution de la méthode `run` redéfinie dans la classe `PremierThread`.

7.1.2 Contrôle et propriétés des threads

La classe `Thread` définit des méthodes et des variables afin de contrôler l'exécution des *threads* dans la machine virtuelle. Les variables définies par la classe `Threads` sont déclarées d'accès privé et représentent l'état d'un *thread* au moment de l'exécution. Les méthodes de la classe `Threads` peuvent manipuler un *thread* pour modifier son comportement.

Propriétés des threads

Un *thread* pour être identifié par la machine virtuelle et par le programmeur Java est représenté par un objet de la classe `Thread` et possède un certain nombre d'attributs. Ces attributs sont déclarés sous forme de variables privées et ne sont donc accessibles qu'aux méthodes de la classe `Thread`. Ces attributs sont représentés dans le tableau 7.1.

Variable	Type	Description
`name`	`char[]`	nom du thread
`priority`	`int`	priorité du thread
`privateInfo`	`int`	informations utilisées par la machine virtuelle
`deamon`	`boolean`	indique si le thread est un démon
`still_born`	`boolean`	indique si le thread a été tué avant qu'il s'exécute
`target`	`Runnable`	fonction d'exécution du thread
`group`	`ThreadGroup`	groupe auquel le thread appartient

TAB. 7.1: Attributs d'un thread Java

Tous les *threads* créés portent un nom stocké dans le tableau de caractères `name`. Le nom d'un *thread* peut être défini par le développeur en le passant au constructeur de la classe `Thread`. Dans le cas où aucun nom n'est utilisé lors de la création, la classe *thread* en construit un par défaut en utilisant un compteur statique défini dans la classe `Thread`.

L'attribut `priority` indique la priorité que possède un *thread*. Cette variable est utilisée par l'ordonnanceur de la machine virtuelle pour décider de la répartition du temps d'exécution entre les différents *threads*. L'attribut `priority` peut prendre une des valeurs entières comprises entre `MIN_PRIORITY` et `MAX_PRIORITY`. Ces deux constantes définies par la classe `Thread` valent respectivement 1 et 10. Les *threads* possédant une priorité plus élevée sont exécutés plus souvent.

L'attribut `privateInfo` représente des informations privées utilisées par la machine virtuelle. Les informations stockées dans cet attribut sont laissées à la discrétion des développeurs de machine virtuelle et ne sont pas utilisées par les librairies Java.

L'attribut `deamon` est une variable de type booléenne qui indique que le *thread* est un démon. Lorsque l'attribut `deamon` d'un *thread* est vrai, le *thread* continue à s'exécuter normalement mais n'empêche pas la machine virtuelle de s'arrêter. En effet, la machine virtuelle doit continuer son exécution tant que tous les *threads* n'ont pas été tués, sauf si il s'agit de *threads* démons. En d'autres termes la machine virtuelle arrête de s'exécuter dès que tous les *threads* encore vivants sont des démons[3].

L'attribut `still_born` est une variable booléenne qui indique, quand elle est vraie, que le *thread* est mort avant d'avoir commencé son exécution. Cela peut arriver si, après la création d'un *thread*, la méthode `stop()` est appelée avant son lancement avec la méthode `start()`.

L'attribut `taget` contient l'objet qui implémente l'interface `Runnable` qui contient forcément la définition de la méthode `run()`. La méthode `run` constitue le corps du *thread*. Lorsque la méthode `run()` finit son exécution, le *thread* meurt.

L'attribut `group` contient une référence sur le groupe auquel le *thread* appartient. Si lors de la création d'un *thread* le groupe n'est pas défini, il appartient par défaut au groupe du *thread* qui l'a créé.

Contrôle des threads

Après leur création par la machine virtuelle, les *threads* peuvent être manipulés pour modifier leur comportement et ainsi jouer sur l'ordre dans lequel ils sont exécutés par la machine virtuelle. Le tableau 7.2 représente les méthodes de la classe `Thread` qui modifient le comportement des *threads*.

[3]Nous pouvons faire une analogie avec les démons Unix qui sont des processus tournant en permanence dans le système. Les *threads* démons de Java sont généralement utilisés dans cette optique : des *threads* présents pour remplir une tâche particulière, mais qui ne s'exécutent que quand nécessaire

Ces méthodes sont toutes déclarées *natives* et représentent l'implémentation des *threads* que nous devons réaliser pour notre machine virtuelle.

Nom	Valeur de retour	Description
`currentThread`	`Thread`	retourne le thread en cours d' exécution
`yield`	`void`	donne l'exécution à un autre thread
`sleep`	`void`	endort le thread courant
`start`	`void`	lance l'exécution d'un thread
`setPriority`	`void`	change la priorité d'un thread
`stop`	`void`	tue un thread
`suspend`	`void`	suspend l'exécution d'un thread
`resume`	`void`	reprend l'exécution d'un thread
`isAlive`	`boolean`	indique si un thread est vivant

TAB. 7.2: Les méthodes contrôlant l'exécution des threads

La méthode `currentThread` retourne un objet de type `Thread` qui est une référence sur le *thread* en cours d'exécution sur la machine virtuelle.

La méthode `yield` permet a un *thread* d'indiquer à la machine virtuelle qu'il désire passer l'exécution à un autre *thread* de même priorité. S'il existe un autre *thread* en attente et de même priorité, il s'exécute. S'il est le seul, il reprend son exécution. La méthode `yield` est le seul moyen sûr d'assurer une bonne répartition du temps d'exécution entre différents *threads*. En effet, comme nous le verrons dans les sections suivantes, les spécifications Java n'assurent pas la manière dont les *threads* sont ordonnancés.

La méthode `sleep` endort le *thread* en cours d'exécution durant un certain temps. Ce temps est déterminé par le paramètre de la méthode `sleep` ; il est exprimé en milisecondes. Le *thread* arrête donc son exécution et attend que la machine virtuelle le réveille. Durant le temps de son sommeil la machine virtuelle peut élire un autre *thread* pour l'exécuter.

La méthode `start` lance l'exécution d'un *thread* nouvellement créé. Elle appelle tout simplement la fonction `run` du *thread* et rend immédiatement la main. La machine peut alors choisir d'élire le *thread* qui vient d'être créé si et seulement s'il possède une priorité supérieure à tous les autres *threads* prêts à s'exécuter.

La méthode `setPriority` change de manière dynamique la priorité d'un *thread*. L'appel de cette fonction peut entraîner le changement du *thread* en cours d'exécution. Elle modifie l'ordonnancement des *threads* de la machine virtuelle.

La méthode `stop` arrête définitivement l'exécution d'un *thread*. Elle ne peut être appelée qu'une seule fois sur un *thread*.

La méthode `suspend` permet de stopper un thread. Le *thread* ainsi stoppé suspend son exécution jusqu'à l'appel de la méthode `resume`. Ces deux méthodes permettent par exemple à un *thread* de suspendre l'exécution d'un ou plusieurs autres *threads* et de les faire repartir quand bon lui semble. C'est une méthode primaire de synchronisation entre plusieurs *threads*.

La méthode `isAlive` indique simplement si un *thread* donné est encore en vie, c'est à dire si ce *thread* n'a pas exécuté sa méthode `stop` ou s'il n'est pas sorti de sa méthode `run`.

7.1.3 Mécanismes de synchronisation

En plus de fournir des fonctions permettant de contrôler et de jouer sur l'ordonnancement des *threads*, le langage Java fournit à l'aide de l'instruction `synchronized` un mécanisme intégré pour synchroniser l'accès aux méthodes et aux objets.

L'instruction `synchronized`

Le mot réservé `synchronized` peut être utilisé comme modificateur pour une méthode d'instance ou une méthode de classe, celle-ci est dîte *synchronisée*. Il indique qu'une méthode modifie l'état interne d'un objet ou d'une classe d'une façon qui n'est pas sûre vis-à-vis des *threads*.

Avant d'exécuter une méthode de classe synchronisée, la machine virtuelle doit verrouiller la classe, afin de garantir qu'une autre méthode exécutée par un autre *thread* ne peut pas modifier cette classe de façon concurrente. Avant d'exécuter une méthode d'instance synchronisée, la machine virtuelle doit verrouiller l'objet sur lequel a été invoquée la méthode, garantissant ainsi qu'un autre *thread* ne modifie pas l'objet en même temps.

Le code Java suivant montre l'utilisation de l'instruction `synchronized` comme modificateur de fonctions d'instances :

```
public CircularList {
    private CircularListNode current;

    public synchronized void add( Object *o ){
        // ajout d'un objet dans la liste circulaire
    }
    public synchronized void remove( Object *o ){
       // enlève un objet de la liste circulaire
    }
}
```

La classe `CircularList` représente ici l'implémentation d'une liste circulaire d'objets de la classe `Object`[4]. Les méthodes `add` et `remove` permet-

[4] L'utilisation d'objets de la classe `Object` permet d'insérer dans la liste tout type d'objet puisque tous les objets Java descendent de cette classe.

tent d'insérer et d'enlever un élément de la liste. Ces deux méthodes sont déclarées `synchronized` afin que ces opérations élémentaires sur la liste se fassent de manière atomique. Pour tenir une cohérence de la liste, il ne faut pas que deux *threads* différents puissent modifier en même temps la liste des objets.

Lorsqu'un *thread* accède à une des deux méthodes d'un objet de type `CircularList`, la machine virtuelle essaye d'obtenir un verrou sur cet objet. Si l'objet est déjà verrouillé, la machine virtuelle met en sommeil le *thread* ayant demandé le verrou. Elle relance l'exécution de ce *thread* pour une nouvelle demande d'obtention de verrou quand le verrou est libéré. Si l'objet n'est pas verrouillé alors le *thread* devient le propriétaire du verrou et exécute le corps de la méthode empêchant ainsi tout autre *thread* d'exécuter la méthode. Quand le *thread* sort de la méthode le verrou est relâché. Ce verrou sur l'objet est libéré même si la méthode lance une exception qui n'est pas attrapée.

Le mot réservé `synchronized` peut être utilisé aussi pour protéger un objet dans une suite d'instructions à l'intérieur d'une méthode. La syntaxe d'une telle utilisation est :

`synchronized(`*objet*`)` *instruction*

objet est un objet ou un tableau Java. *instruction* contient le code protégé, il s'agit généralement d'un bloc d'instructions encadré par { et }. L'instruction `synchronized` force le *thread* courant à obtenir un verrouillage exclusif de l'objet ou du tableau. Par exemple

```
public static void Compute(int[] tab){

    // instructions permettant divers calculs
    // à partir du tableau tab

    // début du tri du tableau
    synchronized(tab){
        //tri du tableau
    }

   // autres calculs sur le tableau trié
}
```

dans la méthode `Compute`, le tableau d'entiers `tab` est utilisé dans un premier temps pour permettre certains calculs qui ne modifient pas le contenu du tableau. Ensuite pour réaliser une autre série de calculs, le tableau doit être trié. Pour synchroniser l'accès au tableau `tab`, la procédure de tri est placée dans un bloc `synchronized` évitant ainsi à plusieurs *threads* de modifier le tableau en même temps. Dans ce cas d'utilisation de l'instruction `synchronized` sur un bloc d'instructions, l'obtention du verrou se déroule de la même manière que pour les méthodes *synchronisées*.

Les fonctions de synchronisation de la classe `Object`

La classe `Object`, qui se trouve au sommet de la hiérarchie des classes Java, définit trois méthodes simplifiant la synchronisation des *threads*. Ce sont les méthodes `wait()` et `notify()`.

L'instruction `synchronized` que nous venons de décrire précédement permet de synchroniser l'accès à une méthode ou à un bloc d'instructions. L'utilisation des méthodes `wait()` et `notify()` de la classe `Object` permet d'étendre ses fonctionnalités. Un *thread* peut avec la méthode `wait()` libérer un verrou à un endroit de son choix et attendre qu'un autre *thread* lui redonne ce verrou en utilisant la méthode `notify()`. Toutes ces opérations de synchronisation doivent impérativement s'exécuter dans un bloc ou une méthode *synchronisée*.

En exécutant la méthode `wait()` depuis un bloc synchronisé, un *thread* libère le verrou qu'il avait acquit et s'endort. Un *thread* peut réaliser cette opération s'il a besoin d'attendre que quelque chose se déroule dans une autre partie de l'application. Cet événement se produit quand un autre *thread* utilise la méthode `notify()` depuis un bloc synchronisé sur cet objet. Maintenant, le premier *thread* se réveille et tente d'obtenir le verrou de cet objet pour continuer son exécution. Il est tout à fait possible que le premier *thread* n'obtienne pas le verrou tout de suite et peut être jamais.

Pour illustrer l'utilisation des méthodes `wait()` et `notify()` considérons l'exemple suivant :

```
public class Test {

    public synchronized void waiter(){
        // effectue un travail

        // on attend que quelque chose se passe
        wait();

        // on reprend le travail
    }

    public synchronized void notifier(){
        // effectue un travail

        // on prévient que le travail est réalisé
        notify();

       // on reprend le travail
   }
```

```
}
```

Supposons qu'un premier *thread* obtienne le verrou sur un objet de la classe `Test` et commence l'exécution de la méthode `waiter`. Pendant ce temps, un second *thread* s'est endormi en attendant d'obtenir ce même verrou dans le but de pouvoir exécuter la méthode `notifier`. Le premier *thread* travaille et appelle la méthode `wait()`. Il libère ainsi le verrou sur l'objet `Test` et s'endort. Le second *thread* qui était en attente obtient le verrou et commence l'exécution de la méthode `notifier`. Après avoir effectué le travail dont avait besoin le premier *thread*, il appelle la méthode `notify()` qui indique au premier *thread* que le travail dont il avait besoin est disponible. Le premier *thread* se réveille et tente d'obtenir le verrou sur l'objet. Or, le verrou est toujours détenu par le deuxième *thread* qui continue d'exécuter la méthode `notifier`. Lorsque ce *thread* sort de la méthode `notifier`, il libère le verrou et permet ainsi au premier *thread* de continuer son exécution là où il l'avait laissé.

Les méthodes `wait()` et `notify()` de la classe `Object` sont particulièrement utiles pour développer des mécanismes mettant en œuvre des relations entre des *threads producteurs* et des *thread consommateurs*.

La méthode `wait()` possède une variante qui permet de spécifier le temps que doit attendre un *thread* avant qu'il soit notifié. Il existe aussi une méthode `notifyAll()` qui prévient tous les *threads* en attente d'un événement (plusieurs *threads* peuvent avoir effectué un `wait`), qu'un événement s'est produit et les réveille tous. En effet, la méthode `notify()` ne réveille que le premier *thread* ayant fait un `wait()`.

7.2 Les différentes méthodes possibles de réalisation des threads

L'implémentation des *threads* dans notre machine virtuelle n'a pas été aussi directe que les autres composantes. Les spécifications Java n'imposent pas d'implémentation particulière des *threads* et laissent le choix au développeur de choisir une technique qui conviendra à la machine qu'il développe et au système sur lequel cette machine s'exécutera.

Les choix d'implémentations interviennent principalement dans la manière dont les *threads* seront ordonnancés et synchronisés lors de leurs exécution. Pour cela la seule restriction que doit respecter une machine virtuelle Java est la suivante :

Si la machine virtuelle exécute plusieurs threads simultanément, les threads de plus hautes priorités doivent s'exécuter avant les threads de plus basses priorités.

L'ordonnancement des *threads* possédant la même priorité n'est pas prévisible et est laissé à la discrétion du développeur de la machine virtuelle. Cela n'est pas sans poser quelques problèmes au programmeur Java. L'exécution d'une application Java utilisant plusieurs *threads* n'est pas la même suivant qu'elle s'exécute sur une machine virtuelle ou une autre. Bien évidemment cela n'arrive que si l'application ne prend par garde à la manière dont elle utilise et synchronise les *threads*. C'est une servitude du développeur Java de faire en sorte que les *threads* partagent correctement leurs temps d'exécution sur la machine virtuelle.

On rencontre deux types d'ordonnanceurs sur les machines virtuelles à l'heure actuelle :

- Le premier est un ordonnanceur de type FCFS[5] où un *thread* qui a été élu, s'exécute tant qu'il ne donne pas explicitement la main à un autre *thread*[6]. Pour une machine virtuelle Java cela ne peut se produire que pour les *threads* de même priorité puisqu'un *thread* de plus haute priorité doit s'exécuter immédiatement[7].
- Le second type d'ordonnanceur est dit à partage de temps [8]. On retrouve le terme de *round-robin* ou *time-slicing* pour le désigner. Les *threads* partagent de manière équitable leurs temps d'exécution suivant leur niveau de priorité. Cette technique, très répandue dans les ordonnanceurs de systèmes d'exploitations, est bien plus naturelle et permet de corriger les fautes de programmation qui sont bien souvent fatales lorsque l'on utilise un ordonnanceur de type FCFS.

Pour ajouter la gestion des *threads* à notre machine virtuelle, nous avons donc étudié plusieurs possibités d'implémentation que nous décrivons maintenant.

7.2.1 Utilisation des threads Plan 9

Le système d'exploitation Plan 9 permet la création de processus partageant un même espace d'adressage[9]. Ce type de processus peut être facile-

[5]FCFS est l'acronyme de Fisrt Come First Served (premier arrivé premier servi).

[6]Les programmeurs d'applications Java *multi-threads* doivent toujours concevoir leurs applications en pensant qu'ils développent pour ce type d'ordonnanceur même si leur machine virtuelle n'utilise pas cette technique.

[7]La version 1.0 de la machine virtuelle Java de Sun pour les systèmes Unix utilise cette technique.

[8]Les versions 1.0 de la machine virtuelle Java pour les systèmes d'exploitations Windows 95 et Windows NT utilisent un ordonnanceur à partage de temps. Il est assez ironique de voir que la qualité des ordonnanceurs Java est inversement proportionnelle à celui du système d'exploitation sur lequel il tourne

[9]La création de processus Plan 9 se réalise à l'aide de l'appel système `rfork`. Cet appel système est plus riche que son équivalent sur les systèmes de types Unix. En lui passant un entier en paramètre, il permet de créer un processus avec des propriétés particulières. Les différentes ressources que peut partager un processus avec son père sont les suivantes : la

ment assimilé à des *threads*. Ils s'exécutent sur le système, non pas comme les *threads* qui eux s'exécutent à l'intérieur d'un processus. Ceci n'est à priori pas trop gênant dans notre cas puisqu'ils peuvent partager des données. L'ordonnancement est pris en charge par le système et ils sont ordonnancés comme tous les autres processus s'exécutant sur le système.

De plus pour gérer la synchronisation de plusieurs processus, Plan 9 fournit une série d'appels de fonctions qui permettent la gestion de verrous[10]. Ce mécanisme de verrou est identique à celui que nous devons implémenter dans notre machine virtuelle et que nous avons détaillé à la section 7.1.3.

Le système Plan 9 fournit aussi en ensemble de fonctions systèmes qui permettent le contrôle, le dialogue et l'échange d'informations entre plusieurs processus. Il existe notamment un système d'envoi de *note* à un ou des processus que l'on peut comparer au système d'envoi de signaux des systèmes Unix[11]. Un appel système `sleep` permet d'endormir un processus pendant un temps donné[12] (en milisecondes).

Le système Plan 9 fournit donc tous les mécanismes dont nous avons besoin pour l'implémentation des *threads* dans notre machine virtuelle. Nous avons donc commencé par développer une première version qui utilisait pleinement les possibilités qu'offrait le système.

Les problèmes rencontrés

Comme nous venons de le voir, les *threads* du système Plan 9 sont des processus qui partagent un même espace d'adressage[13] et sont ordonnancés par le système. L'ordonnanceur du système répartit uniformément le temps d'exécution des différents processus actifs sans notion de priorité. La notion de priorité pour un processus Plan 9 est absente. Or comme nous l'avons vu la notion de priorité pour les *threads* Java est très importante puisqu'elle constitue la principale spécification que doit respecter une machine virtuelle implémentant les *threads*.

Nous avons tout de même commencé l'implémentation de *threads* Java sans nous soucier de ce problème de priorité. Nous nous sommes vite aperçu

mémoire, les descripteurs de fichiers, l'espace de nommage, les variables d'environnements et le groupe de processus.

[10] Les verrous que fournit le système Plan 9 sont accessibles par les fonctions : `lock()` qui essaie d'obtenir un verrou, `unlock()` qui libère un verrou et la fonction `canlock()` qui permet d'obtenir un verrou mais qui n'est pas bloquante.

[11] Les notes Plan 9 représentent un système plus général que les signaux Unix puisqu'une note est une chaîne de caractères. Un processus recevant une note reçoit la chaîne de caractères alors que les signaux Unix sont typés et existent en nombre limités par le système.

[12] Si le temps donné vaut 0 miliseconde, l'ordonnanceur du système passe la main à un autre processus.

[13] La pile d'exécution est privée, bien sûr.

que dans un contexte *multi-threads* comme celui là, le partage des ressources de la machine virtuelle par plusieurs *threads* devient très vite problématique. En effet, imaginons qu'un *thread* exécute une instruction qui demande la résolution d'une variable de classe. Il ne faut en aucun cas que d'autres *threads* réalisent ce type d'opération en même temps sous peine de mettre la machine virtuelle dans un état incohérent[14]. Pour éviter cela, nous avons ajouté des verrous pour synchroniser l'accès aux données critiques de notre machine virtuelle. Ces verrous viennent s'ajouter aux verrous qui sont destinés à gérer l'implémentation de l'instruction `synchronized` qui est utilisée dans les programmes Java.

Lors de tests, nous nous sommes aperçu que l'exécution des *threads* était très lente par rapport à des applications n'utilisant pas les *threads*.[15] En fait, il s'avère que la synchronisation des processus se réalise mal et de manière complètement désordonnée. Les *threads* d'une application Java utilisent beaucoup de méthodes synchronisées se trouvant dans les librairies et qui viennent s'ajouter à la synchronisation nécessaire à la machine virtuelle et provoquent une augmentation du temps système utilisé par les *threads*. Nous avons constaté que les *threads* passait plus de temps à s'exécuter en mode système qu'en mode utilisateur.

Pour résoudre ce problème de priorité que nous avons évoqué précédemment nous aurions pu concevoir un système de synchronisation des différents processus afin d'intégrer la notion de priorité. Mais il nous a semblé que cela alourdirait encore plus notre système puisque dans cette hypothèse notre machine aurait dû elle même distribuer le temps d'exécution entre les différents *threads*. On peut se demander pourquoi utiliser les processus Plan 9 alors que la machine virtuelle fait le travail d'ordonnancement ?

7.2.2 Implémentation d'un ordonnanceur dans la machine virtuelle

La seconde implémentation que nous avons étudiée consiste à considérer notre machine comme un seul système complet qui est capable de créer, manipuler et exécuter des *threads*. Pour cela la machine virtuelle doit être capable d'ordonnancer les *threads* seule sans utiliser les processus du système d'exploitation.

Cette approche permet de mieux appréhender les problèmes que nous

[14] Le mécanisme de résolution peut entraîner le chargement, l'édition de liens et l'intitialisation de une ou plusieurs classes. Si ces étapes sont réalisées simultanément sur les structures de données internes de la machine virtuelle, cela peut entraîner des erreurs graves à divers endroits.

[15] Nous avons créé un simple programme affichant des chaînes de caractères et nous l'avons comparé a une version d'un programme faisant la même chose en utilisant 2 *threads* qui s'exécutait l'un après l'autre. L'utilisation de deux *threads* ne justifiait pas autant la différence de vitesse, de l'ordre de 2 à 3 secondes, entre les deux exécutions.

avons rencontré dans la réalisation précédente. La notion de priorité qui fait partie des spécifications peut être réalisée sans grande difficulté. C'est le rôle de l'ordonnanceur de la machine virtuelle d'élire le *thread* qui convient. Le contrôle est ainsi total si l'ordonnanceur est correctement implémenté.

Les problèmes de synchronisations sont ici réduits puisque le contrôle de l'exécution des *threads* est complètement pris en charge par la machine elle même. La machine peut ainsi choisir l'endroit exact où arrêter le *thread* en cours d'exécution afin de ne pas compromettre l'état interne de ses données.

Cette solution nécessite un travail plus important de développement puisque nous devons implémenter nous même l'ordonnanceur et le mécanisme de verrou alors que, pour la solution précédente, ces deux parties étaient prises en charge par le système d'exploitation.

C'est cette dernière implémentation des *threads* que nous avons choisi pour notre machine virtuelle. Elle est décrite en détail dans les sections qui suivent.

7.3 Ordonnancements des threads

Nous décrivons dans cette section notre implémentation de l'ordonnanceur et des primitives de manipulations des *threads* tels quelles sont imposées par la librairie Java.

7.3.1 Structures des données associées aux threads

Un thread Java est représenté par une instance de la classe `Thread`. Cet objet possède des variables d'accès privés qui définissent le *thread*. On y retrouve son nom, le groupe auquel il appartient et un champ qui contient des données privées (voir le tableau 7.1 page 89). Cette variable `privateInfo` contient des informations spécifiques à notre machine virtuelle et sont nécessaires à l'ordonnanceur.

Ces données sont représentées par la structure `thContext` suivante :

```
struct thContext {
    int state;
    int prio;
    StackFrame *stack;
    int data;
    Object *thread;
    thContext *next;
};
```

Le champ `state` indique dans quel état se trouve le *thread*, il vaut une des valeurs présentées dans le tableau 7.3.

Etat	Valeur	Désignation
TH_RUNNING	1	en cours d'exécution
TH_READY	2	en attente d'exécution
TH_READY_LOCK	3	en attente d'exécution et d'un verrou
TH_SLEEP	4	en sommeil
TH_SUSPEND	5	exécution suspendue
TH_WAIT	6	en attente d'un verrou
TH_DEAMON	8	thread démon

TAB. 7.3: Les états des threads

Le tableau 7.3 regroupe tous les états possibles dans lequel un *thread* peut se trouver. En aucun cas la variable `state` ne peut contenir une autre valeur sous peine de compromettre gravement l'intégrité de notre machine.

Le champ `prio` contient la priorité du *thread*. C'est une valeur comprise entre un et dix[16]

Le champ `stack` est une référence sur la pile de *frames* d'un *thread*. C'est sur cette pile que le *thread* effectue son exécution quand l'ordonnanceur de notre machine virtuelle le choisit.

Le champ `data` est utilisé pour contenir des données variées. Il est utilisé pour l'implémentation des méthodes `wait()` et `notify()` que nous décrivons dans la section 7.4.

Le champ `Object` contient une référence sur l'objet de type `Thread` que représente la structure `thContext`. Elle permet uniquement à la machine virtuelle de retrouver rapidement l'objet de type `Thread`.

Le champ `next` est une référence sur la prochaine structure `thContext` d'une liste. Comme nous le verrons les *threads* sont regroupés dans des listes chaînées qui assemblent les *threads* se trouvant dans le même état.

Une structure `thContext` est créée par *thread* et permet d'identifier les *threads*. C'est cette structure de données que manipule notre ordonnanceur. Nous aurions pu utiliser directement l'objet de type `Thread` mais il est plus difficile à manipuler que la structure `thContext`[17]. On peut donc bien différencier l'utilisation d'un objet de type `Thread` et la structure `thContext` ; l'objet de type `Thread` est utilisé par le programmeur Java et les librairies alors que la structure `thContext` est utilisée par la machine

[16] Ce sont les constantes `MIN_PRIORITY` et `MAX_PRIORITY` définies par la classe `Thread`.

[17] Un *thread* en Java est obligatoirement un objet de type `Thread`. Néanmoins il arrive souvent que le programmeur Java conçoive un *thread* en définissant un objet qui hérite de la classe `Thread`. Dans ce cas l'objet contient les variables de l'objet plus celles qu'il hérite de la classe `Thread`. Ceci implique que la machine virtuelle doit rechercher en permanence l'emplacement des variables définies par la classe `Thread` dans cet objet. Ce mécanisme est plus lent que de manipuler directement la structure `thContext` qui contient toutes les informations directement nécessaires.

virtuelle.

7.3.2 L'implémentation de l'ordonnanceur

Notre ordonnanceur est implémenté par une simple fonction C dans notre machine virtuelle, la fonction `schedule()`. Son but est d'élire un *thread* pour qu'il puisse s'exécuter. L'ordonnanceur choisi le *thread* de plus haute priorité dans les *threads* qui sont en attente d'exécution.

Les listes de threads

Les *threads* de notre machine virtuelle sont maintenus dans des listes qui regroupent un ensemble de *threads* se trouvant dans le même état. Ainsi nous avons créé 10 listes contenant les *threads* par niveau de priorité (les niveaux de priorités des threads varie de 1 à 10). La figure 7.1 montre l'ensemble des listes que gère notre machine virtuelle.

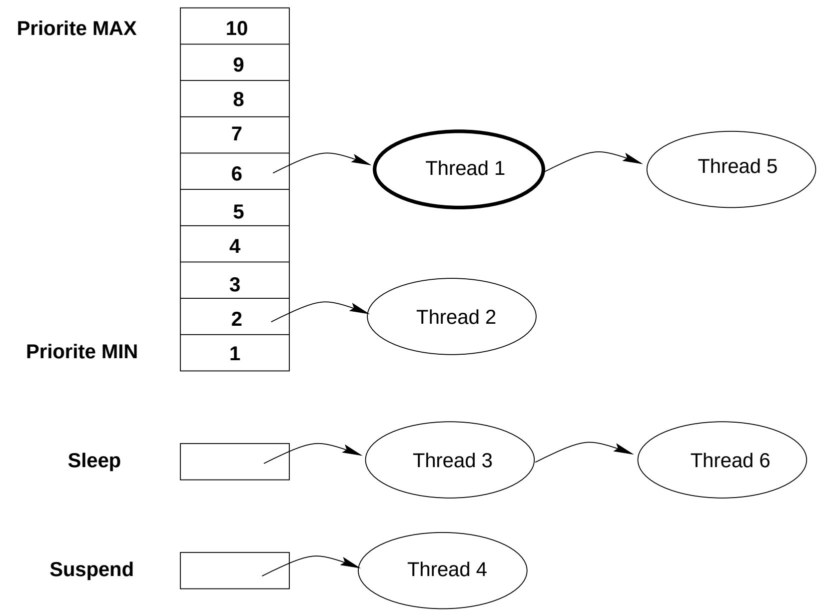

FIG. 7.1: Ensemble des listes de threads

Tous les *threads* créés et en attente d'exécution sont placés sur la liste correspondant à leur priorité. Sur la figure 7.1, les *threads* numérotés 1 et 5 possèdent tous les deux une priorité de 6, tandis que le *thread* numéroté 2 possède une priorité inférieure de 2.

Les *threads* de la figure 7.1 représentent les structures `thContext`. Les structures sont chaînées à l'aide du champ `next` et la dernière structure de la liste ne possède pas de suivant. Tous les *threads* de ces listes de priorités sont en attente d'exécution. Le champ `state` est obligatoirement positionné à `TH_READY` ou `TH_READY_LOCK`. `TH_READY` indique que le *thread* attend son tour pour s'exécuter alors que `TH_READY_LOCK` indique que le *thread* doit, avant de s'exécuter, concourir pour l'obtention d'un verrou (voir la section 7.4 sur la synchronisation). Enfin, le *thread* qui a été élu par l'ordonnanceur et qui s'exécute est toujours placé en tête de liste et son champ `state` est toujours positionné à `TH_RUNNING`. Il est bien évident qu'un seul et unique *thread* peut se trouver dans cet état là (le *thread* 1 de la figure 7.1).

Nous remarquons aussi sur la figure 7.1 la présence de deux autres listes. La première qui contient les *threads* numérotés 3 et 6 représente les *threads* qui sont en sommeil et le champ `state` de leurs structures `thContext` est positionné à `TH_SLEEP`. La deuxième contient tous les *threads* qui ont été suspendus et sont positionnés à `TH_SUSPEND`.

L'ordonnanceur et son principe de fonctionnement

Notre machine virtuelle est capable d'exécuter plusieurs *threads*. Le problème qui se pose alors est de savoir comment passer l'exécution à un autre *thread*. Il faut pouvoir interrompre à tout moment le *thread* courant pour pouvoir élire un *thread* de priorité plus importante. Ce mécanisme de changement de contexte d'exécution est implémenté à l'aide de trois variables globales : les variables `need_schedule`, `current_stack` et `current_frame`.

La variable globale `need_schedule` est un indicateur qui permet de savoir si l'on doit interrompre le *thread* en cours d'exécution. Par défaut cette variable est positionnée à faux. Elle est positionnée à vraie quand la machine virtuelle demande l'exécution de l'ordonnanceur dans le but d'élire un autre *thread*. Le test de la variable `need_schedule` est effectué dans l'interpréteur après l'exécution de chaque instruction.

Les variables `current_stack` et `current_frame` contiennent la pile de *frames* et la *frame* du *thread* en cours d'exécution. Ainsi pour effectuer un changement de contexte, notre ordonnanceur doit juste changer ces deux variables pour qu'elles contiennent respectivement la pile d'exécution et la *frame* courante du nouveau *thread* qui vient d'être élu.

Dans notre machine virtuelle l'ordonnanceur est implémenté par la fonction `schedule()` dont nous donnons le code ci-après.

```
void schedule(void)
{
    int i; thContext *ctx, *cur; Thread *t;

begin:
    need_schedule = 0;
    checkAlarm();

    for(i = MAX_PRIO ; i >= MIN_PRIO ; i--){
        if( runQ[i] ){ /* thread de plus haute priorité */
            ctx = runQ[i];

            if( current_thread ){
                t = findThread(current_thread);
                cur = (thContext *)t->private;
                if( cur->state == TH_RUNNING ) /* replace le thread           */
                    cur->state = TH_READY;     /* pré-empté pret à s'exécuter */
            }

            if( ctx->state == TH_READY_LOCK ){ /* Ce thread doit faire un lock */
                Object *obj = (Object *)ctx->data;

                current_thread = ctx->thread;
                if( getLock(obj) < 0 )
                    goto begin;
            }

            /* changement de context */
            ctx->state     = TH_RUNNING;
            current_thread = ctx->thread;
            current_stack  = ctx->stack;
            current_frame  = current_stack->stack[current_stack->pt];
            return;
        }
    }
    current_thread = 0; current_stack = 0; current_frame = 0;
    if( suspendQ || alarmList ){ /* il y a encore des  threads */
        while( !need_schedule )
            sleep(0);
        goto begin;
    }
    exits("ok"); /* plus de threads quitte la JVM */
}
```

Notre ordonnanceur doit être capable à tout moment de retrouver le *thread* de plus haute priorité qui doit s'exécuter. Pour cela, il consulte simplement toutes les listes de priorités en commençant par la liste de plus haute priorité. Dès qu'il trouve une liste non vide, il choisit toujours de donner l'exécution au premier *thread* de la liste et retourne là où il a été appelé ; c'est à dire dans la boucle de l'interpréteur où le nouveau *thread* élu s'exécute[18].

Si toutefois après le parcours de la liste des priorités aucun *thread* n'est prêt à s'exécuter, l'ordonnanceur teste si il existe des *threads* qui sont suspendus ou endormis. Dans ce cas il boucle jusqu'à ce qu'il se produise un événement qui pourrait remettre un *thread* dans les listes de priorités[19]. Si aucun *thread* n'est endormi ou suspendu la machine virtuelle s'arrête.

7.3.3 Les primitives de contrôle des threads

L'implémentation des fonctions que nous allons décrire maintenant sont les méthodes natives qui sont définies par la classes *thread* et qui ont été présentées à la section 7.1.2 et dans le tableau 7.2 page 91. Elles représentent les méthodes qui permettent de manipuler les *threads* et par conséquent celles qui jouent sur l'ordonnancement des *threads*.

La méthode `yield()`

La méthode `yield()` force la machine virtuelle à donner la main à un autre *thread* de même priorité si cela est possible. Elle récupère simplement la priorité du *thread* courant et le recherche dans la liste des priorités. Il doit se trouver obligatoirement en tête de la liste puisqu'il est en cours d'exécution. La structure `thContext` représentant le *thread* dans la liste de priorité est tout simplement passée en fin de liste. Le *thread* passe à l'état `TH_READY` et la variable globale `need_schedule` est passée à vraie pour donner la main à l'ordonnanceur.

La méthode `sleep()`

La méthode `sleep` endort le *thread* courant pour une durée déterminée. La valeur indiquant le temps de sommeil est passée en paramètre à la méthode. Pour gérer ce mécanisme nous utilisons l'appel système `alarm` qui permet d'envoyer une note de type `alarm` à un processus Plan 9 après un temps donné.

Pour cela notre machine virtuelle commence par enlever des listes de priorités le *thread* courant. Elle le place ensuite dans une liste un peu particulière qui contient tous les *threads* endormis représentés sur la figure 7.2.

Cette liste de *threads* (la liste `alarmList`) contient tous les *threads* endormis classés dans l'ordre de réveil. Lorsqu'un nouveau *thread* est inséré son heure de réveil est calculée en ajoutant à l'heure système le nombre de

[18] L'interpréteur s'exécute toujours sur la pile et la *frame* courante qui sont les variables globales `current_stack` et `current_frame` modifiées par la fonction `schedule()`.

[19] Le réveil d'un *thread* peut repositionner un *thread* sur la listes des priorités (voir la section 7.3.3).

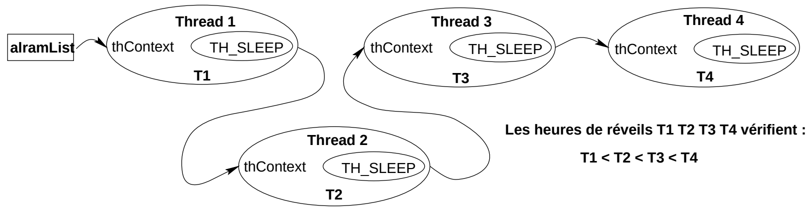

FIG. 7.2: Insertion d'un thread dans la liste des threads endormis

milisecondes que doit dormir le *thread*. Le nouveau *thread* est ensuite ajouté dans la liste `alarmList` de telle sorte que la liste soit triée dans l'ordre croissant de réveil. Sur la figure 7.2, le *thread* 2 est ajouté entre les *threads* 1 et 3 afin que la liste soit toujours triée. La machine virtuelle précise ensuite au système d'exploitation dans combien de temps notre machine virtuelle désire recevoir une note à l'aide de l'appel système `alarm`. Ce temps est calculé par rapport à l'heure de réveil du premier *thread* présent dans la liste.

La variable `need_schedule` est ensuite positionnée à vrai pour forcer l'ordonnanceur à élire un autre *thread*.

Pour récupérer la note, la machine virtuelle à auparavant positionnée un *handler* qui permet d'intercepter la note. Ceci est réalisé lors de l'initialisation de notre machine virtuelle à l'aide de l'appel système `notify()`. Lors de la réception d'une note de type `alarm`, la machine virtuelle est interrompue et notre *handler* est appelé. Celui positionne la variable `need_schedule` à vrai. Ainsi après la fin de l'exécution de l'instruction courante, l'interpréteur détecte que quelque chose s'est passé et donne la main à l'ordonnanceur.

Quand l'ordonnanceur prend la main, il commence toujours par exécuter la fonction `checkAlarm()` (voir le code de la fonction `schedule()` présenté page 103). Cette fonction parcourt la liste courante des *threads* endormis et repositionne tous les *threads* qui doivent être réveillés sur leurs listes de priorités respectives. S'il existe encore des *threads* qui doivent être réveillés plus tard, la fonction `checkAlarm()` relance l'appel système `alarm` afin de pouvoir réveiller le premier des *threads* restants.

Tous les *threads* qui doivent être réveillés sont maintenant prêt à s'exécuter et peuvent être élus par l'ordonnanceur.

La méthode `setPriority`

Cette méthode change la priorité d'un *thread*. Le champ `prio` de la structure `thContext` du *thread* est changé. Si toutefois le *thread* se trouve sur une des listes de priorités, c'est à dire qu'il est dans l'état `TH_RUNNING` ou `TH_READY`, il est changé de liste pour être replacé dans la liste correspondant à sa nouvelle priorité. Enfin, la variable `need_schedule` est positionnée à vrai pour appeler l'ordonnanceur après la fin de l'exécution de l'instruction courante dans l'interpréteur.

La méthode `stop`

Cette méthode stoppe l'exécution d'un *thread* de manière définitive. Le *thread* est simplement enlevé de la liste dans laquelle il se trouve et la variable `need_schedule` est positionnée à vrai de façon à élire un autre *thread*.

La méthode `suspend`

Cette méthode suspend temporairement l'exécution d'un *thread*. Pour cela notre machine virtuelle place simplement le *thread* sur une liste qui contient tous les *threads* suspendus. Un autre *thread* est ensuite élu.

La méthode `résume`

Cette méthode relance l'exécution d'un *thread* précédemment suspendu. La machine virtuelle enlève le *thread* de la liste des *threads* suspendus et le place en tête de sa liste de priorité.

7.4 La synchronisation

7.4.1 Les primitives de verrouillages

Notre machine virtuelle définit deux primitives pour gérer les verrous : la fonction `getLock()` qui permet à un *thread* d'obtenir un verrou et la fonction `releaseLock()` qui permet de libérer un verrou. Ces deux fonctions reçoivent en argument un objet sur lequel une opération de verrouillage doit être réalisée.

Pour cela nous avons ajouté une structure de données qui a été insérée dans la structure `Object` qui définit un objet (voir la section 5.6). Cette nouvelle structure est la suivante :

```
struct Sync {
    int ref;
    Object *thread_owner;
    thContext *wait;
    thContext *event;
};
```

Le champ `ref` de la structure `Sync` contient le nombre de fois que l'objet a été verrouillé. Le mécanisme de verrouillage doit être ré-entrant. En effet, un *thread* après avoir obtenu un verrou peut très bien obtenir une nouvelle fois ce même verrou. Dans ce cas, le champ `ref` est incrémenté autant de fois qu'un *thread* a acquis le verrou.

Le champ `thread_owner` est une référence sur le *thread* qui possède le verrou.

Le champ `wait` est la tête d'une liste chaînée qui contient tous les *threads* qui sont en attente du verrou, tandis que le champ `event` est le début d'une liste de *threads* qui ont exécuté la méthode `wait()` sur cet objet. Ces derniers attendent l'exécution de la méthode `notify()` par un autre *thread*.

La fonction `getLock()`

La fonction `getLock()` est utilisée pour obtenir un verrou sur un objet. Trois cas de figures peuvent se présenter :

1. L'objet sur lequel le *thread* fait la demande de verrou est libre et donc le champ `ref` de la structure `Sync` est à zéro. Dans ce cas, le verrou est accordé au *thread*. Le champ `ref` passe à 1 et le champ `thread_owner` contient la référence sur le *thread* propriétaire du verrou.
2. L'objet est déjà verrouillé par le même *thread*. Dans ce cas, comme la référence `thread_owner` correspond au *thread* qui fait la demande de verrouillage, le champ `ref` est simplement incrémenté.
3. L'objet est déjà verrouillé par un autre *thread* et la variable `ref` est strictement supérieure ou égale à un. Le *thread* courant est enlevé de sa liste de priorité et placé dans la liste des *threads* déjà en attente du même verrou (le champ `wait` de la structure `Sync` de l'objet). La figure

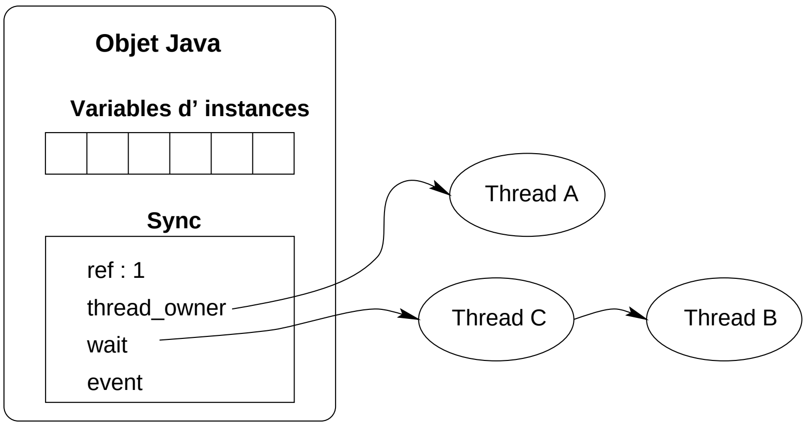

FIG. 7.3: La liste d'attente sur un objet

7.3 représente cette situation. Le `thread C` tente d'obtenir le verrou sur un objet qui est déjà détenu par le `thread A`, il est donc ajouté à la liste des *threads* en attente. Le champ `state` de la structure de `thContext` est positionné à `TH_WAIT`. La variable `need_schedule` est ensuite passée à vrai pour donner la main à l'ordonnanceur.

La fonction `releaseLock()`

La fonction `releaseLock()` est utilisée pour permettre à un *thread* qui a obtenu un verrou de le libérer, afin que d'autres *threads* en attente puissent utiliser cet objet.

Lors de l'appel de la fonction `releaseLock()` sur un objet, le champ `ref` est décrémenté, ce qui indique que le *thread* qui posséde le verrou tente de le libérer. Comme le mécanisme de verrouillage est ré-entrant et qu'un *thread* peut avoir verrouillé plusieurs fois l'objet, la machine virtuelle attend que le champ `ref` soit égal à zéro pour réveiller les *threads* en attente du même verrou.

Quand la variable `ref` passe à la valeur zéro, les *threads* en attente sur la liste référencée par le champ `wait` sont réveillés. Ils sont placés sur leurs listes de priorités respectives et l'ordonnanceur est appelé en positionnant la variable `need_schedule`.

7.4.2 La synchronisation sur l'accès aux objets

Comme nous l'avons vu dans la section 7.1.3 la première manière de synchroniser l'accès à une section critique de code est d'utiliser l'instruction `synchronized` sur un objet. Pour réaliser ce mécanisme le compilateur utilise deux instructions particulières : l'instruction `monitorenter` et l'instruction `monitorexit`. Ces deux instructions englobent le bloc d'instructions qui constitue la section critique.

L'instruction `monitorenter` récupère sur le sommet de la pile des opérandes l'objet sur lequel le verrou doit être obtenu. La machine virtuelle exécute la fonction `getLock()` sur cet objet.

Si le verrou n'a pas été obtenu, la fonction `getLock()` a mis le *thread* sur la liste des *threads* en attente et l'interpréteur abandonne l'exécution de l'instruction en cours. Comme la variable `need_schedule` a été positionnée à vrai par la fonction `getLock()`, l'interpréteur appelle aussitôt l'ordonnanceur qui élit un autre *thread*.

Si le verrou est obtenu par le *thread*, l'instruction se termine et le *thread* courant continue son exécution. Avant de terminer l'instruction et d'exécuter la permière instruction de la section critique, l'interpréteur sauvegarde dans la *frame* courante l'objet qui vient d'être vérouillé. Cela permet à l'interpréteur de pouvoir dévérouiller l'objet en appelant la fonction `releaseLock()` si la section critique est quittée brutalement sans avoir exécuté l'instruction `monitorexit`.

Pour terminer la section critique, l'interpréteur exécute simplement l'instruction `monitorexit`. Elle retrouve l'objet à libérer sur le sommet de la pile des opérandes et appelle tout simplement la fonction `releaseLock()`. C'est cette dernière qui réveille les *threads* qui étaient en attente sur cet objet.

7.4.3 La synchronisation sur l'accès aux méthodes

Les méthodes dites *synchronisées* n'utilisent pas tout à fait le même mécanisme que nous venons de décrire pour les objets. Les méthodes *synchronisées* n'utilisent pas les instructions `monitorenter` et `monitorexit` ; le mécanisme de vérouillage est intégré dans l'implémentation des instructions qui réalisent l'appel des méthodes.

Pour cela la machine virtuelle cherche si le bit `ACC_SYNCHRONIZED` est positionné dans les droits d'accès de la méthode à appeler.

Si c'est le cas la fonction `getLock()` est appelée sur l'objet de type `Class` qui représente la classe de la méthode à exécuter par l'instruction `invokestatic`. Pour les instructions `invokenonvirtual` et `invokevirtual` la demande de verrou se réalise sur l'objet qui se trouve sur la pile des opérandes.

Si le verrou est obtenu par le *thread* alors l'appel de la méthode est réalisé. Avant de continuer sur la nouvelle méthode, l'interpréteur sauvegarde l'objet qui a été verrouillé dans la *frame* de la méthode appelée. Cela permet de libérer l'objet lorsque la fonction exécutera une des instructions `return` ou qu'une exception non attrapée se propagera de méthode en méthode. Pour cela les instructions de type `return` et l'instruction `athrow` qui lance une exception vérifient si la *frame* courante contient un objet verrouillé et le libèrent automatiquement en appelant la fonction `releaseLock()`.

Si le verrou n'est pas obtenu parce qu'il est détenu par un autre *thread*, l'instruction se termine et l'interpréteur appelle l'ordonnanceur pour élire un autre *thread*.

7.4.4 Les méthodes de synchronisations de la classe `Object`

Les méthodes de synchronisations de la classe `Object` sont des méthodes natives qui étendent les possibilités de synchronisations des applications Java *multi-threads*. Ce sont les méthodes `wait()`, `notify()` et `notifyAll()`. Nous pouvons nous reporter à la section 7.4 pour une description complète de ce que réalisent ses méthodes. Nous détaillons uniquement dans cette section leurs implémentations dans notre machine virtuelle.

La méthode `wait()`

La méthode `wait()` ne peut être appelée que dans une méthode *synchronisée*. Cependant si l'appel de cette méthode est réalisé dans une méthode dont le verrou n'appartient pas au *thread* courant, notre machine virtuelle lance une exception de type `IllegalMonitorStateException`.

Notre machine virtuelle lorsqu'elle exécute cette méthode enlève le *thread* courant de sa liste de priorité et le place dans le listes des *threads* en attente d'un événement. Cette liste est contenue dans la structure `Sync` de l'objet (le champ `event`, voir la figure 7.3). Le champ `state` du *thread* est positionné à `TH_WAIT_EVENT`. Enfin le verrou est libéré afin qu'un autre *thread* s'exécute.

Si la méthode `wait()` est appelée avec un paramètre qui indique le temps d'attente du *thread*, la machine virtuelle doit positionner aussi le *thread* courant dans la liste des *threads* en sommeil.

Nous utilisons ici la même technique que pour un *thread* endormi. Une note de type `alarm` sera envoyée pour indiquer à la machine virtuelle qu'elle doit réveiller un *thread*. Si au bout d'un certain temps aucun autre *thread* n'a exécuté la méthode `notify()` sur cet objet, la note est reçue. Elle demande l'exécution de l'ordonnanceur qui commence par vérifier s'il y a des *threads* à réveiller. Elle doit trouver le *thread* qui à exécuté la méthode `wait()`. Il est donc dans l'état `TH_WAIT_EVENT`. Ce *thread* est replacé dans sa liste de priorité prêt à s'exécuter et est enlevé de la liste des *threads* qui attendaient sur l'objet. Il est positionné dans un état particulier qui est `TH_READY_LOCK`. Cet état indique à l'ordonnanceur que ce *thread* doit essayer de reverrouiller l'objet avant de continuer à s'exécuter (car il s'exécute dans une méthode *synchronisée*).

Les méthodes `notify()` et `notifyAll()`

La méthode `notify()` permet de prévenir un *thread* qui a exécuté la méthode `wait()` qu'il peut reprendre son exécution.

Pour cela la machine virtuelle passe le premier *thread* qui est en attente sur la liste `event` sur la liste des *threads* qui attendent que le verrou de cet objet ce libère (la liste `wait`). Quand le *thread* courant libérera le verrou, le *thread* qui vient d'être déplacé sera en position de reconquérir le verrou et de continuer son exécution.

Avant de rendre la main, la machine virtuelle vérifie si le *thread* qu'elle vient de changer de liste n'était pas aussi dans la liste des *threads* en sommeil (un *thread* qui à effectué un `wait` temporisé). Si c'est le cas elle l'enlève de cette liste puisque ce *thread* vient de recevoir un événement avant que son temps de sommeil ne soit écoulé.

La variante de la méthode `notify()`, la méthode `notifyAll()`, effectue exactement le même travail que la méthode `notify()` à part qu'elle passe tous les *threads* de la liste `event` sur la liste `wait`.

7.5 Résumé et remarques sur l'implémentation

Dans notre implémentation des *threads* pour notre machine virtuelle, nous avons décidé d'implémenter nous-même l'ordonnanceur. Notre machine quand elle exécute une application Java *multi-thread* est capable de créer des *threads* ainsi que d'ordonnancer leurs exécutions. Elle se comporte comme un système d'exploitation vis à vis des processus qu'il manipule.

L'ordonnanceur que nous avons choisi d'implémenter est de type FCFS pour les *threads* possédant la même priorité. Lors de l'exécution d'un *thread*, si un autre *thread* de plus haute priorité devient éligible, il est immédiatement sélectionné pour s'exécuter.

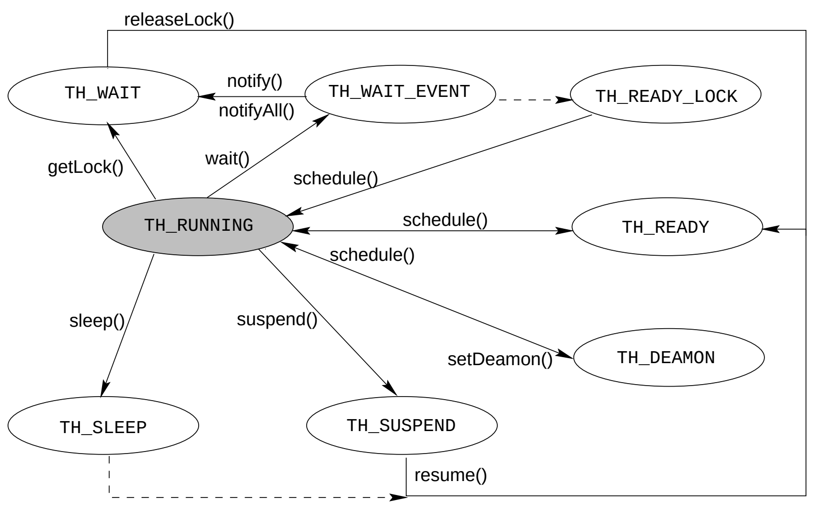

FIG. 7.4: Les transitions d'états des threads

Dans notre machine virtuelle un *thread* continue donc à s'exécuter jusqu'à ce que l'un des événements suivants se produise :

- Le *thread* décide de s'endormir ; lorsqu'il appelle la méthode `sleep()` de la classe `Thread` ou appelle la méthode `wait()` de la classe `Object`.
- Le *thread* attend pour la libération d'un verrou ; lorsqu'il veut exécuter une méthode *synchronisée* déjà utilisée par exemple.
- Le *thread* décide d'appeler la méthode `yield()` ; il laisse volontairement la main à un autre *thread* de même priorité.

- Le *thread* termine son exécution ; la méthode `run()` est finie ou il appelle la méthode `stop()`.
- Le *thread* est suspendu ; un autre *thread* exécute la méthode `suspend()` sur ce *thread.*
- Le *thread* change sa priorité ; il exécute la méthode `setPriority()`.
- Un nouveau *thread* de priorité supérieure est créé.

Par le respect de ces règles d'ordonnancement notre machine virtuelle respecte bien l'ensemble des spécifications Java.

Toutefois, il manque un mécanisme d'ordonnancement qui n'est pas pris en compte actuellement dans notre réalisation : les entrées-sorties bloquantes. En effet lorsqu'un *thread* réalise une opération d'entrée-sortie sur un descripteur de fichier par exemple et qu'il se retrouve bloqué, il devrait redonner la main à un autre *thread* en attendant que son opération devienne possible. Nous remarquerons que ce mécanisme devient vraiment utile lorsque l'on manipule des entrées-sorties via le réseau où les temps de réponses peuvent devenir importants.

Il faut aussi remarquer que l'implémentation d'un ordonnanceur à partage de temps serait certainement plus efficace pour l'exécution d'applications Java *multi-threads.* Nous avons préféré, durant le développement de notre machine virtuelle, opter pour un ordonnanceur plus simple à réaliser afin de pouvoir faire tourner plus rapidement des applications utilisant les *threads.* Néanmoins nous pensons pouvoir intégrer rapidement ce type d'ordonnanceur sur les bases de celui qui fonctionne actuellement.

Chapitre 8

Portage des librairies Java

Le langage Java fournit par défaut des classes qui constituent la librairie de base Java. Notre machine virtuelle doit supporter cette librairie pour exécuter du code Java standard. Bien que la librairie soit écrite en Java, il existe quelques méthodes (les méthodes natives) dont l'implémentation dépend entièrement du système d'exploitation où s'exécute la machine virtuelle. Nous avons donc implémenté ces méthodes en utilisant les spécificités du système Plan 9.

Nous décrivons dans un premier temps comment est réalisé l'appel des méthodes natives de la librairie. Nous poursuivons par une description des classes qui possèdent des méthodes natives et décrivons leur implémentation quand cela est nécessaire.

8.1 Les méthodes natives de Java

La librairie Java est organisée en trois *packages* : le package `java.io`, le package `java.lang` et le package `java.utils`. Ces packages peuvent être comparés aux librairies ou aux primitives d'intéraction fournies par d'autres langages de programmation. Ces librairies sont écrites principalement en Java.

Certaines méthodes, comme les méthodes qui manipulent les fichiers, utilisent des fonctions qui sont propres au système sur lequel s'exécute la machine virtuelle et par conséquent ne peuvent être écrites en Java directement. Ces fonctions sont déclarées avec le mot-clé `native` et ne possèdent pas d'implémentation Java.

8.1.1 Linkage des librairies Java

Les fonctions *natives* doivent être définies dans des librairies et *linkées* à la machine virtuelle. Dans les différentes implémentations de la machine virtuelle réalisées par Sun[1], les machines virtuelles supportent le chargement dynamique de librairies[2]. Les librairies de bases sont écrites sous forme de librairies partagées qui supportent le chargement dynamique.

Le système Plan 9 ne fournit pas le mécanisme de chargement dynamique de librairies. Nous avons donc écrit toutes les méthodes natives sous forme d'une librairie *linkée* statiquement à notre machine virtuelle.

8.1.2 Appels des méthodes natives

Les instructions qui permettent les appels de méthodes, utilisent une référence dans la *constant pool* pour retrouver la méthode dans la classe où elle est définie. Cette référence est une chaîne de caractères. Lorsque l'interpréteur décide d'exécuter une méthode *native*, il doit pouvoir faire correspondre au nom symbolique de la méthode, la fonction C qui représente son implémentation dans notre machine virtuelle.

Pour cela, nous avons mis en place une table de correspondances qui associe un nom symbolique au pointeur de la fonction C qui lui correspond. La structure C suivante représente un élément de notre table :

```
struct function {
    char *name;
    union t {
        void     (*v)(arg_list);
        int      (*i)(arg_list);
        vlong    (*l)(arg_list);
        float    (*f)(arg_list);
        double   (*d)(arg_list);
        Object * (*po)(arg_list);
    };
}
```

Le champ `name` représente le nom symbolique de la fonction. Il est composé du nom de la classe suivit du nom de la méthode. Par exemple, la fonction de la classe `java.lang.Math` qui permet de calculer un sinus est définie par `java/lang/Math.sin`.

L'union correspond au pointeur de fonction qui représente la fonction C dans notre machine virtuelle. Il existe un prototype de pointeur de fonction par valeur de retour possible de la méthode.

Lorsque la machine virtuelle décide d'exécuter une méthode *native*, elle effectue les étapes suivantes :

[1]Les versions 1.0 de la machine virtuelle Java de Sun pour les plateformes Solaris, Windows 95 et Windows NT.

[2]Ceci se retrouve notamment dans la classe `Runtime` de la librairie Java qui contient des méthodes qui permettent de demander le chargement dynamique d'une librairie particulière.

1. Le nom symbolique de la méthode est construit à partir des données qui se trouvent dans la table des symboles.
2. Les arguments de la méthode qui se trouvent sur la pile des opérandes sont dépilés et placés dans un tableau.
3. La machine virtuelle cherche linéairement dans sa table d'association l'entrée qui correspond au nom symbolique de la méthode [3].
4. La fonction C de cette entrée est exécutée avec comme paramètre le tableau qui contient les arguments.

Avant de commencer l'exécution à proprement parler de la fonction, les arguments sont décomposés à partir du tableau qu'elle reçoit en argument. Après l'exécution de la fonction, la valeur de retour est empilée sur la pile de la *frame* courante si la fonction n'a pas lancée d'exception. La main est ensuite rendue à l'interpréteur qui continue son exécution sur l'instruction suivant l'appel de la méthode native.

8.2 La librairie standard

Nous décrivons succinctement dans cette section les classes des différents packages qui ont nécessité une implémentation de méthodes natives. On trouvera dans l'annexe A toutes les méthodes que nous avons portées.

8.2.1 Le package `java.lang`

Le package `java.lang` contient les classes qui définissent la librairie de base du langage Java. On y retrouve par exemple, la classe `Object` qui définit le sommet de la hiérarchie des objets Java, la classe `System` dans laquelle on retrouve les *streams* qui permettent d'écrire ou de lire sur le terminal.

La classe `System`

La classe `System` possède trois méthodes natives. La première permet d'obtenir l'heure système en millisecondes[4]. La seconde effectue une copie de valeurs entre deux tableaux Java. La dernière initialise une liste de propriétés que définit la machine virtuelle par défaut. Cette liste contient notamment le numéro de version de la machine virtuelle, le nom du système d'exploitation sur lequel la machine s'exécute, le nom de l'utilisateur qui a lancé l'exécution de la machine virtuelle et le répertoire où elle s'exécute.

[3] Dans cette table il y a N entrées fixes, nous n'avons donc pas voulu compliquer la recherche et nous avons utilisé une simple recherche linéaire.

[4] Sur le système d'exploitation Plan 9, le nombre de millisecondes depuis le boot du système est obtenu en lisant le fichier `/env/msec`

La classe `Math`

La classe `Math` fournit au programmeur Java un ensemble de méthodes qui permettent de faire des calculs sur des valeurs flottantes double précision. On y retrouve les principales fonctions trigonométriques (sinus, arc sinus, cosinus, arc cosinus, tangente, arc tangente) ainsi que les méthodes de calculs d'une exponentielle et d'un logarithme. Ces méthodes sont implémentés en utilisant les fonctions mathématiques fournies par la librairie C du système Plan 9.

La classe `Thread`

L'implémentation des méthodes natives de la classe `Thread` sont décrites en détail dans le chapitre 7. Ce sont les méthodes qui permettent de créer et de manipuler les *threads*.

La classe `Object`

La classe `Object`, qui est la classe au sommet de la hiérarchie des classes Java, définit les méthodes natives `wait()` et `notify()` dont nous avons détaillé l'implémentation dans le chapitre 7. Elle définit aussi une méthode native `clone()` qui permet de dupliquer un objet. Pour cela, notre machine virtuelle créée une nouvelle structure `Object` et recopie simplement le contenu des variables de l'objet original.

Les classes `Float` et `Double`

Ces deux classes permettent d'encapsuler dans un objet Java les types primitifs `float` et `double`. Elles contiennent notamment la méthode native `toString` qui construit la représentation sous forme de chaîne de caractères de la valeur numérique que l'objet contient.

La classe `Throwable`

La classe `Throwable` contient les méthodes natives `fillInStackTrace` et `printStackTrace` que nous avons décrites au chapitre 6 sur la gestion des exceptions.

La classe `Class`

Cette classe définit plusieurs méthodes natives qui permettent de retrouver le nom symbolique de la classe d'un objet (la méthode `getClassname`) ou sa classe parente (la méthode `getSuperclass`). Elle permet aussi de retrouver le chargeur de classe qui a chargé la classe dans la machine virtuelle (la méthode `getClassLoader`). Cette dernière méthode n'est pas implémentée car elle ne supporte pas d'autre chargeur de classe que celui défini par la machine virtuelle (voir le chapitre 4).

La classe `Runtime`

La classe `Runtime` définit deux méthodes natives. La première, la méthode `exitInternal`, permet de quitter brutalement une application Java. Elle est implémentée en utilisant l'appel système `exits` du système Plan 9[5].

La seconde méthode, la méthode `execInternal`, permet de créer un processus qui est recouvert par un exécutable et exécuté par le système d'exploitation. L'implémentation de cette méthode dans notre machine virtuelle utilise l'appel système `rfork` pour créer un processus fils qui est ensuite recouvert par l'exécution de l'appel système `exec`.

8.2.2 Le package `java.io`

Le package `java.io` fournit un ensemble de classes qui permettent de manipuler les entrées-sorties en Java. Elles permettent de manipuler les canaux d'entrée, de sortie et d'erreur standard ainsi que les opérations sur les fichiers.

La classe `FileDescriptor`

La classe `FileDescriptor` représente les descripteurs de fichiers de Java. Elle possède deux méthodes natives : la méthode `initSystemFD` qui place un entier dans une variable de l'objet et la méthode `valid` qui permet de savoir si le descripteur de fichier d'un objet est valide.

La classe `File`

Cette classe est utilisée par les librairies Java pour représenter un fichier. Elle possède des méthodes natives qui permettent de manipuler les attributs d'un fichier, de créer des répertoires, d'effacer des fichiers ou de les renommer (voir le tableau 8.1).

Ces méthodes ont été implémentées sans grande difficulté en utilisant la librairie C de Plan 9 [6].

[5] L'appel système `exits()` de Plan9 est un peu différent de l'appel système `exit()` que l'on retrouve sur les systèmes Unix. Il renvoie une chaîne de caractères au shell au lieu de renvoyer un entier.

[6] Nous n'avons pas utilisé la librairie `stdio`, qui est là pour simplifier les portages, ni la librairie `Bio` mais les appels systèmes Plan 9.

`exist`	teste l'existence d'un fichier
`canRead`	teste si le fichier est accessible en lecture
`canWrite`	teste si le fichier est accessible en 'écriture
`isFile`	teste si c'est un fichier
`isDirectory`	teste si c'est un répertoire
`length`	donne la taille d'un fichier en octet
`mkdir`	créé un répertoire
`delete`	efface un fichier ou un répertoire
`renameTo`	renomme un fichier ou un répertoire
`isAbsolute`	teste si c'est un nom absolu
`list`	donne le contenu d'un répertoire
`lastModified`	donne la date de dernière modification

TAB. 8.1: Les méthodes de la classe `File`

La classe `RandomAccessFile`

La classe `RandomAccesFile` permet de manipuler le contenu des fichiers par accès direct ; les opérations peuvent se dérouler à n'importe quel endroit dans le fichier. Elle possède les méthodes qui permettent d'écrire un octet ou une suite d'octets, les méthodes qui permettent de lire dans le fichier et une méthode qui permet de déplacer un pointeur dans le fichier. Ces méthodes sont implémentées dans notre librairie en utilisant les appels systèmes `open`, `read`, `write` et `seek` de Plan 9.

La classe `FileInputStream`

Cette classe manipule des flux d'entrées. Elle peut être aussi bien associée à l'entrée standard qu'à un fichier. Les méthodes natives permettent principalement de lire sur ce flux des octets. Elle sont implémentées à l'aide de l'appel système `read`.

La classe `FileOutputStream`

Comme la classe précédente, la classe `FileOutputStream` manipule aussi bien la sortie standard qu'un flux de données dans un fichier. Les méthodes natives sont implémentées en utilisant l'appel système `write`.

8.2.3 Le package `java.util`

Le package `java.util` fournit des classes qui permettent de créer des listes, des tables d'adressages et des dictionnaires. Il contient aussi une classe qui manipule les dates. Cette dernière est la seule qui possède des méthodes natives, toutes les autres sont entièrement écrites en Java.

La classe `Date`

La classe `Date` permet de manipuler les dates. Cette classe définit cinq méthodes natives.

La méthode `expand` permet de déterminer une date à partir d'une valeur numérique qui représente le nombre de secondes écoulées depuis le 1^{er} Janvier 1970. Elle utilise pour cela la fonction de la librairie Plan 9 `localtime` qui retourne une structure `Tm` qui contient l'heure, le jour, le jour de la semaine, le mois et l'année.

La méthode `toString` retourne une date sous la forme d'une chaîne de caractères. Elle est implémentée en utilisant la fonction `ctime` de la librairie Plan 9 qui formatte la date comme les systèmes Unix.

La méthode `toLocalString` est identique à la méthode précédente, mis à part qu'elle retourne la chaîne de caractères en utilisant la fonction `asctime` de la librairie. La fonction `asctime` utilise la variable d'environnement `timezone`, pour déterminer dans quel fuseau horaire se trouve le système local, afin de formatter la date en conséquence.

La méthode `toGMTString` renvoie une chaîne Java qui donne l'heure GMT[7]. Elle utilise la fonction `gmtime` de la librairie Plan 9.

Enfin la méthode `computeValue` construit, à partir des données comme l'heure, le jour, le mois et l'année, une valeur numérique qui représente le nombre de secondes écoulées depuis le 1^{er} Janvier 1970. C'est la réciproque de la méthode `expand`. Cette méthode n'est pas implémentée actuellement dans notre machine virtuelle car il n'existe pas de fonction dans la librairie C Plan 9 qui permette cette transformation. Cette méthode n'est pas très utilisée et n'empêche pas la classe de fonctionner correctement.

8.3 Remarques sur l'implémentation des méthodes natives

Le portage des librairies n'est pas entièrement réalisé. La classe `ClassLoader` du package `java.lang` possède trois méthodes natives qui ne sont pas actuellement implémentées. Cette classe fournit les mécanismes de base qui permettent de créer un nouveau chargeur de classe. Elle n'est vraiment utile que pour les machines virtuelles qui s'exécutent sur des navigateurs Web.

Java fournit aussi un package destiné à créer des applications réseaux : le package `java.net`. Les méthodes natives des classes de ce package ne sont

[7]Le format de l'heure est donné par rapport au méridien de Greenwich.

pas implémentées actuellement.

Il faut noter que le portage a été possible car nous possédions les sources Java des librairies. Sans les sources Java de ses librairies, il n'est pas possible de retrouver les méthodes natives des classes. En effet, les méthodes natives sont généralement déclarées d'accès privé et n'apparaissent pas dans la documentation des classes qui accompagne les librairies. Elles sont cachées au programmeur Java.

Le portage des librairies Java que nous avons réalisé représente environ 2000 lignes de code C.

Chapitre 9

Analyse critique

Nous décrivons dans ce chapitre quelques applications Java que notre machine virtuelle est capable d'exécuter et analysons ses performances, en terme de rapidité, en comparant son exécution à la machine virtuelle développée par Sun.

9.1 Les applications tournant sur notre machine

Notre machine virtuelle est actuellement en mesure d'exécuter du code Java. Nous avons tout au long du développement créé beaucoup de petites applications Java afin de tester des parties bien précises de notre machine. Ces petites applications ne dépassent pas la vingtaine de ligne de code Java et ne nous permettent pas de tester le comportement de notre machine dans des cas d'utilisations plus avancés (des applications qui exécutent quelques milliers de lignes de code Java).

Nous avons pour cela récupéré sur l'internet des applications développées en Java par d'autres personnes. Les quatres applications que nous décrivons dans la suite de cette section utilisent toutes les mécanismes dont nous avons doté notre machine virtuelle et que nous avons abordés précédemment dans les chapitres 4 à 7.

9.1.1 Les classes d'exemples de Sun

Sun fournit un ensemble de petits programmes Java sur son serveur Internet qui permettent au développeur Java de trouver des exemples complets d'utilisation des classes de la librairie Java. A toutes les classes Java de la librairie sont associées un ou plusieurs programmes qui mettent en valeur l'utilisation d'une classe.

Cet ensemble de programmes, nous a aidé à mettre au point notre machine virtuelle. Le tableau 9.1 regroupe le nombre de classes et d'applications proposées par ce package.

Package	Nombre de classes	Nombre de programmes
`java.io`	26	46
`java.lang`	45	82
`java.util`	12	23

TAB. 9.1: Nombre d'applications fournit par les classes d'exemples de Sun

Toutes ces applications peuvent être exécutées par notre machine virtuelle mises à part celles qui utilisent la classe `ClassLoader` et la méthode `computeValue` de la classe `Date` (voir le chapitre 8 sur le portage des librairies).

9.1.2 Le compilateur Java de Sun

Notre machine virtuelle est capable de compiler du code source Java. Nous utilisons pour cela le compilateur Java de Sun. Ce compilateur est écrit en Java et est donc exécutable par notre machine. Il représente environ 25000 lignes de code Java sans aucune méthode native. Il n'a donc nécessité aucun portage.

Il représente pour nous le type même de grosse application Java qui nous permettent de dire que notre machine tourne correctement. Il est d'autant plus intéressant qu'il nous permet de compiler directement du code Java sur le système d'exploitation Plan 9.

Pour utiliser le compilateur, il suffit de taper sur la ligne de commande du shell :

```
term% run sun.tools.javac.Main -verbose mint.java
```

La commande `run` représente notre machine virtuelle. Elle reçoit en argument la classe qui lance le compilateur, la classe `Main` qui se trouve dans le package `sun.tools.javac`. Le compilateur Java prend le nom du fichier sur la ligne de commande, le fichier `mint.java`, et l'option `-verbose` qui permet de suivre les étapes de compilation du fichier source.

Le résultat de la compilation de l'exemple précédent est le suivant :

```
[parsed mint.java in 2880ms]
[loaded classes/classes.zip(java/lang/Object.class) in 450ms]
[checking class mint]
[loaded classes/classes.zip(java/lang/String.class) in 950ms]
[loaded classes/classes.zip(java/lang/System.class) in 630ms]
[loaded classes/classes.zip(java/io/PrintStream.class) in 490ms]
[loaded classes/classes.zip(java/io/FilterOutputStream.class) in 140ms]
[loaded classes/classes.zip(java/io/OutputStream.class) in 120ms]
[wrote mint.class]
[done in 9240ms]
```

Le compilateur affiche sur le terminal les différentes opérations qu'il réalise ainsi que le temps qu'il à mis pour les effectuer. Le fichier exécutable qui résulte de la compilation est `mint.class`. Ce fichier est bien évidemment exécutable par notre machine virtuelle.

Nous disposons avec l'exécution du compilateur Java par notre machine virtuelle d'un moyen de développer entièrement en Java sur le système Plan 9.

9.1.3 L'assembleur Java : Jasmin

Nous disposons pour notre machine virtuelle d'un assembleur Java. Cet assembleur, appelé Jasmin, a été écrit par John Meyer et sert de support au livre qu'il a publié sur la machine virtuelle Java [Meyer and Downing, 1997].

Jasmin est un assembleur Java qui prend en entrée la description d'une classe Java et fournit après assemblage un fichier compilé Java. Les fichiers à assembler utilisent le jeu d'instructions de la machine virtuelle défini par les spécifications Java. Il est entièrement écrit en Java. Pour une description complète de l'assembleur on pourra se reporter à [Meyer, 1997].

Le programme Jasmin suivant affiche la chaîne de caractères `Hello world` ! sur l'écran du terminal :

```
.class public HelloWorld
.super java/lang/Object

.method public <init>()V
   aload_0

   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
   .limit stack 2

   ; empile System.out sur la pile des opérandes
   getstatic java/lang/System/out Ljava/io/PrintStream;

   ; empile la chaine "Hello world!" sur la pile
   ldc "Hello World!"

   ; appel de la méthode PrintStream.println()
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

   ; fin
   return
.end method
```

La classe `helloWorld` définit uniquement deux méthodes. La première qui est l'initialisateur de la classe, la méthode `<init>`, appelle simplement l'initialisateur de sa classe parente. La seconde, la méthode `main`, commence par indiquer la taille de sa pile (`limit stack`). Elle empile ensuite la *stream* d'écriture sur le terminal et la référence sur la chaîne de caractères à afficher. Enfin, la méthode `println` de la classe `PrintStream` est appelée pour afficher la chaîne `Hello World!`.

L'utilisation d'un tel assembleur est utile aux curieux qui désirent approfondir leurs connaissances sur le langage Java ou au développeurs de machines virtuelles Java comme nous. Il est en fait bien pratique de pouvoir générer son propre *byte-code* afin de contrôler l'exécution de certaines instructions dans notre machine virtuelle.

On peut se procurer librement l'assembleur Jasmin sur le serveur Web de l'auteur à l'adresse `http ://www.cat.nyu.edu/meyer/jvm`.

9.1.4 L'interpréteur Tcl : Jacl

Jacl est un interpréteur Tcl [OusterHout, 1993] écrit entièrement en Java. Il supporte les fonctionnalités de la dernière version de Tcl 7.6/8.0. Il a été développé par à l'université Cornell [Lam and Smith, 1997].

L'exemple suivant montre une petite séance de travail avec l'interpréteur Tcl :

```
term% run cornell.Tcl.Shell
> proc power {b p} {
    set result 1
    while {$p > 0} {
        set result [expr $result * $b]
        set p [expr $p - 1]
    }
    return $result
}
> power 2 6
64
> power 1.15 5
2.01136
> ^D
term%
```

Il s'agit dans l'exemple précédent de définir une fonction qui permet de calculer b^p par la procédure `power`.

On peut retrouver l'interpréteur Tcl sur le serveur de l'université Cornell à l'adresse `http ://www.cs.cornell.edu/home/ioi/Jacl`.

9.2 Tests de vitesse comparés avec la machine de Sun

Nous présentons dans cette section des tests qui ont été effectués sur la machine virtuelle de Sun et celle que nous avons réalisée. Notre but est d'évaluer la vitesse d'exécution de programmes Java dans notre machine virtuelle.

Nous avons exécuté ces tests sur deux machines distinctes :

1. La première est un Pentium 100 doté de 16 Mo de mémoire vive. Elle tourne sur le système d'exploitation Plan 9 et utilise le réseau de machines Plan 9 du département MIME[1]. C'est sur cette machine que notre machine virtuelle Java s'exécute lors des tests.
2. La deuxième machine est une Pentium 100 doté de 32 Mo de mémoire vive. Elle tourne sur le système d'exploitation Linux[2]. Nous utilisons pour nos tests la machine virtuelle de Sun portée sur Linux par Randy Chapman [].

9.2.1 Détail des tests réalisés

Pour comparer les vitesses d'exécution des deux machines virtuelles, nous avons du choisir des applications Java sur lesquelles nos mesures devaient être réalisées. Notre choix s'est porté sur quatre applications différentes dont nous décrivons rapidement les caractéristiques.

Algorithme de recuit simulé

Nous voulions dans un premier temps tester notre machine sur une application qui réalise des calculs sur des valeurs flottantes de manière intensive. Nous avons choisi de réaliser une application qui résout le problème du voyageur de commerce[3] en utilisant une méthode de recuit simulé. Nous avons retranscrit cet algorithme en Java à partirt d'un programme C [Press et al., 1995].

Lors de l'exécution de cette application, nous utilisons une série de 10 villes. Le code source de cette application se trouve en annexe B.

[1] Le réseau Plan 9 de MIME est constitué d'un serveur de fichier, d'un serveur d'authentification et de plusieurs terminaux. Le serveur de fichier est un Pentium 100 doté de 16 Mo de mémoire, d'un disque IDE de 1.2 Go et le serveur d'authentification est un simple 486 avec 8 Mo de mémoire.

[2] Cette machine utilise la version 2.0.0 du noyaux Linux et la distribution Redhat 3.0.3 (Picasso).

[3] Le problème du voyageur de commerce revient à déterminer la plus petite tournée d'un commerçant qui doit passer par un série de n villes distinctes sans repasser deux fois par le même endroit.

Application utilisant les threads

La seconde application que nous testons utilise les threads Java. Elle utilise deux threads qui incrémentent une valeur entière de manière concurrente (voir `thread.java` dans l'annexe B). L'application teste donc l'ordonnancement et la synchronisation de threads de notre machine virtuelle.

Le compilateur Java de Sun

Il nous a parut intéressant de tester notre machine virtuelle sur le compilateur Java de Sun. C'est une grosse application qui met en œuvre tous les mécanismes que nous avons implémentés dans notre machine virtuelle.

Pour cela nous réalisons les tests sur un code source Java comprenant une vingtaine de lignes de code et sur un fichier source d'une application Java comprenant 450 lignes de code.

La série de tests de Symantec

Nous avons récupéré une série de tests réalisés par Symantec qui permet de comparer les vitesses d'exécution des machines virtuelles. Ces tests sont lancés les un à la suite des autres et sont décrits dans le tableau 9.2

Test	**Description**
BubbleSort	tri d'un tableau de 1000 entrées
BidirectionalBubble	variante du BubbleSort
QSort	tri d'un tableau de 5000 entrées
Sieve	calcul des nombres premiers de 0 à 4095 (1000 fois)
Hanoi	jeux de la tour de Hanoi
Fibbonacci	calcul de la suite de Fibonnacci de 0 à 1200 (1500 fois)
Array	manipulation sur un tableau de 10000 éléments
Tree	manipulation sur un arbre de 10000 éléments

TAB. 9.2: Les tests de Symantec

Le test de Symantec regroupe tous les types de données et les opérations qui peuvent être utilisés par notre machine virtuelle. On peut retrouver le code source de ces tests sur le serveur Web de Symantec à l'adresse `http ://cafe.symantec.com`.

9.2.2 Résultats et analyse

Nous avons exécuté les quatres programmes que nous venons de décrire en comptabilisant les temps d'exécution à l'aide de la commande `time` des systèmes Plan 9 et Linux. Sur ces deux systèmes la commande `time` permet d'obtenir les informations suivantes sur l'exécution d'un programme :

1. Le temps que le programme a passé en mode utilisateur. Cette valeur représente le temps passé à exécuter le code du programme.
2. Le temps que le programme a passé en mode système. Il représente le temps passé pour l'exécution des appels systèmes ; le code qui se trouve dans le noyau du système d'exploitation.
3. Le temps réel d'exécution du programme. Il représente le temps effectif d'exécution du programme tel qu'il est vu par l'utilisateur. Les systèmes Plan 9 et Linux étant des systèmes multi-processus, les programmes partagent leurs temps d'exécution sur le système et de ce fait la valeur réelle d'exécution est souvent supérieure à la somme du temps utilisateur et du temps système.

Résultats brutes

Nous présentons les mesures que nous avons effectuées dans le tableau 9.3. Il représente les temps d'exécution des différentes applications que nous avons évoquées dans la section précédente.

Tests	Plan 9				Linux			
	u	s	r	%CPU	u	s	r	%CPU
recuit simule	5.69	0.15	6.64	88	2.35	0.42	6.00	46
threads	19.39	0.10	20.05	97	11.87	2.65	23.47	61
compilation 1	6.89	0.75	12.38	62	4.87	0.29	5.95	86
compilation 2	40.33	1.91	63.12	67	12.77	0.59	20.93	63
symantec	88.81	0.35	94.86	94	35.24	0.75	42.42	84

TAB. 9.3: Résultats bruts des tests de rapidité

Les colonnes du tableau 9.3 intitulées `u`, `s` et `r` représentent respectivement les temps utilisateur, système et réel de la commande `time`. La colonne `%CPU` représente le rapport qu'il existe entre le temps réel d'exécution et la somme des temps utilisateur et système[4]. Les résultats obtenus lors de différentes exécution peut varier, ils dépendent de la charge du système ; la valeur `%CPU` reflète cette variation.

De ce fait, nous ne pouvons pas baser notre analyse sur les temps réels d'exécutions. Nous allons donc étudier les résultats en considérant uniquement les valeurs des temps utilisateur et système.

[4] Ce type de données n' est pas fournit par le commande `time` de Plan 9. Nous l'avons calculé de la façon suivante : $\frac{u+s}{r} * 100$

Analyse

Nous présentons dans le tableau 9.4 les mêmes données présentées différemment.

Tests	u + s Plan 9	u + s Linux	Rapports
threads	19.49	14.52	1.33
compilation 1	7.64	5.16	1.48
recuit simulé	5.84	2.77	2.10
symantec	89.16	35.99	2.47
compilation 2	42.24	13.66	3.16

TAB. 9.4: Rapports triés d'exécution entre notre machine et celle de Sun

La colonne **Rapports** du tableau 9.4 représente le rapport entre le temps d'exécution de notre machine virtuelle et la machine virtuelle de Sun. Ces rapports sont classés par ordre croissant.

Les données ainsi représentées, nous constatons que notre machine virtuelle qui tourne sur Plan 9 est de 1.3 fois à 3 fois plus lente que celle de Sun s'exécutant sur Linux.

Il est intéressant de remarquer que le temps d'exécution d'une application Java sur notre machine n'influe pas sur le rapport d'exécution. Ainsi, le test `compilation 2` qui s'exécute moins longtemps que le test de `symantec` possède un rapport à la machine de Sun plus important. Ceci nous force à penser que d'autres paramètres entre en jeu.

Nous avons pour cela calculé la taille maximum occupée par les applications de tests. Nous présentons dans le tableau 9.5 ces nouvelles données associées au rapport d'exécution de notre machine et celle de Sun.

Tests	Rapports	Occupation mémoire
threads	1.33	440 Ko
compilation 1	1.48	2072 Ko
recuit simulé	2.10	1192 Ko
symantec	2.47	3424 Ko
compilation 2	3.16	5500 Ko

TAB. 9.5: Rapports triés d'exécution associés à l'occupation mémoire

On constate maintenant que les temps d'exécution semblent être sensibles à l'occupation mémoire qu'occupe notre machine virtuelle. En effet, nous n'avons pas implémenté de *garbage collector* dans notre machine et donc les objets alloués ne sont jamais libérés.

Les applications qui consomment beaucoup de mémoire semblent ralentir l'exécution de notre machine virtuelle.

Il faut noter que ce dernier teste oblige le système Plan 9 à swapper de la mémoire vive sur le disque pour que l'application puisse s'exécuter[5].

Analyse à l'aide du *profiler*

La remarque précédente est confortée par les observations que nous avons effectuées avec le *profiler* du système Plan 9[6]. Il s'avère que la fonction `malloc`, qui permet d'allouer de la mémoire, se place dans les premières fonctions qui consomment du temps.

Nous constatons aussi que les fonctions qui interviennent dans le chargement des fichiers compilés Java sont gourmandes en temps d'exécution.

On trouvera en annexe C les résultats fournit par le *profiler*.

Analyse sur l'exécution des instructions Java

Enfin, nous avons modifié notre machine virtuelle pour qu'elle teste les temps d'exécution des différentes instructions Java interprétées. Cela nous permet de connaître le nombre de fois qu'une instruction a été exécutée et de connaître le temps moyen d'exécution d'une instruction.

A la vue des résultats que l'on peut consulter en annexe, les instructions suivantes possèdent un temps moyen élevé :

- L'instruction `new` qui permet la création d'un objet possède toujours le temps d'exécution moyen le plus important. La création d'un objet implique bien souvent le chargement, l'édition de liens et l'initialisation d'une ou plusieurs classes et l'allocation d'un espace mémoire.
- Les instructions `ldc` et `ldc_w` qui permettent la création d'un objet de type `String` à partir d'une chaîne de caractères située dans la *constant pool*.
- Les instructions d'appels de méthodes.
- Les instructions qui permettent de positionner ou de consulter les variables contenues dans les objets ; l'instruction `getfield` par exemple.

Là aussi nous constatons que les instructions qui impliquent une allocation mémoire et plus généralement les instructions qui créent les objets ralentissent l'exécution de notre machine virtuelle.

Une étude réalisée à l'université de Berkeley [Chaudhary et al., 1997] met en évidence que les applications Java ont tendance à créer beaucoup

[5]La zone de swap de la machine sur laquelle nous avons effectuée les tests se trouve actuellement sur le serveur de fichier du réseau des machines Plan 9. Ainsi lorsqu'un programme déclenche le mécanisme de swap, les performances du système s'effondre. Ceci explique aussi pourquoi on obtient de si mauvais résultats sur le test `compilation 2`

[6]Le profiler nous permet de comptabiliser le nombre de fois qu'une fonction a été appelée. De plus il les trie en fonction du temps pendant lequel elles se sont exécutées.

d'objets et que les performances des machines virtuelles sont fortement liées à l'architecture orientée objet de Java et de sa machine virtuelle. Ces conclusions semblent être vérifiées par nos résultats.

Ces remarques nous permettent de mettre en évidence les instructions sensibles de notre machine virtuelle. Parce qu'elles prennent du temps à s'exécuter, nous devons faire attention à leur implémentation afin d'améliorer les temps d'exécution de notre machine.

Tests et analyse sur des petits programmes

Nous avons remarqué que les tests précédents conduisent notre machine virtuelle à consommer beaucoup de mémoire (voir le tableau 9.5 page 128). Nous nous sommes donc demandé comment se comporte notre machine virtuelle sur de petits programmes Java qui ne créent pas beaucoup d'objets.

Pour cela nous avons effectué les mêmes mesures sur une série de petits programmes très variés. Ce sont ces derniers que nous avons utilisés durant tout le développement de notre machine virtuelle pour tester son intégrité.

Les mesures brutes sont présentés dans le tableau 9.6 suivant.

Tests	**Plan 9**				**Linux**			
	u	s	r	%CPU	u	s	r	%CPU
CMain	0.14	0.04	0.76	23	0.75	0.17	0.93	98
prop	0.21	0.06	1.08	25	0.81	0.17	1.00	98
Stat	0.22	0.16	1.19	32	0.80	0.26	1.08	90
WhatCase	0.15	0.07	0.82	27	0.76	0.17	0.93	100
pr	0.11	0.16	0.88	30	0.82	0.12	0.99	98
prr	0.07	0.06	0.68	19	0.75	0.18	0.93	100
obj	0.18	0.08	0.91	28	0.81	0.15	0.96	100

TAB. 9.6: Résultats bruts des tests de rapidité sur de petits programmes

Nous présentons dans le tableau 9.7 les rapports d'exécution à la machine virtuelle de Sun ainsi que la consommation mémoire de notre machine virtuelle lors de leurs exécution.

Nous constatons que les résultats sont nettement meilleurs que ceux que nous avons obtenus précédemment. Les rapports d'exécution à la machine virtuelle de Sun sont tous inférieurs à un. Cela signifie que notre machine virtuelle est plus rapide que la machine de Sun pour ce type d'applications. L'espace mémoire occupé par notre machine lors de l'exécution de ces derniers tests ne dépassent pas les 550 Ko.

Tests	Rapports	Occupation mémoire
CMain	0.19	472 Ko
prop	0.27	520 Ko
Stat	0.35	588 Ko
WhatCase	0.23	496 Ko
pr	0.28	516 Ko
prr	0.13	436 Ko
obj	0.27	524 Ko

TAB. 9.7: Rapports triés d'exécution associés à l'occupation mémoire

9.2.3 Conclusion

Les tests que nous avons effectués sont délicats à interpréter et ne mettent pas forcement en évidence tous les paramètres pertinents. De plus, le fait que les deux machines tournent sur des systèmes d'exploitations différents ne simplifie pas la tâche et rend toute conclusion délicate.

Au vu des derniers tests réalisés, notre machine virtuelle est plus rapide que la machine virtuelle de Sun sur des petits programmes qui ne consomment pas de mémoire, alors que les différences d'exécutions seraient de l'ordre de 30 à 40 pour cent pour des applications consommant un peu de mémoire et de l'ordre de 200 à 300 pour cent pour des applications consommant beaucoup de mémoire.

La consommation de mémoire est induite par la création des objets Java dans l'application. Il faut se souvenir que la création d'un objet entraîne bien souvent le chargement de une ou plusieurs classes et donc une augmentation de la consommation de la mémoire. Remarquons aussi qu'a l'aide du *profiler*, nous avons constaté que le temps d'exécution des fonctions qui interviennent dans le chargement des classes est important. [7]

Nous pensons que l'implémentation du *garbage collector* pourrait certainement améliorer les temps d'exécution de notre machine virtuelle. Il permettrait de libérer l'espace mémoire occupé par les objets qui ne sont plus utilisés et par conséquent limiterait la consommation mémoire de notre machine virtuelle.

[7] Ceci peut certainement s'expliquer par le fait que les fichiers du système Plan 9 sont accessibles sur un serveur de fichier qui se trouve sur le réseau. Or les tests sur la machine virtuelle de Sun on été réalisés sur une machine Linux qui possède tous ses fichiers sur le disque local et qui par conséquent retrouve ces fichiers plus rapidement que sur les machines Plan 9. Cette hypothèse est difficilement vérifiable mais nous pensons qu'elle influence aussi pour une bonne part nos résultats.

Chapitre 10

Conclusion

Nous venons de présenter ici l'implémentation de la machine virtuelle Java que nous avons réalisé pour le système d'exploitation Plan 9. Notre réalisation représente environ 7000 lignes de code C, plus 2000 lignes pour le portage des librairies Java de base.

10.1 Java, Plan 9, Inferno

Depuis août 1996, l'équipe des Bell Laboratories qui a développé Plan 9 distribue un système d'exploitation nommé Inferno, qui présente de nombreuses relations à la fois avec le système d'exploitation Plan 9 et avec Java : il s'agit à la fois d'une machine virtuelle, qui ressemble passablement à celle de Java[1], d'un langage de programmation *Limbo* destiné à être compilé pour cette machine qui s'inspire de C et de CSP et limite considérablement la liberté d'usage des pointeurs, et également d'un système d'exploitation qui ressemble fortement à Plan 9[2].

Les ambitions des concepteurs d'Inferno sont du même ordre que celles de Sun pour Java : offrir un environnement de programmation qui permet de développer des applications susceptibles de tourner sur tout type de matériel relié à l'Internet.

La similarité des environnements Plan 9 et Inferno nous laisse penser que si un compilateur C est développé pour Inferno, notre interprète sera facile à porter sous ce système ; il est amusant de constater que l'absence de

[1]Une différence essentielle de *Dis*, la machine virtuelle d'Inferno, avec celle de Java est que le garbage collecting est intégré directement dans l'architecture de la machine : un mécanisme de bas niveau utilise un compteur de référence dans tous les mots de la mémoire pour savoir s'ils sont encore référencés ; un algorithme classique, utilisé indépendamment permet de libérer les structures de données où les références sont cycliques.

[2]Dans la première distribution de *Kit de développement Inferno*, de nombreuses pages du manuel sont une copie conforme de celui de Plan 9.

garbage collector dans notre interprète ne sera plus un problème puisqu'il sera alors assuré par la machine virtuelle Dis elle-même.

Les concepteurs d'Inferno ont adopté pour l'intégration de Java une voie différente de la notre : ils se proposent de compiler le byte-code Java en Dis. Si ce projet est mené à bien, il permettra d'avoir dans un environnement qui ressemble à Plan 9 mais susceptible de tourner sur toute sorte d'architecture une interprétation du byte-code Java.

10.2 Les extensions nécessaires

Cette section présente les différentes extensions que nous pourrions réaliser rapidement pour compléter et améliorer notre machine virtuelle.

10.2.1 Entrées-sorties non-bloquantes

Notre ordonnanceur de *threads* respecte bien tous les mécanismes qui permettent de manipuler et de synchroniser les *threads* d'exécution de notre machine virtuelle. Cependant le mécanisme important, qui ne fait pas partie des méthodes de contrôle de la classe `Thread` et qui est relégué à la machine virtuelle, des entrées-sorties non-bloquantes n'est pas encore réalisée.

Quand un *thread* effectue une opération sur un descripteur de fichier qui ne peut être réalisée immédiatement par le système, la machine virtuelle stoppe son exécution attendant que le système satisfasse à sa requête. Tous les *threads* en attente d'exécution se trouvent bloqués et ne peuvent s'exécuter.

Il serait plus judicieux de stopper le *thread* bloqué afin de donner la main à un autre *thread* d'exécution, s'il en existe un en attente d'exécution. Le *thread* bloqué redeviendrait éligible dès que son opération serait possible.

Pour implémenter le mécanisme des entrées-sorties non-bloquantes dans notre machine virtuelle, nous pouvons utiliser les *threads* Plan 9. Un *thread* Plan 9, ordonnancé et contrôlé par le système, peut être créé pour réaliser l'opération d'entrée-sortie pendant que la machine virtuelle exécute un autre de ces *threads*. Une fois l'opération sur le descripteur de fichier accomplie, la machine virtuelle est prévenue et peut rendre de nouveau éligible le *thread* auparavant stoppé.

10.2.2 La librairie réseau

Le librairie réseaux Java, `java.net`, possède environ 30 fonctions natives qui doivent être portées pour permettre à notre machine virtuelle d'exécuter des applications réseaux. Cette librairie est basée sur le modèle des *sockets* Unix.

Le système Plan 9 simplifie ce modèle ; l'ouverture d'une connexion réseau sur une machine distante n'utilise qu'une opération pour récupérer le descripteur de fichier alors qu'au moins trois opérations sont nécessaires pour un modèle de *sockets* Unix. Par conséquent, certaines méthodes des classes réseaux doivent être laissées sans implémentation parce qu'elles n'ont pas d'équivalent sur le système Plan 9.

10.2.3 Le garbage collector

Nous avons souligné dans le chapitre 9, sur les tests de vitesse, que la gestion de la mémoire par notre machine semble influencer les résultats des tests de vitesse. De plus, il est assez gênant de se retrouver bloqué lors de l'exécution d'une application Java parce que l'espace mémoire utilisé par notre machine virtuelle est devenu trop important pour le système. Nous pensons donc que la réalisation d'un mécanisme de récupération mémoire est nécessaire.

La réalisation d'un type précis de *garbage collector* n'est pas imposée ni même recommandée par les spécifications Java, son implémentation est laissée à la discrétion des développeurs. On peut trouver de nombreux algorithmes de *garbage collector* dans [Jones, 1990].

10.2.4 Les instructions rapides

Si l'on regarde les instructions machine Java, nous pouvons constater qu'il n'y a qu'environ 160 instructions définies sur 256 possibles. Nous pouvons donc librement prendre avantage de cet espace libre pour ajouter de nouvelles instructions à notre machine virtuelle.

Nous pouvons implémenter par exemple un ensemble d'instructions qui permettent de réduire le nombre d'opérations que réalise une instruction après qu'elle ait été exécutée pour la première fois par notre machine virtuelle. Il faut dans ce cas effectuer un changement de code dans la méthode ; l'instruction Java est remplacée par sa variante qui peut contenir en paramètre les résultats de certaines opérations obtenues durant sa première exécution.

Cette technique de remplacement de code, qui est utilisée par la machine virtuelle de Sun, permet ainsi d'accélérer l'exécution des boucles et des méthodes appelées fréquemment.

10.3 Des extensions possibles

Nous présentons dans cette section ce qui à notre sens pourrait être réalisé à plus long terme pour enrichir et améliorer notre machine virtuelle Java.

10.3.1 La librairie graphique

Java fournit une librairie qui permet de manipuler des images et des objets graphiques comme des fenêtres, des menus déroulants ou des boites de dialogues. Les *applets* Java utilisent cette librairie.

Le portage de la librairie graphique de Java nécessite un gros travail puisqu'elle est basée, dans sa première implémentation par Sun, sur la librairie X11/Motif. Le système Plan 9 dispose d'une librairie, la librairie panel, qui pourrait certainement fournir une base de travail mais qui n'est pas assez complète pour implémenter tous les objets graphiques de la librairie Java.

Cependant un groupe de développeurs écrit actuellement une version de la librairie Java sans utiliser Motif. Ils développent en Java tous les objets graphiques fournis par la librairie Motif et elle est entièrement compatible avec la librairie de Sun. Le portage de cette librairie serait donc dans notre cas une meilleure solution.

Après un portage de la librairie graphique pour notre machine virtuelle rien ne nous empêcherait d'introduire notre interprète Java dans le navigateur Web de Plan 9 : mothra [].

10.3.2 Développement d'un compilateur

Le vitesse du code exécuté par un interpréteur Java ne peut pas rivaliser avec celle des exécutables générés par les compilateurs C ou C++. Pour améliorer les performances, on peut très bien imaginer de construire un compilateur qui, au lieu de produire du *byte code* Java, produirait du code natif. Les performances seraient certainement meilleures mais nous perdrions quelques caractéristiques comme la portabilité et la sécurité.

Une alternative serait de créer un compilateur incrémental qui transformerait le *byte code* Java en code natif à la volée. Les méthodes, avant d'être exécutées, seraient compilées en code natif, comme le font les compilateurs C et C++ par exemple, et ce dernier serait exécuté profitant ainsi de la pleine puissance du processeur.

Bibliographie

[Aho et al., 1991] Aho, A., Sethi, R., and Ullman, J. (1991). *COMPILATEURS Principes, techniques et outils.* Inter Edition.

[Chaudhary et al., 1997] Chaudhary, A., Chawathe, Y., and Wong, T. (1997). Study of the java instruction set architecture. http ://http.cs.berkeley.edu/ yatin/cs252/main.html.

[Flanagan, 1996] Flanagan, D. (1996). *JAVA in a nutshell.* O'Reilly.

[Flandrena, 1995] Flandrena, B. (1995). Plan 9 mkfiles. *Plan 9 Programmer's Manual*, volume 2.

[Floyd and Biegel, 1990] Floyd, R. and Biegel, R. (1990). *Le langage des machines.* International Publishing France.

[Gosling and McGilton, 1995] Gosling, J. and McGilton, H. (1995). The java language environment : A white paper. http ://java.sun.com/doc/language/whitepaper.ps.

[Jones, 1990] Jones, R. (1990). *Garbage Collection, Algorithms for Automatic Dynamic Memory Management.* John Willey & Sons, Ltd.

[Lam and Smith, 1997] Lam, I. and Smith, B. (1997). Jacl user guide.

[Lindholm and Yelin, 1996] Lindholm, T. and Yelin, F. (1996). *The Java Virtual Machine Specification.* Addison-Wesley.

[Meyer, 1997] Meyer, J. (1997). Jasmin user guide. http ://www.cat.nyu.edu/meyer/jvm/guide.html.

[Meyer and Downing, 1997] Meyer, J. and Downing, T. (1997). *JAVA Virtual Machine.* O'Reilly, first edition.

[OusterHout, 1993] OusterHout, J. K. (1993). *Tcl Language Syntax.* Adison-Wesley.

[Pike, 1995a] Pike, R. (1995a). 8 1/2, the plan9 window system. *Plan 9 Programmer's Manual*, volume 2.

[Pike, 1995b] Pike, R. (1995b). Acme : A user interface for programmer. *Plan 9 Programmer's Manual*, volume 2.

[Pike, 1995c] Pike, R. (1995c). How to use the plan 9 c compiler. *Plan 9 Programmer's Manual*, volume 2.

[Pike et al., 1995] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. (1995). Plan 9 from bell labs. *Plan 9 Programmer's Manual*, volume 2.

[Press et al., 1995] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1995). *Numerical Recipes in C.* Cambrige University Press.

[Reiser, 1991] Reiser, M. (1991). *The Oberon System.* Addison Wesley.

[Sedgewick, 1990] Sedgewick, R. (1990). *Algorithmes en langage C.* Inter Edition.

[Speer, 1990] Speer, E. (1990). *Gestion dynamique de la mémoire dans les langages de programmation.* Inter Edition.

[Stroustrup, 1992] Stroustrup, B. (1992). *Le langage C++.* Addison Weslay.

[Sun, 1996] Sun (1996). Hotjava(tm) : The security story. http ://javasoft.sun.com.

[Sun Microsystems, 1996] Sun Microsystems (1996). *The Java Language Specification.* Technical report, Sun Microsystems.

[Winterbottom, 1995a] Winterbottom, P. (1995a). Acid : A debugger built from a language. *Plan 9 Programmer's Manual*, volume 2.

[Winterbottom, 1995b] Winterbottom, P. (1995b). Acid manual. *Plan 9 Programmer's Manual*, volume 2.

Annexe A

Les fonctions natives de la librairie Java

Nous présentons ici les méthodes natives qui sont actuellement implémentées dans notre machine virtuelle.

Package	**Classe**	**Méthode**
java.lang	Thread	currentThread
java.lang	Thread	setPriority0
java.lang	Thread	start
java.lang	Thread	yield
java.lang	Thread	sleep
java.lang	Thread	stop0
java.lang	Thread	suspend0
java.lang	Thread	resume0
java.lang	Thread	isAlive
java.lang	System	currentTimeMillis
java.lang	System	arraycopy
java.lang	System	initProperties
java.lang	Math	sin
java.lang	Math	cos
java.lang	Math	tan
java.lang	Math	asin
java.lang	Math	acos
java.lang	Math	atan
java.lang	Math	exp
java.lang	Math	log
java.lang	Math	sqrt
java.lang	Math	IEEEremainder
java.lang	Math	ceil

Package	Classe	Méthode
java.lang	Math	floor
java.lang	Math	rint
java.lang	Math	atan2
java.lang	Math	pow
java.lang	Object	wait
java.lang	Object	notify
java.lang	Object	hashCode
java.lang	Object	clone
java.lang	Object	getClass
java.lang	Object	notifyAll
java.lang	Double	toString
java.lang	Double	valueOf
java.lang	Double	doubleToLongBits
java.lang	Double	longBitsToDouble
java.lang	Float	toString
java.lang	Float	valueOf
java.lang	Float	floatToIntBits
java.lang	Float	intBitsToFloat
java.lang	Throwable	fillInStackTrace
java.lang	Throwable	printStackTrace0
java.lang	Class	forName
java.lang	Class	newInstance
java.lang	Class	getSuperclass
java.lang	Class	getName
java.lang	Class	getInterfaces
java.lang	Class	isInterface
java.lang	Class	getClassLoader
java.lang	Runtime	exitInternal
java.lang	Runtime	execInternal
java.lang	Runtime	initializeLinkerInternal
java.lang	Runtime	buildLibName
java.lang	Runtime	loadFileInternal
java.io	FileDescriptor	initSystemFD
java.io	FileDescriptor	valid
java.io	RandomAccessFile	open
java.io	RandomAccessFile	read
java.io	RandomAccessFile	readBytes
java.io	RandomAccessFile	write
java.io	RandomAccessFile	writeBytes
java.io	RandomAccessFile	getFilePointer
java.io	RandomAccessFile	seek

Package	Classe	Méthode
java.io	RandomAccessFile	length
java.io	RandomAccessFile	close
java.io	File	exists0
java.io	File	canRead0
java.io	File	canWrite0
java.io	File	isFile0
java.io	File	isDirectory0
java.io	File	length0
java.io	File	mkdir0
java.io	File	delete0
java.io	File	renameTo0
java.io	File	isAbsolute
java.io	File	list0
java.io	File	lastModified0
java.io	FileInputStream	open
java.io	FileInputStream	read
java.io	FileInputStream	readBytes
java.io	FileInputStream	skip
java.io	FileInputStream	available
java.io	FileInputStream	close
java.io	FileOutputStream	open
java.io	FileOutputStream	write
java.io	FileOutputStream	writeBytes
java.io	FileOutputStream	close
java.util	Date	expand
java.util	Date	computeValue
java.util	Date	toString
java.util	Date	toLocalString
java.util	Date	toGMTString

Annexe B

Listing des tests de performances

Nous donnons ici le code source Java des applications que nous avons utilisé pour réaliser les tests du chapitre 9.

B.1 Recuit simulé

```
import java.applet.*;
import java.awt.*;
import java.util.*;


public class test {
     private static int N = 10;

     public static void main( String av[] ) {
      int i;
      double x[] = new double[N+1];
      double y[] = new double[N+1];
      int order[] = new int[N+1];
      Anneal an = new Anneal();

      x[1] = 6.2; y[1] = 10.9;
      x[2] = 60.3; y[2] = 67.3;
      x[3] = 50.5; y[3] = 43.7;
      x[4] = 15.6; y[4] = 09.3;
      x[5] = 23.4; y[5] = 51.5;
      x[6] = 122.2; y[6] = 78.9;
      x[7] = 88.5; y[7] = 61.3;
      x[8] = 15.5; y[8] = 67.8;
      x[9] = 34.9; y[9] = 111.3;
      x[10] = 48.6; y[10] = 28.7;
```

```
      for(i=1 ; i <= N ; i++)
           order[i] = i;


      System.out.print("\ncomputing ...\n\n");
      an.doAnneal(x, y, order, N);
      System.out.print("\n");

      for(i=1 ; i <= N ; i++)
           System.out.println("x[" + i + "] = " + x[i] +
                  " y[" + i + "] = " + y[i] +
                  " order[" + i + "] = " + order[i]);

      System.out.print("\nend.\n");
     }

}

import java.util.*;

public final class Anneal {
     private static double TFACTR = 0.9;

     Anneal() {
      /* ran3 init */
      iff = 0;
      ma = new long[56];

      /* metrop init */
      gljdum = new long[1];
      gljdum[0] = 1;
     }

     public void doAnneal( double x[], double y[], int iorder[], int ncity ) {
      int nover, nlimit, i1, i2, i, j, k, nsucc, nn, idec;
      int n[] = new int[7];
      boolean ans;
      double path, de, t;

      long idum[] = new long[1];
      idum[0] = -1;
      long iseed[] = new long[1];
      iseed[0] = 111;

      nover  = 100 * ncity;
      nlimit =  10 * ncity;
      path = 0.0;
      t    = 0.5;
```

```
for(i=1 ; i < ncity ; i += 1){
     i1 = iorder[i];
     i2 = iorder[i+1];
     path += ALEN(x[i1], x[i2], y[i1], y[i2]);
}
i1 = iorder[ncity];
i2 = iorder[1];
path += ALEN(x[i1], x[i2], y[i1], y[i2]);


for(j=1 ; j <= 100 ; j += 1){
     nsucc = 0;
     for(k=1 ; k <= nover ; k += 1){
      do{
       n[1] = 1 + (int)(ncity * ran3(idum));
       n[2] = 1 + (int)((ncity-1) * ran3(idum));
       if( n[2] >= n[1] )
            ++n[2];
       nn = 1 + ((n[1]-n[2]+ncity-1) % ncity);
      }while( nn < 3 );
      idec = irbit1(iseed);
      if( idec == 0 ){
       n[3] = n[2] + (int)(Math.abs(nn-2)*ran3(idum))+1;
       n[3] = 1 + ((n[3]-1) % ncity);
       de = trncst(x, y, iorder, ncity, n);
       ans = metrop(de, t);
       if( ans ){
            ++nsucc;
            path += de;
            trnspt(iorder, ncity, n);
       }
      }else{
       de = revcst(x, y, iorder, ncity, n);
       ans = metrop(de, t);
       if( ans ){
            ++nsucc;
            path += de;
            reverse(iorder, ncity, n);
       }
      }
      if( nsucc >= nlimit )
       break;
     }
     System.out.println("T ="+ t +"      Path Length ="+ path);
     System.out.println("Successful Moves: "+ nsucc);
     t *= TFACTR;
     if( nsucc == 0 )
      return;
```

```
    }

   }

   private static double ALEN(double a, double b, double c, double d) {

    return Math.sqrt(((b)-(a)) * ((b)-(a)) + ((d)-(c)) * ((d)-(c)));

   }

   private double revcst( double x[], double y[], int iorder[], int ncity, int n[] ) {
    double de;
    int j, ii;

    double xx[] = new double[5];
    double yy[] = new double[5];

    n[3] = 1 + ((n[1]+ncity-2) % ncity);
    n[4] = 1 + (n[2] % ncity);
    for(j=1 ; j <= 4 ; j += 1){
         ii = iorder[n[j]];
         xx[j] = x[ii];
         yy[j] = y[ii];
    }
    de = -ALEN(xx[1], xx[3], yy[1], yy[3]);
    de -= ALEN(xx[2], xx[4], yy[2], yy[4]);
    de += ALEN(xx[1], xx[4], yy[1], yy[4]);
    de += ALEN(xx[2], xx[3], yy[2], yy[3]);

    return de;
   }

   private void reverse( int iorder[], int ncity, int n[] ) {
    int nn, j, k, l, itmp;

    nn = (1 + ((n[2]-n[1]+ncity)) % ncity) / 2;
    for(j=1 ; j <= nn ; j++){
         k = 1 + ((n[1]+j-2) % ncity);
         l = 1 + ((n[2]-j+ncity) % ncity);
         itmp = iorder[k];
         iorder[k] = iorder[l];
         iorder[l] = itmp;
    }
   }

   private double trncst( double x[], double y[], int iorder[], int ncity, int n[] ) {
    double de;
    int j, ii;
```

```
 double xx[] = new double[7];
 double yy[] = new double[7];

 n[4] = 1 + (n[3] % ncity);
 n[5] = 1 + ((n[1]+ncity-2) % ncity);
 n[6] = 1 + (n[2] % ncity);
 for(j=1 ; j <= 6 ; j++){
      ii = iorder[n[j]];
      xx[j] = x[ii];
      yy[j] = y[ii];
 }
 de = -ALEN(xx[2], xx[6], yy[2], yy[6]);
 de -= ALEN(xx[1], xx[5], yy[1], yy[5]);
 de -= ALEN(xx[3], xx[4], yy[3], yy[4]);
 de += ALEN(xx[1], xx[3], yy[1], yy[3]);
 de += ALEN(xx[2], xx[4], yy[2], yy[4]);
 de += ALEN(xx[5], xx[6], yy[5], yy[6]);

 return de;
}

private void trnspt( int iorder[], int ncity, int n[] ) {
 int m1, m2, m3, nn, j, jj, jorder[];

 jorder = new int[ncity+1];

 m1 = 1 + ((n[2]-n[1]+ncity) % ncity);
 m2 = 1 + ((n[5]-n[4]+ncity) % ncity);
 m3 = 1 + ((n[3]-n[6]+ncity) % ncity);

 nn = 1;

 for(j=1 ; j <= m1 ; j++){
      jj = 1 + ((j+n[1]-2) % ncity);
      jorder[nn++] = iorder[jj];
 }
 for(j=1 ; j <= m2 ; j++){
      jj = 1 + ((j+n[4]-2) % ncity);
      jorder[nn++] = iorder[jj];
 }
 for(j=1 ; j <= m3 ; j++){
      jj = 1 + ((j+n[6]-2) % ncity);
      jorder[nn++] = iorder[jj];
 }
 for(j=1 ; j <= ncity ; j++)
      iorder[j] = jorder[j];

}
```

```
final int MBIG = 1000000000;
final int MSEED = 161803398;
final int MZ = 0;
final double FAC = (1.0/MBIG);

private long ma[];
private int inext, inextp;
private int iff;

private double ran3( long idum[] ) {
 long mj, mk;
 int i, ii, k;

 if( idum[0] < 0 || iff == 0 ){
      iff = 1;
      mj = MSEED - (idum[0] < 0 ? -idum[0] : idum[0]);
      mj %= MBIG;
      ma[55] = mj;
      mk = 1;
      for(i=1 ; i <= 54 ; i++){
       ii = (25 * i) % 5;
       ma[ii] = mk;
       mk = mj - mk;
       if(mk < MZ) mk += MBIG;
       mj = ma[ii];
      }
      for(k=1 ; k <= 4 ; k++)
       for(i=1 ; i <= 55 ; i++){
        ma[i] -= ma[1 + (i+30)%55];
        if(ma[i] < MZ) ma[i] += MBIG;
       }
      inext = 0;
      inextp = 31;
      idum[0] = 1;
 }
 if( ++inext == 56 ) inext = 1;
 if( ++inextp == 56 ) inextp = 1;
 mj = ma[inext] - ma[inextp];
 if(mj < MZ) mj += MBIG;
 ma[inext] = mj;

 return (double)mj * FAC;
}


private long gljdum[];

private boolean metrop( double de, double t ) {
```

```
      return de < 0.0 || ran3(gljdum) < Math.exp(-de/t);
     }

     final int TB1 = 1;
     final int TB2 = 2;
     final int TB5 = 16;
     final int TB18 = 131072;

     private int irbit1( long iseed[] ) {
      long newbit;

      newbit = (iseed[0]  & TB18) >> 17
           ^ (iseed[0]  & TB5) >> 4
            ^ (iseed[0]  & TB2) >> 1
             ^ (iseed[0]  & TB1);
      iseed[0] = (iseed[0] << 1) | newbit;

      return (int) newbit;
     }
}
```

B.2 Thread

```
class th5 extends Thread {
    static int i;
    int time;

    th5(String name, int t){
        super(name);
        time = t;
    }
    public void run(){
        for(int i=0 ; i < 100000 ; i++){
            inc();
            yield();
        }

    }
    public synchronized void inc(){
        i++;
    }

    public static void main(String av[]){
        th5 a = new th5("a", 100);
        th5 b = new th5("b", 10);

        a.start();
        b.start();
```

```
    }
}
```

Annexe C

Résultats complets des tests

Nous trouvons dans cet annexe les résultats numériques des tests que nous avons effectués dans le chapitre 9.

C.1 Tests sur l'exécution des instructions Java

La première colonne des listings que nous présentons contient le nombre de fois que l'instruction a été exécutée. La seconde colonne donne le temps total d'exécution et la troisième donne le temps moyen d'exécution (tous les temps sont donnés en millisecondes).

recuit simule :

```
 120649  5830 0.048 aload
 114504  5410 0.047 iload
 105026  5180 0.049 aload_0
  88017  6570 0.075 getfield
  70121  3150 0.045 iconst_1
  64560  2890 0.045 daload
  58371  2640 0.045 dload
  46279  2270 0.049 dsub
  45199  2080 0.046 iadd
  44819  2040 0.046 iaload
  38014  1840 0.048 aload_1
  37095  1840 0.050 laload
  37073  1800 0.049 bipush
  36592  1860 0.051 dmul
  34097  1820 0.053 lload_2
  25931  1590 0.061 iconst_2
  25004  3240 0.130 invokestatic
  23784  1030 0.043 iconst_0
  22038  1060 0.048 istore
  21520   820 0.038 dload_0
  21520  1200 0.056 dload_2
```

21520	1210	0.056	dastore
21458	1140	0.053	dreturn
19465	950	0.049	ldc2_w
18847	1390	0.074	putfield
18516	1040	0.056	dup
18490	1120	0.061	if_icmpne
18414	770	0.042	iconst_4
17857	680	0.038	dup_x1
17787	710	0.040	if_icmple
17443	930	0.053	lconst_0
17399	850	0.049	lcmp
16788	730	0.043	iconst_3
16514	750	0.045	iinc
16290	820	0.050	dadd
16235	600	0.037	iastore
15695	820	0.052	invokenonvirtual
14870	740	0.050	aload_3
14837	780	0.053	lstore_2
13903	690	0.050	isub
13463	540	0.040	iflt
13141	650	0.049	aload_2
13046	580	0.044	dstore
11380	600	0.053	lastore
10818	520	0.048	irem
10743	480	0.045	ifne
10039	570	0.057	ifge
9039	450	0.050	if_icmplt
8837	500	0.057	lsub
8562	340	0.040	l2d
8544	400	0.047	land
7732	300	0.039	iconst_5
6507	330	0.051	d2i
6507	350	0.054	i2d
6408	300	0.047	lxor
6408	250	0.039	lshr
6092	230	0.038	ireturn
4969	210	0.042	iload_1
4503	260	0.058	newarray
4428	230	0.052	astore
4217	270	0.064	dload_1
4191	220	0.052	dcmpg
4191	210	0.050	dneg
4157	250	0.060	iload_2
4122	170	0.041	ladd
3094	130	0.042	ifeq
2804	150	0.053	goto
2639	150	0.057	invokevirtual
2599	120	0.046	iload_0
2404	200	0.083	return

```
  2139   150 0.070 lconst_1
  2137    80 0.037 dconst_0
  2136   140 0.066 lshl
  2136   120 0.056 l2i
  2136    80 0.037 lor
  2055    70 0.034 ddiv
  2055   120 0.058 dload_3
  1629    80 0.049 dup2
  1525    80 0.052 iload_3
   908    40 0.044 arraylength
   683    30 0.044 bastore
   683    30 0.044 int2byte
   679     0 0.000 caload
   388    10 0.026 istore_2
   359    10 0.028 areturn
   188    20 0.106 lload
   158    10 0.063 ifnonnull
   143    30 0.210 ineg
   139     0 0.000 astore_3
   130   430 3.308 new
   106    10 0.094 astore_2
   100     0 0.000 if_icmpge
    97     0 0.000 istore_3
    85     0 0.000 imul
    81     0 0.000 lstore
    76   530 6.974 ldc
    73     0 0.000 idiv
    62     0 0.000 castore
    60    40 0.667 getstatic
    59     0 0.000 monitorexit
    59     0 0.000 monitorenter
    50     0 0.000 pop
    50     0 0.000 int2char
    50    10 0.200 istrore_0
    45     0 0.000 istore_1
    43     0 0.000 ishr
    30     0 0.000 putstatic
     3     0 0.000 sipush
     3     0 0.000 ifnull
     1     0 0.000 iconst_m1
     1     0 0.000 aastore
     1     0 0.000 anewarray
     1     0 0.000 lrem
     1     0 0.000 lneg
```

thread :

```
200117  7630 0.038 aload_0
200059 12270 0.061 return
200010 19790 0.099 invokestatic
200006  7900 0.039 iconst_1
```

```
200006  9770 0.049 getstatic
200006  9400 0.047 putstatic
200006 12650 0.063 invokevirtual
200006  8450 0.042 iload_1
200005  9560 0.048 ldc
200003 10580 0.053 iadd
200002 10510 0.053 if_icmplt
200000 10230 0.051 iinc
    49     0 0.000 invokenonvirtual
    46     0 0.000 putfield
    42     0 0.000 getfield
    33    10 0.303 aload_1
    19     0 0.000 iconst_0
    16     0 0.000 dup
    13   340 26.154 new
    10     0 0.000 aload_2
     8     0 0.000 newarray
     7     0 0.000 iload_2
     6     0 0.000 aload
     6     0 0.000 astore_2
     5     0 0.000 arraylength
     5     0 0.000 ifnonnull
     5     0 0.000 goto
     4     0 0.000 aconst_null
     4     0 0.000 istore_1
     3     0 0.000 ifnull
     3     0 0.000 aastore
     3     0 0.000 bipush
     3     0 0.000 sipush
     3     0 0.000 ifeq
     3     0 0.000 astore_1
     2     0 0.000 areturn
     2     0 0.000 astore
     2     0 0.000 if_icmpne
     2     0 0.000 aload_3
     1     0 0.000 iconst_4
     1     0 0.000 anewarray
     1     0 0.000 iconst_2
     1     0 0.000 iconst_m1
```

compilation 1 :

```
185275  8710 0.047 iload
163728  7230 0.044 aload_0
120050  8180 0.068 getfield
 92154  3950 0.043 iadd
 70830  3340 0.047 iload_1
 67593  2870 0.042 iinc
 66811  2970 0.044 aload
 56622  2570 0.045 iload_3
 53177  2420 0.046 aload_1
```

```
52108  2480 0.048 baload
49469  2060 0.042 iand
46881  2050 0.044 sipush
41742  2190 0.052 iload_2
41424  2020 0.049 bipush
40708  1940 0.048 ifgt
36119  2670 0.074 putfield
33970  1960 0.058 ireturn
33752  1590 0.047 istore
32191  2130 0.066 invokevirtual
30163  1590 0.053 caload
28253  1370 0.048 castore
26373  1100 0.042 int2char
26260  1080 0.041 if_icmplt
25335  1240 0.049 iconst_1
24553  1110 0.045 aload_3
23282  1100 0.047 istore_1
19645   970 0.049 dup
19102   950 0.050 aload_2
16166   700 0.043 return
15957   820 0.051 goto
15721   760 0.048 istore_2
14897   660 0.044 imul
14219  2280 0.160 invokestatic
13017  1520 0.117 invokenonvirtual
12647   600 0.047 isub
12639   490 0.039 dup_x1
12495   460 0.037 ishl
12476   480 0.038 ior
10798   480 0.044 astore
10148   480 0.047 if_icmpeq
 9837   470 0.048 iconst_0
 9744   310 0.032 ifge
 9671   420 0.043 ifne
 9365   450 0.048 areturn
 8079   380 0.047 iconst_4
 7363   210 0.029 ifnonnull
 7241   300 0.041 arraylength
 6332   370 0.058 if_icmpne
 6158   340 0.055 aaload
 5804   290 0.050 tableswitch
 5616   220 0.039 ifeq
 5356   350 0.065 istore_3
 5170   210 0.041 astore_3
 5121   520 0.102 invokeinterface
 5055  8670 1.715 new
 4582   230 0.050 ishr
 4580   260 0.057 if_icmple
 4463   170 0.038 ifnull
```

```
3587  1060 0.296 ldc
3371   120 0.036 astore_2
3221   130 0.040 astore_1
3196   140 0.044 iconst_2
3169   300 0.095 i2l
3135   100 0.032 checkcast
3065   150 0.049 lreturn
2863  2090 0.730 getstatic
2486   160 0.064 if_icmpge
2394   120 0.050 irem
2364    80 0.034 aconst_null
2234   110 0.049 newarray
2139   120 0.056 aastore
2033   120 0.059 lookupswitch
2011   100 0.050 iflt
1291    60 0.046 bastore
1283    60 0.047 int2byte
1202    30 0.025 int2short
1065   160 0.150 instanceof
1021    50 0.049 idiv
1013    70 0.069 lcmp
 975    30 0.031 pop
 949    70 0.074 ifle
 909    60 0.066 iconst_m1
 871    70 0.080 ladd
 609     0 0.000 if_acmpeq
 598    50 0.084 iload_0
 590    50 0.085 dup_x2
 466    10 0.021 lload_3
 444    20 0.045 if_icmpgt
 412    40 0.097 ldc2_w
 390    10 0.026 lastore
 389     0 0.000 lload_1
 276    10 0.036 lstore_3
 243     0 0.000 l2i
 224   860 3.839 anewarray
 223     0 0.000 monitorenter
 223     0 0.000 monitorexit
 210    10 0.048 lconst_0
 185    10 0.054 iushr
 168    10 0.060 lconst_1
 163    10 0.061 land
 138     0 0.000 lload_2
 137     0 0.000 if_acmpne
 108    20 0.185 putstatic
  82     0 0.000 laload
  82     0 0.000 lshr
  79     0 0.000 iconst_5
  75     0 0.000 iconst_3
```

```
    74     0 0.000 lshl
    64    10 0.156 iaload
    57    10 0.175 iastore
    54    10 0.185 fload_2
    36     0 0.000 lsub
    32    10 0.313 lor
    27     0 0.000 ixor
    25     0 0.000 ret
    25    10 0.400 jsr
    20    10 0.500 lstore_2
    18     0 0.000 i2f
    18     0 0.000 lmul
    18     0 0.000 fmul
    18     0 0.000 f2d
    18     0 0.000 dconst_0
    18    10 0.556 dcmpg
    18     0 0.000 f2i
     9     0 0.000 pop2
     9     0 0.000 ldiv
     9     0 0.000 lushr
     8     0 0.000 lstore
     7    20 2.857 ldc_w
     7     0 0.000 lload
     4     0 0.000 dup2_x1
     2    40 20.000 athrow
     1     0 0.000 lstore_1
     1     0 0.000 lload_0
```

compilation 2 :

```
916148 42170 0.046 aload_0
779395 61790 0.079 getfield
562916 26190 0.047 iload
278786 13650 0.049 iinc
273497 12750 0.047 aload
271581 13240 0.049 iload_1
258307 12600 0.049 iadd
231369 14350 0.062 invokevirtual
205325  9550 0.047 aload_1
198167 15460 0.078 putfield
187286  8240 0.044 iload_3
164398  8210 0.050 ireturn
164264  7900 0.048 caload
159581  7580 0.047 iload_2
126776  6500 0.051 istore
125861  5600 0.044 if_icmplt
123226  5720 0.046 iconst_1
115600  5760 0.050 bipush
113288  5300 0.047 aload_2
103701  4880 0.047 iand
102917  4380 0.043 dup
```

```
 102159  5180 0.051 istore_1
 100491  4570 0.045 aload_3
  92488  4360 0.047 baload
  92406  3890 0.042 sipush
  87232  4730 0.054 return
  86847  3690 0.042 goto
  79623  3650 0.046 ifgt
  72045  3630 0.050 areturn
  69995  3320 0.047 ifne
  69373  3200 0.046 dup_x1
  66591  3220 0.048 castore
  64993  2860 0.044 if_icmpeq
  64207  3010 0.047 if_icmpne
  62124  7290 0.117 invokenonvirtual
  61255  2940 0.048 istore_2
  54142  2660 0.049 astore
  51873  2790 0.054 arraylength
  51416  2390 0.046 iconst_0
  48810  2550 0.052 imul
  48769  2210 0.045 ifnonnull
  47014  2410 0.051 int2char
  40903  1780 0.044 ifeq
  35974  1780 0.049 ifge
  33605  1490 0.044 isub
  31667  1660 0.052 lookupswitch
  29979  4040 0.135 invokestatic
  28857  1580 0.055 tableswitch
  27692  1340 0.048 istore_3
  27588  1470 0.053 aaload
  24876  1410 0.057 astore_2
  24072  1250 0.052 astore_3
  22873  1080 0.047 ifnull
  22360  1140 0.051 iconst_4
  21884  2250 0.103 invokeinterface
  20569  1040 0.051 checkcast
  19963   960 0.048 ishl
  19877   960 0.048 ior
  19487  1000 0.051 ishr
  19460   880 0.045 if_icmple
  18385  9890 0.538 new
  15582   660 0.042 astore_1
  15263   700 0.046 iconst_m1
  14154  1450 0.102 ldc
  13204  2680 0.203 getstatic
  10763   470 0.044 irem
  10521   620 0.059 newarray
  10049   480 0.048 if_icmpge
   8823   510 0.058 aconst_null
   8673   450 0.052 bastore
```

8662	490	0.057	int2byte
8593	500	0.058	if_acmpeq
7830	520	0.066	instanceof
6956	320	0.046	iconst_2
6764	340	0.050	iflt
6592	240	0.036	aastore
4780	260	0.054	lreturn
4576	300	0.066	int2short
4499	210	0.047	i2l
4406	230	0.052	iload_0
4189	160	0.038	if_acmpne
4136	220	0.053	if_icmpgt
3895	180	0.046	pop
2800	70	0.025	idiv
2686	100	0.037	ifle
2653	80	0.030	lload_3
2541	120	0.047	dup_x2
2428	80	0.033	lcmp
2059	130	0.063	monitorenter
2059	130	0.063	monitorexit
1552	90	0.058	lload_1
1510	70	0.046	land
1505	120	0.080	iushr
1493	50	0.033	ladd
1484	80	0.054	lstore_3
1243	60	0.048	lconst_0
1201	90	0.075	lconst_1
1196	780	0.652	anewarray
841	20	0.024	ldc2_w
822	20	0.024	lload_2
775	90	0.116	l2i
712	50	0.070	lshr
712	30	0.042	laload
654	50	0.076	iconst_3
589	50	0.085	lshl
434	20	0.046	iaload
431	10	0.023	iconst_5
390	40	0.103	lastore
359	20	0.056	ixor
223	40	0.179	jsr
223	20	0.090	ret
183	10	0.055	fload_2
180	20	0.111	putstatic
169	0	0.000	lsub
157	0	0.000	lor
116	0	0.000	lstore
115	0	0.000	lload
61	0	0.000	dconst_0
61	0	0.000	i2f

```
        61      0 0.000 f2i
        61     10 0.164 f2d
        61     10 0.164 dcmpg
        61     10 0.164 fmul
        51      0 0.000 iastore
        49     10 0.204 lstore_2
        29      0 0.000 dup2_x1
        26     10 0.385 pop2
        18      0 0.000 lmul
        16      0 0.000 lushr
         9      0 0.000 ldiv
         7     10 1.429 ldc_w
         5      0 0.000 ineg
         2      0 0.000 lstore_1
         2     30 15.000 athrow
         1      0 0.000 lload_0
```

C.2 Résultats du *profiler* Plan 9

Les listings suivant présentent les données fournies par le *profiler* Plan 9 après l'exécution des applications.

recuit simule :

```
  %   Time Calls Name
97.1  6.970     5923 freadU2
 0.6  0.040    28938 memmove
 0.6  0.040    22282 freadU1
 0.4  0.030    28912 Bread
 0.3  0.020      456 qsorts
 0.1  0.010       22 findThread
 0.1  0.010      691 convU4
 0.1  0.010     2979 sysMalloc
 0.1  0.010      310 cp_resolution
 0.1  0.010      255 getSignatureFromCP
 0.1  0.010       43 lookupClass
 0.1  0.010       19 loadClassFile
 0.1  0.010    12882 strcmp
```

thread :

```
  %   Time Calls Name
19.4  5.850  2012576 strcmp
17.1  5.170       10 interpreter
 7.8  2.360   600102 hashkey
 6.9  2.090   200014 invokeNative
 6.8  2.060   400796 strconv
 6.4  1.920   600014 findThread
 4.3  1.300   600085 getClass
 3.3  1.010   200014 nativeCall
```

3.3	1.000	200407	doprint
3.0	0.910	400008	schedule
2.9	0.860	200051	newFrame
2.5	0.750	803606	malloc
2.1	0.630	802606	sysMalloc
2.1	0.630	200058	freeFrame
1.7	0.500	400100	getSignatureFromCP
1.6	0.480	200000	yieldThread
1.5	0.440	400796	sconv
1.5	0.440	800244	free
1.3	0.400	1000540	signatureSize
0.8	0.230	200005	releaseLock
0.7	0.210	200061	pushFrame
0.5	0.140	400008	checkAlarm
0.4	0.130	200407	sprint
0.4	0.120	200014	sign2type
0.4	0.110	200000	java_lang_Thread_yield
0.3	0.090	200000	addLastQueue
0.3	0.080	200003	rmQueue
0.2	0.070	200058	popFrame
0.2	0.060	200005	getLock
0.1	0.030	24484	memmove
0.1	0.020	24458	Bread

compilation 1 :

%	Time	Calls	Name
31.5	3.340	253	interpreter
26.0	2.760	158439	cp_resolution
9.2	0.970	849332	strcmp
2.8	0.300	164450	getSignatureFromCP
2.5	0.260	62396	newFrame
2.4	0.250	278831	freadU1
2.3	0.240	345499	Bread
2.2	0.230	136	loadClassFile
1.9	0.200	225594	sysMalloc
1.8	0.190	227854	malloc
1.8	0.190	347244	memmove
1.7	0.180	21612	strconv
1.5	0.160	487865	signatureSize
1.1	0.120	62646	freeFrame
1.0	0.110	136	preparation
0.9	0.100	15190	hashkey
0.8	0.090	1327	ReadCodeAttribute
0.8	0.080	5414	resolveVars
0.8	0.080	2231	invokeNative
0.7	0.070	10880	doprint
0.6	0.060	190533	free
0.6	0.060	61671	freadU2
0.6	0.060	71393	convU2

0.5	0.050	20146	strncpy
0.5	0.050	62649	pushFrame
0.5	0.050	15054	getClass
0.4	0.040	1621	java_lang_System_arraycopy
0.4	0.040	5193	newObject
0.3	0.030	13892	schedule
0.2	0.020	842	lookupClass
0.2	0.020	5155	getNameFromCP
0.2	0.020	62646	popFrame
0.2	0.020	16377	memset
0.2	0.020	10483	arrayLength
0.2	0.020	4035	sbrk
0.1	0.010	1058	realloc
0.1	0.010	975	strncmp
0.1	0.010	4861	convU4
0.1	0.010	21611	sconv
0.1	0.010	14835	releaseLock
0.1	0.010	456	qsorts
0.1	0.010	1390	strlen
0.1	0.010	13892	findThread
0.1	0.010	1837	swapi
0.1	0.010	375	newFrameFromName
0.1	0.010	2457	newArray
0.1	0.010	977	loadClass

compilation 2 :

%	Time	Calls	Name
38.4	24.300	983086	cp_resolution
27.4	17.340	254	interpreter
10.8	6.830	6254336	strcmp
3.4	2.140	1015793	getSignatureFromCP
2.5	1.560	4370214	signatureSize
2.2	1.370	328357	newFrame
1.7	1.100	1088154	malloc
1.3	0.830	1085800	sysMalloc
1.2	0.730	328608	freeFrame
1.1	0.710	17151	invokeNative
1.1	0.690	86106	strconv
0.8	0.520	1003378	free
0.7	0.470	77509	hashkey
0.6	0.410	328611	pushFrame
0.6	0.370	389693	Bread
0.5	0.340	77355	getClass
0.5	0.320	75978	schedule
0.5	0.290	328608	popFrame
0.4	0.270	43136	doprint
0.4	0.250	154	loadClassFile
0.4	0.240	401329	memmove
0.4	0.240	315629	freadU1

```
 0.4   0.240     18763 resolveVars
 0.3   0.170       154 preparation
 0.2   0.140     68606 freadU2
 0.2   0.130     17151 nativeCall
 0.2   0.130     21956 getNameFromCP
 0.2   0.100     82304 releaseLock
 0.2   0.100     82304 getLock
 0.2   0.100     65981 strncpy
 0.1   0.090     86105 sconv
 0.1   0.090      1461 ReadCodeAttribute
 0.1   0.080     75978 findThread
 0.1   0.070     21956 getClassNameFromCP
 0.1   0.070      9209 sbrk
 0.1   0.070     43136 sprint
 0.1   0.040     10745 newArray
 0.1   0.040     18541 newObject
```

symantec :

```
  %   Time Calls Name
70.8 102.400       44 interpreter
10.6  15.300   879660 cp_resolution
 7.4  10.640 18320780 popLong
 2.6   3.760  3665746 strcmp
 1.1   1.520   438572 newFrame
 1.0   1.430   906469 getSignatureFromCP
 0.8   1.190  1382472 malloc
 0.8   1.120  2758670 signatureSize
 0.7   1.040   438615 freeFrame
 0.7   1.020  3627032 _addv
 0.7   0.970  1381224 sysMalloc
 0.4   0.580  1822578 _gtv
 0.4   0.550  1821346 _ltv
 0.4   0.540   438616 pushFrame
 0.4   0.510  1316141 free
 0.3   0.480  1826345 _sl2v
 0.2   0.290   438615 popFrame
 0.1   0.210    10154 resolveVars
 0.1   0.130    27829 schedule
 0.1   0.130    81034 _vasop
 0.1   0.100    27000 lushr
 0.1   0.080    27956 releaseLock
```

Annexe D

Les instructions machine Java

Code opératoire	**Instruction**
0	nop
1	aconst_null
2	iconst_m1
3	iconst_0
4	iconst_1
5	iconst_2
6	iconst_3
7	iconst_4
8	iconst_5
9	lconst_0
10	lconst_1
11	fconst_0
12	fconst_1
13	fconst_2
14	dconst_0
15	dconst_1
16	bipush
17	sipush
18	ldc
19	ldc_w
20	ldc2_w
21	iload
22	lload
23	fload
24	dload

Code opératoire	Instruction
25	`aload`
26	`iload_0`
27	`iload_1`
28	`iload_2`
29	`iload_3`
30	`lload_0`
31	`lload_1`
32	`lload_2`
33	`lload_3`
34	`fload_0`
35	`fload_1`
36	`fload_2`
37	`fload_3`
38	`dload_0`
39	`dload_1`
40	`dload_2`
41	`dload_3`
42	`aload_0`
43	`aload_1`
44	`aload_2`
45	`aload_3`
46	`iaload`
47	`laload`
48	`faload`
49	`daload`
50	`aaload`
51	`baload`
52	`caload`
53	`saload`
54	`istore`
55	`lstore`
56	`fstore`
57	`dstore`
58	`astore`
59	`istrore_0`
60	`istore_1`
61	`istore_2`
62	`istore_3`
63	`lstore_0`
64	`lstore_1`
65	`lstore_2`

Code opératoire	**Instruction**
66	`lstore_3`
67	`fstore_0`
68	`fstore_1`
69	`fstore_2`
70	`fstore_3`
71	`dstore_0`
72	`dstore_1`
73	`dstore_2`
74	`dstore_3`
75	`astore_0`
76	`astore_1`
77	`astore_2`
78	`astore_3`
79	`iastore`
80	`lastore`
81	`fastore`
82	`dastore`
83	`aastore`
84	`bastore`
85	`castore`
86	`sastore`
87	`pop`
88	`pop2`
89	`dup`
90	`dup_x1`
91	`dup_x2`
92	`dup2`
93	`dup2_x1`
94	`dup2_x2`
95	`swap`
96	`iadd`
97	`ladd`
98	`fadd`
99	`dadd`
100	`isub`
101	`lsub`
102	`fsub`
103	`dsub`
104	`imul`
105	`lmul`
106	`fmul`

Code opératoire	Instruction
107	`dmul`
108	`idiv`
109	`ldiv`
110	`fdiv`
111	`ddiv`
112	`irem`
113	`lrem`
114	`frem`
115	`drem`
116	`ineg`
117	`lneg`
118	`fneg`
119	`dneg`
120	`ishl`
121	`lshl`
122	`ishr`
123	`lshr`
124	`iushr`
125	`lushr`
126	`iand`
127	`land`
128	`ior`
129	`lor`
130	`ixor`
131	`lxor`
132	`iinc`
133	`i2l`
134	`i2f`
135	`i2d`
136	`l2i`
137	`l2f`
138	`l2d`
139	`f2i`
140	`f2l`
141	`f2d`
142	`d2i`
143	`d2l`
144	`d2f`
145	`int2byte`
146	`int2char`
147	`int2short`

Code opératoire	Instruction
148	`lcmp`
149	`fcmpl`
150	`fcmpg`
151	`dcmpl`
152	`dcmpg`
153	`ifeq`
154	`ifne`
155	`iflt`
156	`ifge`
157	`ifgt`
158	`ifle`
159	`if_icmpeq`
160	`if_icmpne`
161	`if_icmplt`
162	`if_icmpge`
163	`if_icmpgt`
164	`if_icmple`
165	`if_acmpeq`
166	`if_acmpne`
167	`goto`
168	`jsr`
169	`ret`
170	`tableswitch`
171	`lookupswitch`
172	`ireturn`
173	`lreturn`
174	`freturn`
175	`dreturn`
176	`areturn`
177	`return`
178	`getstatic`
179	`putstatic`
180	`getfield`
181	`putfield`
182	`invokevirtual`
183	`invokenonvirtual`
184	`invokestatic`
185	`invokeinterface`
186	
187	`new`
188	`newarray`

Code opératoire	**Instruction**
189	`anewarray`
190	`arraylength`
191	`athrow`
192	`checkcast`
193	`instanceof`
194	`monitorenter`
195	`monitorexit`
196	`wide`
197	`multianewarray`
198	`ifnull`
199	`ifnonnull`
200	`goto_w`
201	`jsr_w`
202	`breakpoint`
209	`ret_w`