Virtualisation logicielle

De la machine réelle à la machine virtuelle abstraite

Bertil FOLLIOT et Gaël THOMAS

Introduction

Masquer l'hétérogénéité est un des grands challenges de l'informatique moderne : le nombre de configuration matériel est colossal et il est impossible de développer une application pour chacune de ces configurations spécifiques. La virtualisation logicielle apporte une réponse à ce problème en uniformisant l'accès au matériel, que ce soit l'accès au périphérique ou au processeur centrale. Deux domaines de l'informatique s'occupe de virtualisation : le domaine des systèmes d'exploitation s'occupe de masquer l'hétérogénéité des périphériques uniquement et le domaine des machines virtuelles s'occupe de masquer l'hétérogénéité des processeurs centraux.

1 Contexte de la virtualisation

Avec la naissance de l'informatique est né le besoin de factoriser des fonctionnalités communes à différentes applications et le besoin d'unifier l'accès au matériel. Le but de cette factorisation est double. D'une part, il s'agit de diminuer le travail de développement d'une application en réutilisant du code développé par d'autres acteurs du marché, d'autre part, il s'agit de développer une application une unique fois pour différents matériels. Cette factorisation de fonctionnalités a donné naissance à une couche logicielle qui s'interpose entre le matériel et l'application : le système d'exploitation. Un système d'exploitation virtualise les ressources matérielles pour l'application et offre une interface de programmation uniforme aux applications, indépendante du matériel présent dans l'ordinateur.

De nombreux acteurs du monde industriel et du monde académique se sont attaqués à la construction de systèmes d'exploitation : MSDos, Windows, OS/2, MacOS, Linux, FreeBSD, OpenBSD, Unix, Minix, L3, Chorus, Mach ou Solaris pour n'en citer que quelques-uns. Cette tentative d'uniformisation a donc simplifié le travail de développement puisqu'une application est développée pour un système d'exploitation et profite des abstractions fournies par ce système.

Toutefois, le problème n'est pas totalement résolu par les systèmes d'exploitation puisqu'une application ne peut pas s'exécuter sur tous les systèmes d'exploitation et sur tous les matériels : une application utilise l'interface de programmation d'un unique système d'exploitation et doit donc être modifiée pour pouvoir s'exécuter sur une autre système d'exploitation. De plus, pour des raisons de performances, les systèmes d'exploitation virtualisent les périphériques, mais ne virtualisent pas le processeur lui-même. Donc une application est écrite pour un unique processeur P et pour un unique système d'exploitation E.

La conséquence directe est qu'un utilisateur qui a besoin d'exécuter deux applications écrites pour deux systèmes différents doit posséder les deux systèmes. Comme deux systèmes d'exploitation ne peuvent pas cohabiter simultanément en mémoire, l'utilisateur n'a pas d'autre alternative que de redémarrer sa machine pour passer d'une application à l'autre ou d'avoir deux machines différentes chacune avec

son système d'exploitation. Si, en moyenne, un processeur peut exécuter n systèmes et qu'il existe m processeurs, il faut posséder n*m systèmes pour exécuter l'ensemble des applications possibles, et donc n*m machines.
Ce problème d'explosion combinatoire a donné naissance à un nouveau niveau de virtualisation : la virtualisation du processeur et la construction de machines virtuelles. Une machine virtuelle est une machine complète exécutée de façon logicielle. Une machine virtuelle est donc un programme qui exécute des applications virtuelles développées pour la machine virtuelle : elle traduit l'application virtuelle pour la machine concrète. Une machine virtuelle répond bien au problème de l'explosion combinatoire car une seule machine est suffisante pour exécuter l'ensemble des machines virtuelles existantes et donc l'ensemble des applications existantes, ce qui réduit drastiquement les coûts pour équiper une entreprise.
On définit donc deux processeurs : un processeur virtuel et un processeur concret. Le processeur virtuel est le processeur de la machine virtuelle. Il est implanté de façon logicielle et émule l'exécution d'une application virtuelle. Le processeur concret est le processeur matériel de la machine. Le rôle de la machine virtuelle est de transformer du code écrit pour un processeur virtuel en du code pour le processeur concret.
On distingue deux types principaux de machines virtuelles : (i) les machines virtuelles complètes, et (ii) les machines para-virtuelles. Une machine virtuelle complète, ou machine virtuelle, émule un processeur et l'ensemble de la machine. Elle permet d'exécuter toute application développée pour le processeur virtuel sur toute architecture concrète, pourvu qu'il existe une machine virtuelle pour cette architecture. Une machine para-virtuelle n'émule pas le processeur : elle exécute le code directement sur le processeur concret, mais elle permet de récupérer certaines instructions machines pour pouvoir les émuler. Le terme para-virtualisation vient du fait qu'une partie seulement du processeur est virtualisé. Les machines para-virtuelles évitent la transformation entre processeur virtuel et processeur concret, ce qui les rend nettement plus performantes. En contrepartie, elles ne peuvent exécuter que des applications virtuelles développées pour le processeur concret. Parmi les machines para-virtuelles, citons Xen, VMWare, Parallels ou Mol.
Dans la famille des machines virtuelles complète, on distingue souvent deux sous-familles : les machines virtuelles concrètes et les machines virtuelles abstraites. Une machine virtuelle concrète est une machine virtuelle qui exécute du code écrit pour une machine réelle et existante. Elle permet d'exécuter sur un processeur P et un système d'exploitation E une application développée pour un processeur existant P' et un système d'exploitation E'. Parmi les machines virtuelles concrètes, citons Qemu, Bochs, VirtualBox ou VirtualPC.
Une machine virtuelle abstraite est une machine virtuelle qui ne possède pas d'équivalent matériel. Elle définit un jeu d'instruction complet indépendant de toute réalité physique. Les machines virtuelles abstraites sont souvent livrées avec leur propre système d'exploitation indépendant. L'avantage des machines virtuelles abstraites et qu'elles n'ont pas à se conformer à une réalité matérielle et qu'elles peuvent donc offrir des macro-instructions machines beaucoup plus évoluées que ce qu'un processeur réel est capable de fournir. Elles permettent aussi d'assurer un haut niveau de sécurité du code exécuté en imposant que le code virtuel respecte des règles de sécurité. En revanche, le but d'une machine virtuelle abstraite n'est pas d'exécuter du code développé pour une autre architecture, mais de fournir un moyen élégant pour exécuter du code de façon uniforme sur n'importe quel couple processeur/système d'exploitation. Parmi les définitions de machines virtuelles

abstraites citons les JVM, les CLR (.Net), les interpréteurs GhostScript, Python, Perl ou Caml.
Aujourd'hui, il existe donc deux branches distinctes de l'informatique qui s'occupent de la virtualisation : les systèmes d'exploitation, présentés section 1 et les machines virtuelles (concrète, abstraite, ou para), présentées section 2. Ces deux domaines sont complémentaires et répondent à des problématiques différentes. Appréhender la virtualisation dans son ensemble demande de bien comprendre chacun de ces domaines.

2 Système d'exploitation

Le rôle d'un système d'exploitation est de virtualiser les périphériques et la mémoire d'une machine. Le but est d'offrir un accès uniforme aux ressources matérielles. Un système d'exploitation n'a pas pour rôle de virtualiser le processeur lui-même car son but est de maintenir de bonnes performances. En effet, exécuter nativement une application sur un processeur est toujours plus rapide que d'interpréter ou de re-compiler le code machine à l'exécution. De plus, les langages évolués masquent partiellement les différentes architectures de processeurs. Par exemple, un programme écrit en C peut être compilé pour différents processeurs, à condition qu'il existe un compilateur pour le processeur. Ainsi, le même programme en C pourra être compilé pour un processeur Pentium 32 bits ou AMD 32 bits, mais aussi pour un processeur PowerPC, Sparc, AMD 64 bits ou M68K. En revanche, l'accès au processeur est virtualisé par un système d'exploitation de façon à pouvoir simuler l'exécution de plusieurs applications indépendantes en même temps sur le même processeur.
Il existe un nombre colossal de périphériques et, pour chaque périphérique, différentes façons de s'y adresser. Les systèmes de fichier fournissent un bon exemple. Accéder à un fichier demande de connaître la nature du système de fichier (HFS, NTFS, ext3, reiserfs ou autre), mais aussi la façon de gérer les partitions sur le disque (MacOS et Windows gèrent différemment leurs tables des partitions) et enfin la façon de communiquer avec le contrôleur du disque (IDE, SCSI ou autre). Le système d'exploitation va donc fournir une vue virtuelle d'un système de fichier basée sur les notions de fichier, de répertoire et de chemin. Un programme va alors pouvoir ouvrir un fichier indépendamment du système de fichier utilisé.
Les cinq grandes classes d'éléments virtualisés par les systèmes d'exploitations sont :

- L'accès au processeur pour émuler plusieurs processeurs. Il ne faut pas confondre cette virtualisation avec celle du processeur lui-même.
- Les erreurs d'exécution signalées par le processeur.
- La mémoire pour donner un grand espace d'adressage à chaque processus et isoler les processus les uns des autres
- Les périphériques en mode caractère, c'est-à-dire les périphériques qui communiquent octet par octet.
- Les périphériques en mode bloc, c'est-à-dire les périphériques qui communiquent uniquement par ensembles d'octets.

Cette section n'a pas pour but de décrire de façon exhaustive un noyau de système d'exploitation, mais de montrer quels sont les mécanismes de virtualisation utilisés et quel est leur rôle. Le lecteur pourra se référer à des textes ne traitant que des systèmes d'exploitation pour approfondir le sujet [1][2][3][4].

2.1 Virtualisation de l'accès au processeur

Virtualiser l'accès au processeur permet d'émuler plusieurs processeurs sur la même machine pour exécuter simultanément plusieurs applications. Les systèmes qui virtualisent l'accès au processeur sont appelés des systèmes multi-tâches. Le but poursuivi est triple :

- Pour exécuter d'autres applications lorsqu'une application fait des entrées/sorties.
- Pour permettre l'utilisation d'une machine par plusieurs utilisateurs en parallèle.
- Pour fournir un mécanisme simple pour attendre des entrées/sorties sur plusieurs périphériques en même temps dans la même application

Chaque processeur émulé est identique à l'original et possède exactement le même jeu d'instruction. Dans ce sens, ce n'est pas le processeur qui est virtualisé mais bien son accès.

En revanche, pour que le système puisse assurer un minimum d'équité entre les processus, certaines instructions du processeur ne sont pas accessibles au niveau des applications. Par exemple, une application ne peut pas redéfinir comment le temps est géré et ne peut donc pas modifier l'interruption horloge. Dans un système multi-tâche, le processeur doit donc avoir au moins deux modes de fonctionnement : un mode privilégié où toutes les instructions peuvent être exécutées et un mode restreint où certaines instructions sont interdites. Le système d'exploitation s'exécute en mode privilégié et les applications en mode restreint.

2.1.1 Processus et commutation

Un processus est défini par un ensemble d'instruction et un état courant. L'ensemble d'instruction est le code du processus, c'est-à-dire la description de ce que doit exécuter le processus. L'état d'un processus consiste en l'état de sa mémoire et l'état des périphériques virtuels qu'il manipule. Un système d'exploitation mono-tâche ne peut exécuter qu'un seul processus de bout en bout avant d'en démarrer un nouveau, alors qu'un système multi-tâches permet d'entrelacer l'exécution de plusieurs processus. Un système multi-tâches est donc capable de gérer plusieurs états en parallèle.

Une machine possède un ou plusieurs processeurs. À chaque instant, un unique processus s'exécute sur un processeur. Un système multi-tâches va régulièrement changer les processus qui s'exécutent sur les processeurs et donner l'illusion que ces processus s'exécutent en parallèle. Le système va donc *commuter* des processus, c'est-à-dire qu'il va préserver l'état du processus qui s'exécute (processus sortant) et restaurer l'état d'un nouveau processus régulièrement (processus entrant). L'état d'un processus est constitué de l'état du processeur (les registres), de l'état de sa mémoire et de l'état de ses périphériques virtuels (fichiers ouverts, signaux…). Finalement, un processus est la virtualisation d'une machine complète avec son processeur, sa mémoire et ses périphériques.

Chaque processus est décrit dans le noyau du système d'exploitation par une structure de processus. Cette structure contient à la fois les périphériques virtuels associés au processus et l'état du processus (lorsqu'il est préservé). Le système maintient une table des processus : la table de tous les processus que doit exécuter le système d'exploitation.

Comme les périphériques sont virtualisés par le système d'exploitation, il est facile de multiplexer les périphériques réels vers des périphériques virtuels : chaque processus possède ses propres instances de périphériques virtuels et il n'est donc

pas nécessaire de les préserver lors d'une commutation. Par exemple, chaque processus possède ses propres fichiers ouverts et il n'y a donc aucune raison de les sauvegarder lors d'une commutation puisqu'ils sont déjà décrits dans la structure de processus. En revanche, la mémoire et le processeur ne sont pas virtualisés et il faut les préserver à chaque commutation.
Préserver l'état du processeur demande de stocker les registres du processeur dans la structure de processus. Les processeurs actuels possèdent entre une dizaine et une centaine de registres. Préserver et restaurer les registres n'est donc pas coûteux puisqu'il suffit d'exécuter au maximum une centaine d'instructions machines.
Préserver la mémoire devrait demander de sauvegarder toute la mémoire d'un processus à chaque commutation pour assurer l'isolation entre processus : de cette façon, on est assuré qu'un processus A ne va pas pouvoir modifier la mémoire d'un processus B. Ce mécanisme serait très lent. Heureusement, les processeurs récents virtualisent de façon matérielle la mémoire et permettent de donner l'illusion que chaque processus possède sa propre mémoire indépendante (voir section 2.3). Ainsi, préserver la mémoire et la restaurer lors d'une commutation n'a qu'un coût très limité puisqu'il va s'agir de préserver et de restaurer un unique registre qui indique comment la mémoire est virtualisée.

2.1.2 Virtualisation du temps

Deux grandes familles de systèmes multi-tâches existent :

- Les systèmes batch : un processus s'exécute de bout en bout, et n'est commuté que lorsqu'il fait une entrée/sortie et qu'il attend que cette entrée/sortie soit achevée.
- Les systèmes à temps partagé : un processus s'exécute pendant au maximum un quantum de temps avant d'être commuté. Comme dans un système batch il est également commuté en cas d'entrée/sortie bloquante.

Un système mono-tâche n'est autre qu'un système batch qui ne fait pas de commutation lors d'une entrée/sortie. Et un système batch n'est autre qu'un système à temps partagé dans lequel le quantum a une durée infinie.
Parmi les systèmes à temps partagé, on distingue souvent les systèmes temps réels des autres. Un système temps réel respecte des bornes temporelles imposées par les applications. Dans un système à temps partagé, aucun respect de bornes temporelles n'est imposé, en revanche, il faut que l'utilisateur ait l'illusion que les processus s'exécutent simultanément.
Dans un système multi-tâches à temps partagé, un processus s'exécute réellement par tranche de quantum, mais le temps est virtualisé et le processus a l'impression de s'exécuter en continu.
À l'expiration d'un quantum, le système doit choisir quel nouveau processus exécuter. C'est le rôle de l'ordonnanceur de choisir quel est le nouveau processus élu. Les choix du quantum et de l'algorithme d'ordonnancement sont liés. Un quantum est en général fixe, c'est-à-dire que tous les processus possèdent le même quantum et que ce quantum ne varie pas au cours du temps, mais certains algorithmes d'ordonnancement temps réel peuvent modifier le quantum pour assurer le respect de délais temporels. De nombreux algorithmes d'ordonnancement ont été proposés, du plus simple au plus complexe : la file d'attente FIFO aussi appelé tourniquet (round-robin), le tourniquet à priorités fixes, le tourniquet à priorités variables, le prochain délai en premier (EDF, Earliest Deadline First), plus petit délai en premier (RM, Rate Monotonic)…
Sur les machines de bureau, l'algorithme d'ordonnancement est en général un

tourniquet à priorités variables (pour les processus lorsqu'ils s'exécutent) couplé avec un tourniquet à priorité fixe (pour les processus lorsqu'ils se réveillent après une attente d'entrée/sortie).

2.2 Virtualisation des fautes matérielles

Une interruption ou une faute est un message envoyé par le matériel au processeur pour signaler un événement. Ce message est constitué d'un unique entier qui identifie la nature du message. Lorsque le processeur reçoit le message, il déroute son exécution pour exécuter un gestionnaire associé à l'événement. Concrètement, une table de fonctions se trouve en mémoire et lors de la réception de l'interruption ou de la faute n, le processeur appelle la fonction qui se trouve à l'entrée n de cette table. La réception d'une interruption ou d'une faute ne nécessite donc pas d'attente active de la part du processeur. Cette table est toujours gérée par le noyau du système d'exploitation pour éviter qu'un processus ne puisse s'approprier des périphériques et donc empêcher l'équité de l'accès aux ressources. Pour la même raison, les gestionnaires d'interruptions et de fautes se trouvent toujours dans le noyau.

Deux exemples classiques sont donnés par le clavier et la division par zéro. Lorsqu'un utilisateur presse une touche, une interruption est générée par le clavier. Le processeur exécute alors le gestionnaire de l'interruption clavier. L'exécution de ce gestionnaire fait basculer le processeur en mode privilégié. Le noyau récupère alors la touche pressée en communiquant directement avec le clavier (via un port d'entrée/sortie) et l'achemine jusqu'au périphérique virtuel associé à l'entrée standard du processus en cours d'exécution.

Lorsqu'un programme fait une division par zéro, le processeur détecte la faute et génère une faute. De la même façon, le processeur déroute l'exécution courante pour exécuter le gestionnaire de faute de division par zéro du noyau. Certaines fautes, comme la division par zéro, l'exécution d'une instruction illégale ou l'accès à une zone de mémoire interdite, doivent être signalées au processus qui a engendré la faute. Pour permettre au système de communiquer cette information, un mécanisme d'interruption virtuelle a été mis en place dans les systèmes d'exploitation : le signal.

Un signal est un message envoyé par un processus ou par le noyau à un autre processus. Ce message est constitué d'un entier qui identifie la nature du signal. De la même façon qu'une interruption, un signal déroute l'exécution du processus vers un gestionnaire de signal (en mode restreint) qui va traiter le signal. La figure Figure 1 - illustre le fonctionnement : un signal est la virtualisation pour un processus d'une faute. Le système délivre un signal correspondant à l'interruption d'origine au processus fautif, qui n'est autre que le processus actif sur le processeur qui reçoit la faute.

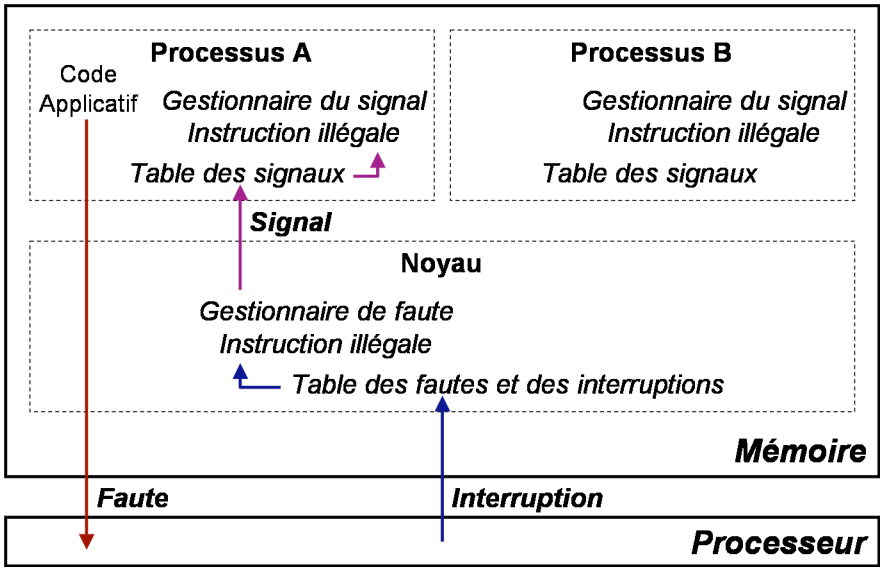

Figure 1 - Virtualisation des interruptions et des fautes

Les signaux permettent donc de multiplexer les interruptions au niveau des processus. Les signaux sont aussi utilisés comme système de communication entre processus : un processus peut envoyer un signal à un autre processus via des fonctions du système. On peut donc aussi voir qu'un processus émetteur se comporte comme un périphérique vis-à-vis du récepteur.

2.3 Virtualisation de la mémoire

Pour pouvoir exécuter plusieurs processus logés dans la même mémoire, il faut que le système s'assure que ces processus sont bien isolés les uns par rapport aux autres. En effet, que ce soit pour confiner les erreurs aux processus défaillants ou pour éviter des attaques de processus vers d'autres processus, il ne faut pas qu'un processus puisse accéder directement à la mémoire d'un autre processus. De la même façon, il faut aussi isoler la mémoire du noyau du système d'exploitation des processus pour éviter qu'un processus prenne le contrôle du noyau et, par transitivité, le contrôle de tous les processus puisque le noyau contrôle l'ensemble des processus.

La technique la plus simple pour isoler les processus entre eux et de ne laisser en mémoire qu'un unique processus à un instant donné (celui qui s'exécute) et de décharger les autres dans de la mémoire secondaire. Outre le coût prohibitif d'une telle solution, cette technique ne permet pas d'isoler le noyau des processus car des portions du noyau doivent toujours être présentes en mémoire, ne serait-ce que les gestionnaires d'interruptions.

Pour résoudre le problème, le noyau se repose sur un mécanisme matériel fournit par les processeurs : la mémoire virtuelle. La mémoire virtuelle permet d'offrir une vue virtuelle de la mémoire au processeur lui-même. La mémoire est séparée en pages de taille de fixe (4Ko ou 4Mo sur les processeurs IA32). Avant un accès en lecture ou écriture à une adresse dans la mémoire, le processeur convertit cette adresse en une adresse réelle (appelée adresse physique). Une table de correspondance en mémoire associe des pages virtuelles avec des pages physiques (parfois appelées cases mémoire). En plus de l'association adresse virtuelle/adresse physique, la table de correspondance associe des droits aux pages virtuelles : lecture seule, écriture et exécution. Comme cette table de correspondance associe des pages virtuelles et des pages physiques, elle est appelée table de pages.

Dans un système récent, chaque processus possède sa propre table des pages. Il ne peut donc accéder qu'aux pages physiques qui ont été associées dans sa table des pages. Un moyen simple pour assurer l'isolation entre processus est de ne jamais associer une page physique à deux processus. Toutefois, la mémoire est un support très efficace de communication entre processus car elle évite de passer par le noyau du système d'exploitation pour envoyer des données à un processus. Le noyau fournit donc des moyens pour partager certaines pages physiques entre processus. Ces techniques sont :

- Des segments de mémoire partagés tels que défini par POSIX ou les IPC System V. Ce sont des ensembles de pages physiques qui peuvent être associées à plusieurs processus. Rien n'empêche deux processus d'associer ces pages physiques à différentes adresses virtuelles,
- Des processus à mémoire partagée : ce sont les processus légers. Ils partagent toute leur mémoire et ne sont donc pas isolés les uns des autres. Les processus légers sont très souvent utilisés au sein de la même application pour pouvoir attendre des entrées/sorties sans bloquer le reste de l'application.

Une partie du noyau du système d'exploitation doit toujours être accessible en mémoire : la table des processus, la table des pages du processus en cours d'exécution, la table des interruptions et des fautes et les gestionnaires associés. Linux, par exemple, choisit de toujours laisser tout le système d'exploitation dans l'espace virtuel de tous les processus. Pour isoler le noyau des processus, la table des pages décrit aussi si les pages sont accessibles en mode privilégié ou en mode restreint.

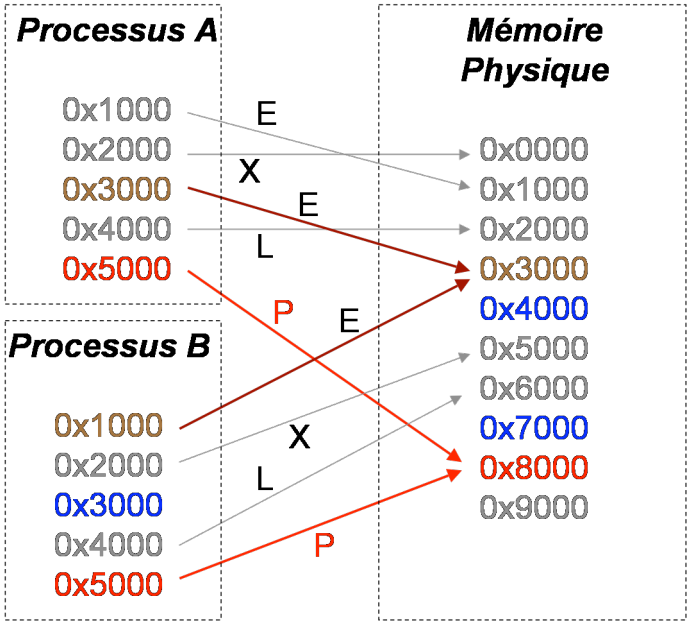

Figure 2 - Table des pages et virtualisation de la mémoire

La figure Figure 2 - décrit une table des pages classique. Deux processus A et B ont un espace d'adressage virtuel. Chaque page virtuelle de chaque processus est décrite dans la table des pages. Par exemple, à l'adresse virtuelle 0x1000 du processus A correspond l'adresse 0x1000 physique. Le droit associé à cette page est écriture (E : écriture, L : lecture, X : exécution et P : privilégié). Il faut noter que les pages physique 0x4000, 0x7000 et 0x9000 ne sont associées à aucun processus, c'est à dire qu'elles sont libres. A noter également qu'un processus n'a

pas forcément toutes ses pages virtuelles associées à des pages physiques. C'est le cas de la page virtuelle 0x3000 du processus B. Le noyau se trouve dans la page virtuelle 0x5000 du processus A et du processus B, mais ceux-ci ne peuvent pas y accéder car le droit est privilégié. Les pages virtuelle 0x3000 de A et 0x1000 de B désignent en fait la même page physique 0x3000 : il s'agit d'une page partagée. L'isolation est assurée par deux mécanismes. Premièrement, un processus ne peut pas modifier sa propre table des pages puisqu'elle se trouve dans le noyau et qu'il n'y a pas accès. Ensuite, un processus, B par exemple, ne peut pas accéder aux pages privées d'un autre processus, la page physique 0x1000 de A par exemple.
Ne pas associer des pages virtuelles à des pages physiques a deux rôles principaux :

- Donner un espace d'adressage virtuel plus grand que la quantité de mémoire physique disponible,
- Permettre de stocker une page virtuelle sur un autre support que la mémoire physique, en général dans le fichier d'échange (le swap). Ainsi, la mémoire virtuelle est plus grande que la mémoire physique disponible. La mémoire virtuelle peut donc virtualiser la mémoire physique, mais aussi le fichier d'échange.

La virtualisation de la mémoire pour les processus nécessite une collaboration avec le processeur : il faut que le processeur lui-même soit capable de virtualiser la mémoire. Toutefois, l'association entre une page physique et une page virtuelle est construite par le noyau, donc c'est aussi lui qui s'occupe de virtualiser la mémoire.

2.4 Virtualisation des entrées/sorties

Tous les systèmes d'exploitation introduisent la notion de fichier. Un fichier est un ensemble de données ordonnées qu'un processus peut consulter ou modifier. Historiquement, un fichier virtualise des données structurées sur un disque dur. Aujourd'hui, un fichier virtualise tout périphérique matériel ou logiciel permettant de faire des entrées/sorties, c'est-à-dire des lectures ou des écritures. Par exemple, la carte réseau est virtualisée par des fichiers spéciaux appelés socket (pour les protocoles basés sur IP), le clavier est virtualisé par un fichier qui ne permet que de faire des lectures.
Tout fichier appartient à un système de fichier virtuel. Un système de fichier virtuel masque l'hétérogénéité des systèmes de fichiers réels et des protocoles de communication avec les périphériques en définissant une interface d'accès commune. Un système de fichier virtuel est capable de manipuler des fichiers.
Un fichier est représenté en mémoire par une structure de fichier ouvert. Cette structure indique principalement la position actuelle dans le fichier et son mode d'ouverture (lecture, écriture). Un fichier ouvert possède aussi une référence vers un inode (appelé vnode sur BSD). Un inode décrit la structure du fichier (taille, date de création, propriétaire…) et possède un tableau de fonction permettant d'effectuer les lectures et écritures.
Le fichier ouvert et l'inode sont deux structures différentes : le fichier ouvert décrit l'état de lecture ou d'écriture du fichier alors que l'inode décrit le fichier lui-même. Plusieurs fichiers ouverts peuvent se partager le même inode.
On trouve, en général, cinq fonctions dans le tableau de fonction d'accès au fichier (appelé table des opérations sous Linux) :

- int open(String path) : cette fonction permet d'ouvrir un fichier identifié par le chemin path. Elle utilise le système de fichier virtuel pour construire un fichier virtuel en mémoire. Quand le fichier désigné correspond à un fichier réel d'un

système de fichier réel, cette fonction renvoie un fichier virtuel qui virtualise le fichier réel. Quand le fichier désigné correspond à un autre type de média, elle renvoie un fichier virtuel qui n'est qu'une abstraction permettant de faire des lectures et écritures sur le média. Cette fonction renvoie un identifiant de fichier ouvert.

- close(int file_descriptor) : cette fonction ferme un fichier ouvert.
- read(int file_descriptor) : cette fonction lit des octets à partir du fichier.
- write(int file_descriptor) : cette fonction écrit des octets dans le fichier.
- ioctl(configuration) : cette fonction permet de modifier la façon d'accéder aux données et dépend du média de communication.

La figure Figure 3 - présente le chaînage d'un processus jusqu'au périphérique. Chaque processus possède une table de descripteurs. Cette table des descripteurs contient des références vers des fichiers ouverts. Un descripteur de fichier manipulé par une application n'est autre qu'un index dans cette table. Plusieurs processus peuvent se partager le même fichier ouvert. Ce cas arrive après la duplication d'un processus (fonction Posix fork()). Par exemple, sur la figure, A et B se partagent f3.
Un fichier ouvert référence un inode virtuel (vnode). Plusieurs fichiers ouverts peuvent référencer le même inode car un inode représente le fichier alors que le fichier ouvert représente l'état d'accès au fichier. Ce cas peut arriver lorsque deux processus indépendant ouvrent le même fichier : ils possèdent chacun leur position courante dans le fichier et leur mode d'ouverture (lecture et/ou écriture). Sur la figure, les fichiers ouverts f3 et f4 se partagent l'inode i2.
Un inode possède une table des opérations d'entrée/sortie. Chaque inode va donc accéder au périphérique en utilisant le pilote de périphérique spécifique. Sur la figure, i1 est un inode qui représente un disque dur complet. Par exemple, i1 peut correspondre sous un Unix, à l'ouverture du fichier périphérique /dev/hda. L'inode i2 quant à lui correspond à un fichier classique d'un système de fichier local (ici, une partition reiserfs). Il pourrait, de la même façon, représenter un fichier distant (en utilisant le système de fichier nfs ou samba). L'inode i3 correspond à une socket, c'est-à-dire à un point de communication réseau. L'inode i4 correspond à un terminal, c'est-à-dire à un périphérique de sortie de la machine.
Cette figure ne décrit pas un système d'exploitation particulier et quelques variations peuvent exister entre deux systèmes d'exploitation. En revanche, cette figure représente un canevas commun à la plupart des systèmes d'exploitation.

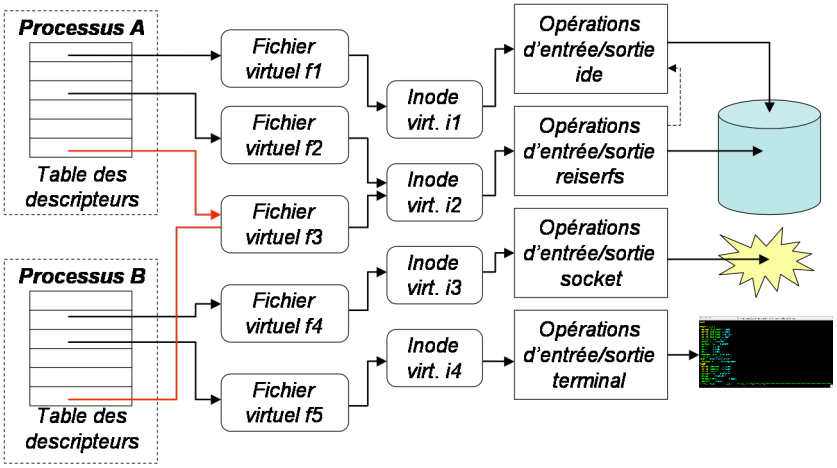

Figure 3 - Virtualisation des périphériques

La couche de virtualisation des périphériques permet d'unifier l'accès aux différents périphériques. Toutefois, les périphériques peuvent être très différents et cette couche de virtualisation peut ne pas être totalement satisfaisante. Prenons l'exemple d'un point de communication réseau en UDP : une socket. Une socket est nommée non par une chaine de caractère, mais par une adresse IP et un port. La fonction d'ouverture générique n'est donc pas suffisante pour les sockets et des fonctions annexes sont proposées par le système d'exploitation pour ouvrir une socket. Après ouverture, une socket est matérialisée dans le système par un fichier ouvert et un inode et les fonctions de lecture, écriture, fermeture et de contrôle peuvent être appliquée. Lors d'une communication en UDP, il est nécessaire d'indiquer quel est le destinataire du message, c'est-à-dire que lors d'une écriture sur le descripteur associé à la socket, il faut transmettre l'adresse du récepteur. Les fonctions d'opérations ne permettent pas de spécifier ce récepteur, donc une fonction annexe est proposée par le système d'exploitation (sendto), qui remplace la fonction d'écriture générique. Pour les sockets connectés, les fonctions de base de lecture et d'écriture peuvent toujours être invoquées puisqu'il n'est pas nécessaire de préciser l'adresse de l'émetteur ou du récepteur lors d'une communication.
Pour conclure, la couche de virtualisation des périphériques est relativement satisfaisante car elle abstrait les applications d'une connaissance précise des périphériques qu'elles manipulent, de la façon de stocker les données sur ce périphérique et de la façon d'accéder au périphérique (par bloc ou par caractère par exemple). Cette couche de virtualisation ne peut pas unifier toutes les particularités des périphériques et doit être complétée pour des besoins spécifiques, comme avec les sockets. Le nombre de cas spécifiques est relativement limité et les fichiers virtuels fournissent une abstraction suffisante dans la plupart des cas.

3 Machine virtuelle

Les systèmes d'exploitation fournissent une couche de virtualisation entre le matériel et les applications, mais comme un système d'exploitation ne virtualise pas le processeur, une application classique ne peut s'exécuter que sur un seul processeur et un seul système. L'essor des applications distribuées nécessite de faire interagir du code dans un environnement hétérogène, or de nombreuses applications nécessitent de migrer du code d'une machine à l'autre, que ce soit des applications web, comme des applets ou du javascript, des applications à base d'agents mobiles, ou encore des applications distribuées qui nécessitent de charger des souches d'appels pour des objets distribués. Comme du code compilé ne peut s'exécuter que sur un seul processeur et un seul système, il est impossible de faire migrer ce code dans un environnement hétérogène. Pour résoudre ce problème, une couche de virtualisation du processeur est introduite. Celle-ci peut émuler une machine réelle, c'est le cas des machines virtuelles concrètes (Qemu, Bochs, VirtualBox ou VirtualPC) qui permettent d'émuler un processeur existant sur un processeur quelconque et un système quelconque. La couche de virtualisation peut aussi émuler un processeur abstrait qui n'a pas d'équivalent matériel, c'est le cas des machines virtuelles abstraites (JVM, CLR, GhostScript, Python…).
On définit une machine virtuelle comme un processeur et un ensemble de périphériques exécutés de façon logicielle. Une machine virtuelle est donc un programme capable de charger et d'exécuter d'autres programmes. La machine virtuelle s'exécute sur une machine hôte. Les programmes exécutés par la machine virtuelle utilisent les périphériques et le jeu d'instruction de la machine virtuelle. Une machine virtuelle exécute donc des applications de manière indépendante de la

machine réelle et de ses caractéristiques physiques : processeur, système d'exploitation, mémoire ou périphériques. On définit aussi une application virtuelle comme une application développée pour une machine virtuelle.
Une machine hôte peut exécuter simultanément plusieurs machines virtuelles et donc exécuter plusieurs applications virtuelles. Dans le cas des machines virtuelles concrètes, une machine hôte peut exécuter en parallèle des applications développées pour différents systèmes et processeurs réels. Une machine virtuelle peut aussi être elle-même une machine hôte : un utilisateur peut tout à fait lancer une machine virtuelle Java au dessus d'un Linux/ia32 qui s'exécute dans une machine virtuelle VirtualPC au dessus d'un système MacOS sur un PowerPC. Dans ce cas, la machine virtuelle VirtualPC est à la fois une machine virtuelle vis-à-vis du PowerPC et une machine hôte vis-à-vis de la machine virtuelle Java.

Une machine virtuelle se comporte comme une machine réelle : elle doit donc charger des applications. On considère que la machine hôte comporte au moins une source de données en lecture pour pouvoir charger le code de l'application virtuelle. Cette source peut être une entrée réseau, un périphérique de stockage, un fichier ou encore un clavier. La source peut être physique (périphérique de stockage) ou une virtualisation d'une source physique (fichier).

Le fonctionnement général s'effectue en trois étapes :

- Chargement de l'application virtuelle dans la mémoire de l'application virtuelle. Cette étape charge une application virtuelle à partir de la source de données.
- Installation de l'application dans la machine virtuelle. À cette étape, la machine virtuelle prépare l'exécution de l'application virtuelle.
- Exécution de l'application virtuelle. Pendant cette étape, le code de l'application virtuelle est exécuté sur le processeur de la machine hôte.

3.1 Présentation de l'application virtuelle et installation

Une application virtuelle peut se présenter sous deux formes différentes : un programme source ou une représentation intermédiaire binaire (bytecode en anglais). Un programme source est écrit dans un langage de haut niveau et nécessite une phase de transformation pour reconnaître les expressions du langage. Techniquement, le programme source doit au moins passer par une phase d'analyse lexicale et une phase d'analyse syntaxique pendant l'étape d'installation de l'application virtuelle dans la machine virtuelle. Un bytecode est directement un code exécutable par un processeur virtuel ce qui évite les phases d'analyse lexicale et syntaxique pendant l'étape d'installation. De plus, le bytecode peut être annoté pour optimiser la vérification de certaines propriétés lors de l'installation de l'application virtuelle. Bien souvent, le bytecode possède de nombreuses informations sur le type des entités manipulées qui sont utilisées par la machine virtuelle.
Les deux types de machine virtuelles existent : une machine virtuelle Scheme prend en entrée un programme source Scheme alors qu'une machine virtuelle Java prend en entrée un bytecode Java. Dans le cas des machines virtuelles concrètes, elles ne prennent en entrée que du bytecode : le code machine du processeur concret émulé par la machine virtuelle.
Qu'une application virtuelle se présente sous la forme d'un programme source ou d'un bytecode, la machine virtuelle va posséder une représentation intermédiaire de l'application virtuelle. Une représentation intermédiaire est une représentation en mémoire de l'application virtuelle telle que le travail à effectuer par la machine

virtuelle pour exécuter l'application virtuelle soit facilité. Cette notion est bien sûr suggestive : on dira que le bytecode est une meilleure représentation intermédiaire qu'un programme source car le nombre d'étape pour exécuter l'application virtuelle est plus petit. Des machines virtuelles peuvent aussi considérer que la représentation sous forme de bytecode n'est pas encore une bonne représentation intermédiaire et peuvent utiliser une représentation intermédiaire enrichie, voire différente. Nous considérerons dans la suite que la représentation intermédiaire constitue le code virtuel de l'application : c'est le code que doit exécuter le processeur virtuel de la machine virtuelle.
À la fin de l'étape d'installation, la machine virtuelle a construit en mémoire une représentation intermédiaire qui lui semble satisfaisante. Du point de vue d'un compilateur, cette étape d'installation de l'application virtuelle correspond à la partie avant du compilateur (avec l'analyse sémantique).

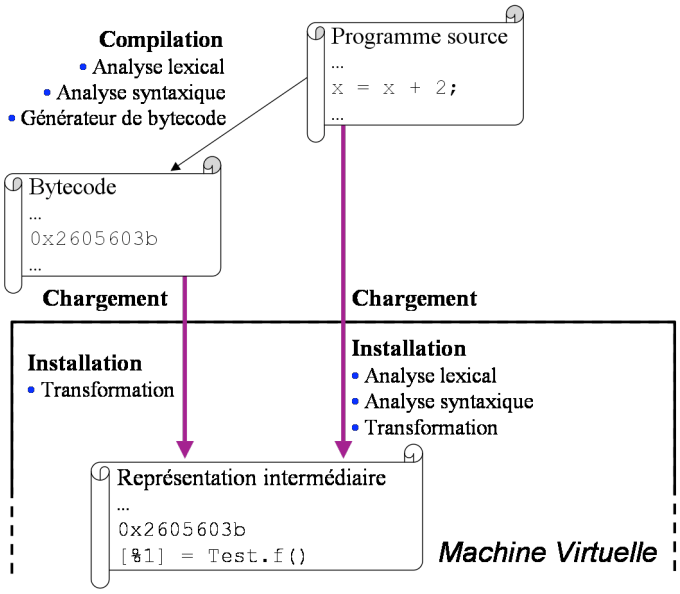

Figure 4 - Installation d'une application virtuelle

La figure Figure 4 - présente l'installation de l'application virtuelle dans les deux cas. À droite, les étapes de traitement du fichier source sont effectuées dans la machine virtuelle, alors qu'à gauche, les étapes de traitement du source sont effectuées en dehors de la machine virtuelle par un compilateur. Ce compilateur d'un langage source vers un bytecode (ne pas confondre avec l'appellation classique d'un compilateur, qui compile du code source vers le code natif d'un processeur donné), n'a besoin d'être écrit qu'une seule fois, et n'a pas besoin d'être présent sur les machines qui exécutent la machine virtuelle, puisque cette dernière ne chargera que le bytecode produit.

3.2 Exécution d'une application virtuelle

Une fois l'application virtuelle chargée, l'étape suivante consiste à exécuter le code virtuel. Il existe deux manières d'exécuter ce code :

- La machine virtuelle interprète chaque instruction virtuelle, ce qui conduit à exécuter un certain nombre d'instructions du processeur.
- La machine virtuelle compile l'application virtuelle vers du code natif puis exécute ce code. Il ne faut pas confondre cette étape de compilation de la représentation intermédiaire vers du code natif avec l'étape de compilation du

programme source vers le bytecode.

Les différences entre ces deux solutions concernent principalement les performances à l'exécution. La figure Figure 5 - présente les deux solutions. Soit la représentation intermédiaire est interprétée, soit elle est transformée en code natif via une phase de compilation puis exécutée ensuite.

3.2.1 Interprétation

L'interprétation entraîne un temps de réponse multiplié par un facteur 100 par rapport à du code natif car l'exécution de chaque instruction virtuelle entraîne l'exécution de nombreuses instructions de la machine hôte. Une instruction pour lire l'instruction virtuelle courante, une instruction pour sélectionner la fonction capable d'interpréter cette instruction virtuelle et l'exécution de cette fonction qui peut elle-même être assez lourde.

3.2.2 Compilation à la volée

La compilation de la représentation intermédiaire vers du code natif entraîne un temps de latence qui peut être important puisqu'il dépend du nombre d'instructions virtuelles. Le temps de réponse est en revanche nettement amélioré puisque du code natif est exécuté. Lors des exécutions suivantes, la représentation intermédiaire n'a plus besoin d'être compilée : le code natif est directement invoqué. Le temps de latence est supprimé et le temps de réponse est optimal. Plus le code natif compilé est invoqué souvent, plus il devient rentable de faire de la compilation à la volée. En effet, compiler du code à la volée et l'exécuter une unique fois est presque toujours plus lent que d'interpréter les instructions virtuelles. Comme exécuter le code compilé est plus rapide à l'exécution que d'interpréter le code virtuel d'un facteur 100, le temps de compilation à la volée est rapide à amortir.
Un compromis existe entre les deux solutions précédentes : si du code est peu invoqué, il est interprété, sinon, il est compilé à la volée. En pratique, la code est compilé à la volée dès la seconde invocation dans la plupart des machines virtuelles qui utilisent cette technique.

Avec un compilateur à la volée se pose la question de l'instant de compilation et de la granularité. Dans le cas spécifique où le compilateur à la volée compile l'application par partie (en général par méthode) et qu'il ne compile ces parties qu'au moment de les exécuter (l'instruction virtuelle suivante est la première instruction virtuelle de la partie), on dit que le compilateur à la volée est un compilateur juste à temps ("JIT-C" : Just In Time Compiler). Dans la pratique on confond souvent le cas particulier du compilateur juste à temps et le cas général du compilateur à la volée qui compile du code qui va être exécuté sans être stocké sur un support persistant.

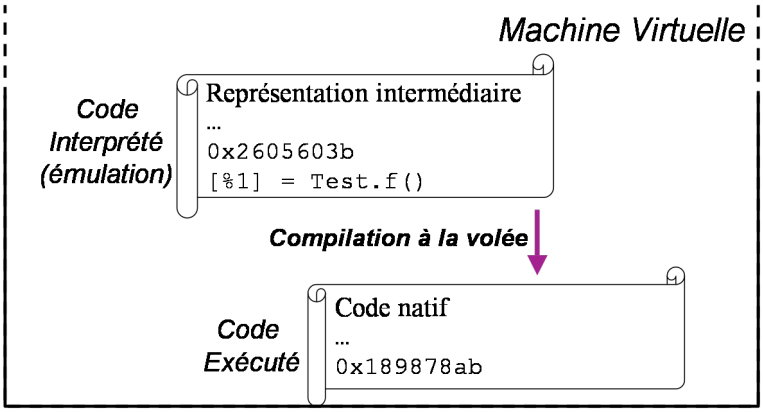

Figure 5 - Compilation à la volée et interprétation

Fondamentalement, la compilation à la volée d'une représentation intermédiaire est de la compilation statique (compilation avant exécution). Tout compilateur statique possède une représentation intermédiaire et transforme cette représentation intermédiaire vers du code machine. Cette partie du compilateur s'appelle la partie arrière. Toutefois, contrairement aux compilateurs statiques, les compilateurs à la volée doivent compiler rapidement les fonctions car ce temps est perdu à l'exécution. La section suivante traite de la compilation à la volée et décrit les différentes représentations intermédiaires possibles ainsi que les techniques les plus fréquemment utilisées pour compiler le code à la volée.

3.3 Processeur Virtuel

Une machine virtuelle est capable d'exécuter la représentation intermédiaire d'une application virtuelle. On identifie donc la représentation intermédiaire avec les instructions machines d'un processeur virtuel. Trois grandes familles de processeurs virtuels existent :

- Les processeurs à pile : le processeur possède une pile et effectue des opérations sur la pile.
- Les processeurs à registres purs : le processeur possède un nombre infini de variables internes appelées registres et effectue les opérations sur les registres.
- Les processeurs mixtes : le processeur possède un nombre fini de registres et peut manipuler une pile, il effectue les opérations sur les registres et sur la pile. Les processeurs réels (IA-32, IA-64, PowerPC…) sont tous des processeurs mixtes.

Les machines virtuelles concrètes possèdent bien sûr des processeurs mixtes, puisqu'il n'existe que des processeurs mixtes dans la nature. En revanche, parmi les machines virtuelles abstraites, on trouve des processeurs à pile comme le processeur virtuel d'une machine virtuelle Java [5], et des processeurs à registres purs, comme le processeur virtuel du canevas de compilation LLVM [6].
Les trois familles de processeurs sont capables d'accéder à la mémoire en lecture ou écriture, d'effectuer des opérations arithmétiques (à valeurs entières ou à valeurs flottantes) et d'effectuer des actions sur le flot de contrôle (sauts, appels de sous-fonctions).
De plus, chacun de ces processeurs gère un espace mémoire spécifique appelé cadre de fonction (frame en anglais). Un cadre de fonction gère le chaînage des appels de fonctions, les arguments d'une fonction et les variables locales d'une fonction. Bien comprendre la notion de cadre de fonction est essentielle pour

comprendre le fonctionnement d'un processeur virtuel.

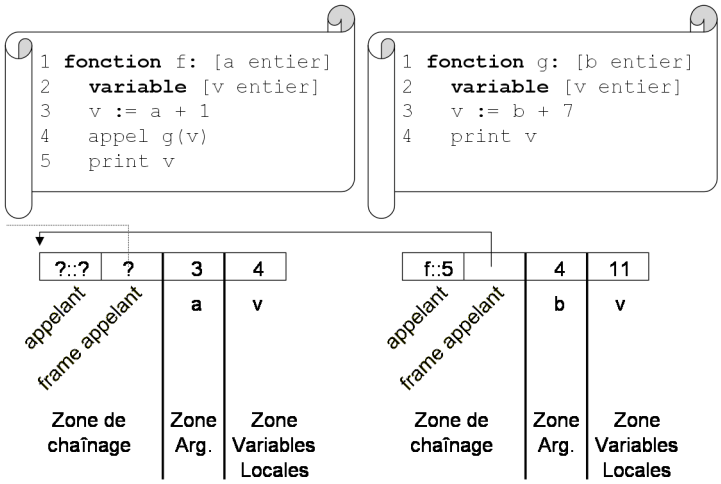

Figure 6 - Cadre de fonctions

La figure Figure 6 - présente un exemple de chaînage entre cadres de fonctions. La fonction f() possède un paramètre a de type entier et une variable locale v de type entier. La fonction g() possède un paramètre b de type entier et un paramètre v de type entier. L'état des cadres est présenté à la fin de l'exécution de la ligne 4 de la fonction g(). Dans la zone de chaînage se trouve le chaînage des fonctions. L'exécution reprendra à la fin de la fonction g() à la ligne 5 de la fonction f() avec le cadre de f(). La zone des arguments contient des arguments de la fonction. Dans cet exemple, les arguments sont copiés du cadre appelant vers le cadre appelé. Les variables locales se trouvent aussi dans le cadre. On remarque que les variables v de f() et g() ne référencent pas le même emplacement. Chaque fonction possède ses propres variables locales.

Compiler le code intermédiaire revient à transformer une séquence d'instruction d'un processeur virtuel en une séquence d'instruction pour le processeur de la machine hôte. Comme les machines hôtes peuvent être réelles ou virtuelles et que les machines réelles possèdent des processeurs mixtes, compiler le code intermédiaire revient à passer d'un processeur virtuel à un autre et donc à transformer une représentation intermédiaire en une autre.
Dans la suite de la section, nous nous focaliserons principalement sur la compilation à la volée d'un des trois types de processeurs vers un processeur mixte. En effet, les autres transformations sont beaucoup plus simples. Nous ne traitons pas ici l'interprétation qui est maintenant un mécanisme minoritairement utilisé dans les machines virtuelles optimisées.

3.3.1 Processeur à pile

Un processeur à pile est défini par une pile, un cadre et de la mémoire. Les instructions de gestion de la pile d'un processeur virtuel permettent de charger (ou décharger) dans la pile des valeurs (constantes, valeurs provenant de la mémoire ou provenant du cadre), de déplacer des valeurs dans la pile et d'effectuer des opérations sur le sommet de la pile.
La figure Figure 7 - présente le code exécuté à la ligne 3 de la fonction f() de la figure Figure 6 - avec du bytecode Java. L'instruction 3.a charge la valeur du premier emplacement du cadre de f au sommet de la pile. Il s'agit du paramètre a. Le

indique qu'il s'agit de la valeur du symbole a et non du symbole a lui-même. L'instruction 3.b. charge la constante 1, l'instruction 3.c additionne les deux éléments au sommet de la pile, l'instruction 3.d écrit le sommet de la pile dans le second emplacement du cadre de f : la variable locale v. Pour finir, l'instruction 3.e supprime l'élément qui se trouve au sommet de la pile pour que les instructions suivantes travaillent sur une pile vierge.

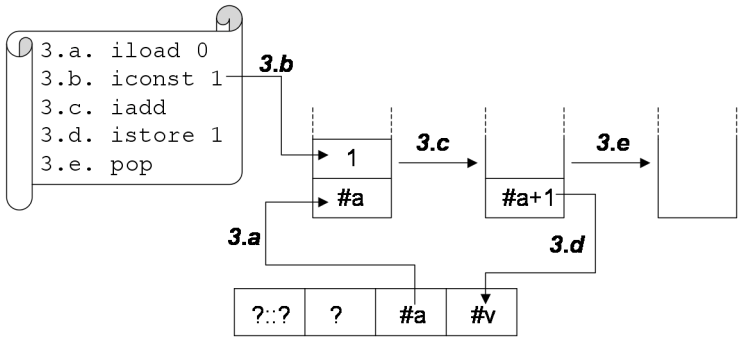

Figure 7 - Exécution dans un processeur à pile

Les processeurs réels sont des processeurs mixtes. Ils possèdent donc un nombre fini de registres sur lesquels le processeur peut effectuer des opérations. Pour compiler une représentation intermédiaire d'un processeur à pile, il faut donc que les éléments qui se trouvent au sommet de la pile soient dans des registres pour effectuer l'opération.

Un algorithme naïf de compilation va placer la pile du processeur virtuel dans le cadre et passer par des registres pour chaque opération. Par exemple, une opération comme 3.c va charger dans deux registres les opérandes qui se trouvent au sommet de la pile, effectuer l'opération sur les registres et écrire le registre résultat au sommet de la pile. Cette solution naïve impose une lecture et une écriture mémoire pour chaque opérande de chaque opération et les performances du code résultant ne sont pas optimales. L'avantage de cet algorithme est qu'il évite tout débordement de registre puisque seul le nombre minimal de registres requis pour l'opération sont utilisés. Cet algorithme nécessite très peu de passes sur le code virtuel et permet donc de compiler rapidement la représentation intermédiaire. Dans le cas d'un compilateur à la volée, minimiser le temps de compilation est essentiel.

Un algorithme plus évolué va essayer de maintenir en permanence dans des registres les éléments qui se trouvent au sommet de la pile. Cet algorithme simple évite le chargement/déchargement systématique des opérandes des opérations et est relativement rapide. Le code produit est de meilleure qualité pour un coût en temps de compilation qui n'est pas excessif. Toutefois, cet algorithme ne permet pas de maintenir des variables locales dans des registres et nécessite des lectures et écritures pour chaque accès à ces variables. Il est bien sûr plus intéressant de maintenir dans des registres les variables locales les plus souvent utilisées.

Il s'avère que pour ce problème d'optimisation de l'utilisation des registres, la représentation intermédiaire d'une machine à pile est tout à fait inadéquate. Les machines virtuelles récentes vont bien souvent éviter cette représentation intermédiaire. Dans le cas de Java, le bytecode utilise une représentation à pile, mais ce bytecode est transformé vers une nouvelle représentation intermédiaire pour un processeur à registres pur.

Passer d'une machine à pile à une machine à registre pur permet d'unifier les variables locales et les cases de la pile. A chaque variable locale et à chaque case

de la pile, un registre abstrait est assigné. Ensuite le processeur à registre pur peut transformer le code virtuel.

3.3.2 Processeur à registres pur

Un processeur à registres pur est défini par un ensemble infini de registres et de la mémoire. Il peut effectuer des opérations sur les registres et lire ou écrire dans la mémoire avec un registre. On définit trois types de registres : les registres des variables paramètres qui sont les paramètres d'une fonction (par exemple, a), les registres des variables locales qui sont les variables locales d'une fonction (par exemple, v) et les registres de travail qui servent à stocker les résultats intermédiaires d'un calcul. Les registres de travail correspondent en fait à des cases dans un processeur à pile.
La figure Figure 8 - présente l'exécution de la ligne 3 de la fonction f() de la figure Figure 6 - pour un processeur virtuel à registres pur (le code donné n'a pas d'équivalent parmi les machines virtuelles existantes). Le paramètre a est dans le registre p0 et la variable locale v dans le registre l0. L'instruction 3.a copie le contenu de p0 dans r0, l'instruction 3.b charge la constante 1 dans le registre r1, l'instruction 3.c additionne r0 et r1 puis met le résultat dans le registre r2 et enfin, l'instruction 3.d copie le contenu de r2 dans l0. Les registres de travails sont r0, r1 et r2 dans cette séquence. Cette séquence d'instruction pourrait bien sûr être optimisée en utilisant p0 à la place de r0 et l0 à la place de r2, ce qui éviterait deux copies de registre inutiles. Bien souvent, la représentation intermédiaire va contenir de nombreuses copies inutiles car la phase de génération de la représentation intermédiaire ne va que peu optimiser le code. Le code produit ici correspond exactement à la transformation de la séquence de la figure Figure 7 - pour un processeur à registres pur.

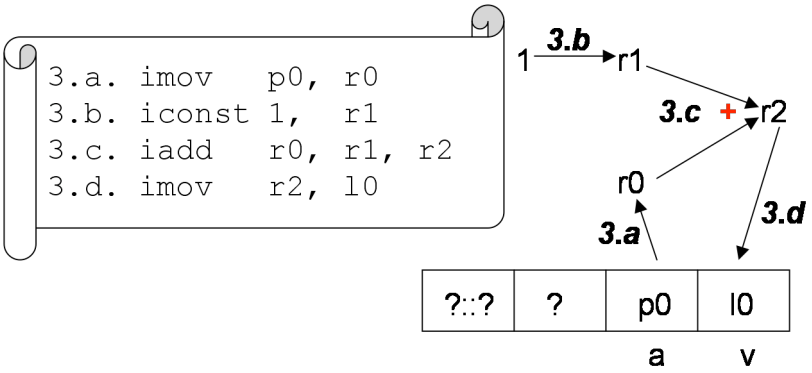

Figure 8 - Exécution dans un processeur à registres pur

Une compilation naïve d'une séquence d'instruction pour un processeur à registres pur est simplement l'utilisation d'une pile et l'application de l'algorithme de la sous-section précédente. Chaque registre est placé dans une pile (en premier les paramètres, en second, les variables locales et en dernier les registres de travail). Une telle génération est loin d'être optimale, mais elle permet d'obtenir une séquence d'instructions concrètes assez rapidement.
Comme les variables de travail, les variables locales et les paramètres sont tous dans des registres virtuels, il est beaucoup plus facile de trouver des algorithmes qui optimisent l'utilisation des registres réels. Des algorithmes de compilation évolués vont chercher à calculer la durée de vie des registres, à éviter les copies inutiles et à mettre en pile les registres les moins souvent utilisés. Les techniques de

compilations à la volée proposées sont en général les mêmes que les techniques de compilation statique, mais l'accent est aussi mis sur le temps de compilation pour éviter d'avoir un temps de réponse trop long lors de la première invocation. Pour approfondir le sujet, le lecteur pourra se référer à de nombreux ouvrages et articles spécialisés traitant de la compilation et de la compilation à la volée [7][8].

3.3.3 Processeur mixte : l'émulation

Un processeur mixte est un processeur réel avec un nombre fini de registres. Les variables d'un programme se trouvent donc soit dans des registres, soit dans le cadre de la fonction. Analyser le code intermédiaire de ce type de processeur est beaucoup plus complexe pour trois raisons :

- Les processeurs mixtes sont en général des processeurs réels et possèdent beaucoup plus d'instructions que les processeurs à pile ou les processeurs à registres pur.
- Il est beaucoup plus difficile de différencier les accès au cadre des accès à la mémoire globale. En effet, le cadre se trouve en mémoire et les mêmes instructions accèdent au cadre et à la mémoire globale.
- Les instructions de branchement peuvent être indirectes et ne peuvent donc pas être connue pendant la phase de transformation.
- Il n'y a pas une gestion unique du cadre, c'est-à-dire que le processeur laisse une certaine liberté quant à la construction et au référencement du cadre.

Pour ces raisons, les machines virtuelles abstraites sont en général basées sur des processeurs à pile ou des processeurs à registres pur qui offrent plus de possibilités d'optimisation. En revanche, les machines virtuelles concrètes sont forcément basées sur des processeurs mixtes puisque seuls des processeurs mixtes existent physiquement. Les machines virtuelles concrètes vont souvent interpréter le code car la compilation est très difficile. Dans ce cas, on parle d'émulateur. Toutefois, des compilateurs de processeurs mixtes (aussi appelé convertisseurs binaires) ont été développés. Citons le célèbre cas de l'émulateur de M68K d'Apple lorsque Apple est passé sur PowerPC ou Rosetta, le traducteur de PowerPC vers IA-32 d'Apple lorsque Apple est passé sur une architecture IA-32.

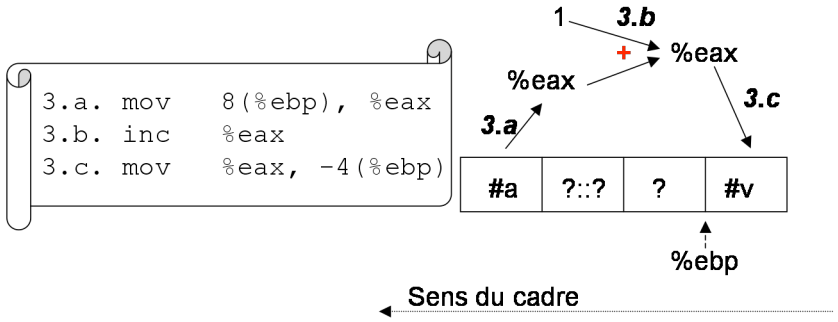

Figure 9 - Exécution dans un processeur mixte

La figure Figure 9 - présente le code correspondant à l'addition de la fonction f() de la figure Figure 6 - sur le processeur mixte réel IA-32. Le paramètre se trouve à l'index 8 du cadre et la variable locale à l'index -4. Le pointeur de cadre est donné par le registre %ebp et se trouve au milieu du cadre réel.
Une technique pour compiler une telle séquence est d'essayer de reconstruire une représentation intermédiaire permettant d'appliquer des algorithmes connus : soit pour un processeur à pile, soit pour un processeur à registres pur. De plus, certaines

formes du code doivent être reconnues pour détecter les actions sur le cadre (par exemple, la séquence push %ebp - mov %esp, %ebp - sub n, %esp). L'une des grandes difficultés de cette technique qui reconstruit les cadres et qu'elle suppose que les cadres de fonctions sont toujours construits de la même façon. Or pour un même processeur, il est possible de construire les cadres de fonctions de différentes façons. Par exemple, rien n'empêche de choisir que le pointeur de cadre se trouve à un autre endroit dans le cadre dans notre exemple. Pour cette raison, cette technique ne peut pas être utilisée en général.
Une autre technique de compilation pour des processeurs mixtes repose sur l'observation d'un émulateur (ou interpréteur). Un émulateur possède une boucle qui récupère la prochaine instruction, exécute le code associé à l'instruction et passe à l'instruction suivante. En l'absence de saut, il est possible de juxtaposer les code associés à des instructions virtuelles qui se suivent. Le code obtenu correspond exactement à ce que l'interpréteur aurait exécuté mais évite de charger l'instruction courante. De plus, le compilateur peut supprimer des instructions inutiles, comme le positionnement du bit de dépassement dans l'instruction 3.b de la figure Figure 9 - .
Le compilateur peut aussi essayer d'utiliser des registres réels pour stocker des registres virtuels.
Le code compilé obtenu avec cette technique est un mélange de code compilé et d'interprétation. En effet, le problème des sauts doit être pris en compte. Concrètement, la machine virtuelle qui doit exécuter l'instruction se trouvant à l'index CO va d'abord vérifier si cette séquence de code a déjà été compilée. Si ce n'est pas le cas, elle va générer un code natif jusqu'au prochain saut en utilisant la technique précédente. Finalement, la machine virtuelle va invoquer le code généré et positionner CO sur le point de saut de la dernière instruction du bloc de base compilé et recommencer le processus.
Cette technique de compilation s'appelle la traduction binaire dynamique. Les performances sont nettement meilleures qu'avec du code interprété car le chargement de l'instruction suivante est évité, le code inutile est éliminé et des registres de la machine hôte sont utilisés. Toutefois, le code obtenu est de moins bonne qualité que du code exécuté nativement. De manière générale, le code généré à partir d'un processeur à pile ou à registres pur sera donc de meilleure qualité que du code généré à partir d'un processeur mixte.
Dans le cas particulier où le processeur abstrait et le processeur de la machine hôte sont identiques, d'autres techniques sont proposées pour augmenter les performances. Dans ce cas, on parlera de machine para-virtuelle. Ce cas particulier est traité en détail dans la sous-section suivante.

3.3.4 Machine para-virtuelle

Dans le cas particulier des processeurs virtuels exécutés sur des processeurs hôtes identiques, il devient intéressant d'essayer d'exécuter nativement les instructions. Toutefois, il est impossible de laisser une application virtuelle, qui peut être un système d'exploitation complet, interférer avec le système d'exploitation hôte ou avec les systèmes d'exploitation des autres machines virtuelles. On parle de para-virtualisation lorsqu'une machine virtuelle exécute nativement ses applications mais protège les systèmes d'exploitation. Le terme para-virtuel vient du fait que l'exécution est native, mais que les instructions privilégiées sont exécutées virtuellement par la machine virtuelle. Une partie du processeur seulement est virtualisé. Une machine para-virtuelle est souvent appelée un moniteur de machine virtuel ou hyperviseur. Fondamentalement, un moniteur de machine virtuelle est une machine virtuelle, mais

le moniteur ne va pas émuler les instructions : il laisse le processeur exécuter les instructions non privilégiées. En revanche, il traque (to monitor) les instructions privilégiées et simule leur comportement.
Deux types de moniteurs de machines virtuelles existent. Les moniteurs de premier type (appelé type I) s'exécutent sur machine nue, c'est-à-dire sur une machine sans autre système d'exploitation. Xen [9] est l'exemple le plus connu de moniteur de type I. Le second type (appelé type II) s'exécute en mode applicatif au-dessus d'un autre système d'exploitation. VMWare est l'exemple le plus connu de moniteur de type II.

Un moniteur de machines virtuelles exécute le code des systèmes d'exploitation en mode non privilégié. Ainsi, lorsque le système d'exploitation va tenter de faire un accès privilégié, une faute matérielle est déclenchée. Cette faute est récupérée par le moniteur qui émule le comportement normal. Dans le cas des moniteurs de type I, cette faute est récupérée directement via la table des interruptions. Dans le cas des moniteurs de type II, cette faute est récupérée via le signal qui correspond à la faute matérielle.
Par exemple, lorsqu'un système modifie la table des interruptions (voir section 2.2), une faute de privilège est générée par le processeur. Le moniteur gère sa propre table des interruptions pour les multiplexer pour les différents systèmes. Il va enregistrer la table d'interruption et acheminer ainsi les interruptions au système actif.

Toutefois, les processeurs réels ne génèrent pas de fautes de privilège pour toutes les instructions. Par exemple, sur un processeur IA-32, l'instruction POPF dépile dans le registre EFLAGS le haut de la pile. Ce registre contient des bits généraux, comme le bit de dépassement de capacité d'un entier, mais aussi des bits qui indiquent le privilège courant pour les entrées/sorties. Si cette instruction est exécutée avec un privilège suffisant, ces bits sont modifiés, sinon, ces bits ne sont pas modifiés et ce, de façon silencieuse. Il faut donc que le moniteur de machine virtuelle soit capable de détecter ce type d'instruction. Deux grandes réponses sont apportées par les moniteurs de machines virtuelles :

- La réécriture du code source du système. C'est le cas de Xen. Certaines instructions privilégiées et les instructions qui posent problème comme POPF sont remplacées par des appels au moniteur.
- La modification à la volée du code du système. C'est le cas de VMWare. Les instructions privilégiées sont détectées et remplacées par des appels au moniteur.

Quelle que soit la technique utilisée, seul le code du système doit être réécrit. En effet, les processus s'exécutent de toute façon en mode non privilégié.

3.4 Système d'exploitation d'une machine virtuelle abstraite

Les machines virtuelles concrètes sont principalement développées pour pouvoir exécuter des systèmes d'exploitation réels développés pour d'autres processeurs que le processeur hôte. Il n'y a donc aucun intérêt à fournir un système d'exploitation avec la machine virtuelle. En revanche, le but d'une machine virtuelle abstraite et d'offrir une machine virtuelle sur laquelle les applications peuvent directement s'exécuter, et ce, indépendamment du système d'exploitation hôte. Elles sont donc, en général, fournies avec un système d'exploitation intégré. Ce système d'exploitation va se reposer sur le système d'exploitation hôte pour fournir des

services et va fournir de nouveaux services systèmes indépendants.
La plupart des abstractions classiques des systèmes d'exploitation se retrouvent dans les machines virtuelles : fichiers, sockets, processus, signaux… Toutefois, chaque machine virtuelle ne fournit pas forcément ces services. Par exemple, la machine virtuelle Java n'offre pas d'équivalent aux signaux et avant la version 7 et l'api d'isolation (JSR-121), elle fournissait un équivalent aux processus légers (thread), mais pas aux processus lourds. Depuis la version 7, une machine virtuelle Java fournit bien l'équivalent des processus lourd (à mémoire séparée), mais pas d'équivalent à la mémoire partagée (voir section 2.3).
La plupart des machines virtuelles abstraites fournissent deux mécanismes systèmes particulièrement novateurs par rapport aux systèmes d'exploitation classiques :

- Un vérifieur de code utilisant les informations de typage qui assure qu'une application ne corrompt pas la mémoire. Ce nouveau mécanisme système ne peut exister que dans les machines virtuelles abstraites car il demande d'annoter le code avec des informations de typage.
- Un récupérateur mémoire ou ramasse-miettes : il permet d'automatiser la libération de la mémoire et évite les fuites mémoire. Ce nouveau mécanisme peut exister sur des machines réelles mais à un coût prohibitif. Dans les machines virtuelles abstraites, avec les informations de typage, le coût est très largement diminué et rend son utilisation intéressante.

3.4.1 Isolation logicielle et vérification du code

Les processeurs assurent l'isolation entre le noyau et les applications en utilisant des possibilités offertes par le matériel : un niveau d'exécution du processeur et la mémoire virtuelle (voir sections 2.1 et 2.3). Il existe toutefois une autre façon d'assurer l'isolation : en utilisant des langages de haut niveau et en limitant l'accès aux données.

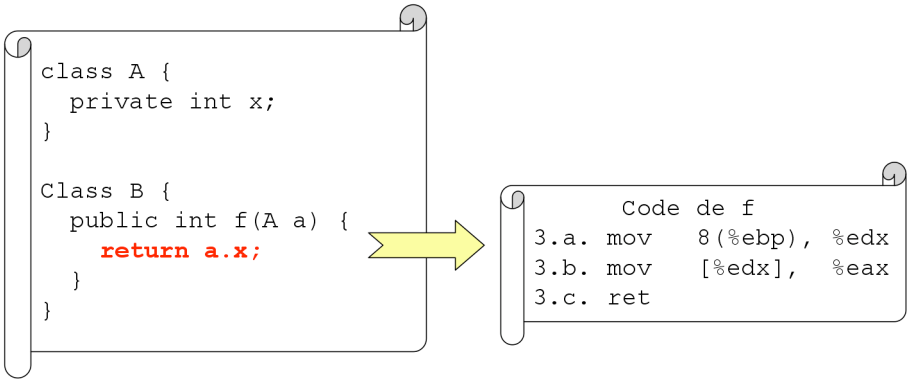

Figure 10 - Isolation logicielle

La figure Figure 10 - montre un exemple en Java d'isolation logicielle. Le champ x de la classe A est privé, c'est-à-dire qu'il n'est pas accessible de l'extérieur de la classe A. Dans la fonction f(), l'accès au champ x sera donc interdit. Vérifier de telles propriétés sur le source du programme est très facile. En revanche, à moins d'annoter le code et de passer par une phase de vérification, il est impossible de détecter que l'instruction 3.b de la séquence de droite est interdite. En effet, dans cette séquence, le paramètre est une référence vers un objet de type A. Le registre %edx pointe donc vers A et permet l'accès au champ qui se trouve à l'index 0 de cette structure : le champ a. Avec ce code assembleur, il n'y a aucun moyen de savoir que %edx est une référence vers une instance de A et il n'y a donc aucun

moyen de savoir que l'instruction 3.b accède à un champ privé d'une classe.
Les machines virtuelles abstraites vont pouvoir utiliser des instructions de plus haut niveau que du bytecode machine pour vérifier qu'un programme ne fait pas d'accès illicite dans la mémoire : en annotant le bytecode avec des informations de typage, en donnant des règles d'accès et en évitant des instructions non typées, comme le mov en 3.b.
On définit l'isolation matérielle comme les mécanismes qui permettent d'isoler les processus et le noyau en utilisant le mode d'exécution du processeur et la mémoire virtuelle. On définit l'isolation logicielle comme les mécanismes qui permettent d'isoler des morceaux de code en utilisant le typage et les règles d'accès (privé, public, protégé…).
Utiliser le typage pour isoler les applications a deux grands avantages :

- Une fois vérifié, un accès à une donnée est direct. Avec l'isolation matérielle, le système passe son temps à copier les données dans l'espace accessible en lecture du processus : toute donnée qui se trouve dans le noyau et qui doit être accédée par un processus doit préalablement copiée. Avec l'isolation logicielle, ces copies sont évitées.
- Pour les machines virtuelles abstraites qui s'exécutent en mode non privilégié, il n'y a aucun moyen d'utiliser l'isolation matérielle pour assurer l'isolation entre la machine virtuelle et les applications. L'isolation logicielle est donc nécessaire.

Un processeur abstrait va donc posséder des instructions spécifiques pour invoquer des méthodes ou consulter des champs d'une structure ou d'un objet. De plus, il a accès aux descriptions des structures manipulées.

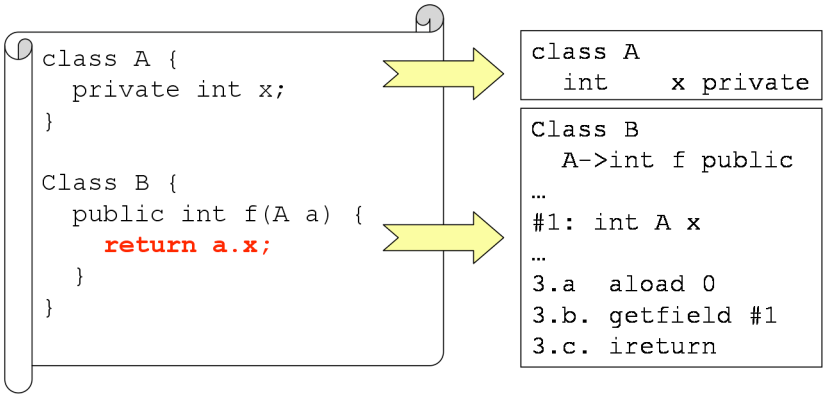

Figure 11 - Vérification de type en Java

La figure Figure 11 - présente un exemple avec du bytecode Java. Le bytecode de la classe A décrit la classe avec ses champs. En particulier, le champ x, de type entier, a une portée limitée à la classe elle-même. La classe B utilise le champ x de la classe A. Ce champ est référencé dans le bytecode sous le nom #1. Le code de la fonction f() est aussi donné. L'accès au champ se fait en utilisant un opcode de haut niveau, getfield, et non via un accès direct à la mémoire. La machine virtuelle va passer par une phase de liaison et de vérification. Lors de la liaison, elle met en correspondance le nom #1 avec la description de x dans la classe A. Ensuite, lors de la vérification, elle détecte un accès interdit puisque le champ x est n'est pas accessible à partir de la classe B. De cette façon, la machine virtuelle isole bien la classe A et la classe B de façon logicielle et n'a à aucun moment besoin du matériel. De plus, on remarque que si l'accès au champ x était autorisé, la machine virtuelle pourrait générer le code d'accès sans vérification pour un processeur IA-32 donné à

la figure Figure 10 - .
Assurer l'isolation logicielle nécessite trois propriétés fortes :

- Les champs doivent être décrit au minimum par leur type et leur portée et tout accès à un champ doit utiliser cette description.
- Les méthodes doivent être décrites au minimum par leur signature (le type des paramètres et du retour) et leur portée, et tout appel de méthode doit utiliser cette description.
- Il est interdit de transtyper un entier vers une référence de classe ou de structure. Sinon, un utilisateur pourrait utiliser sa propre classe avec une portée publique pour accéder à un champ privé d'une autre classe.

La transtypage d'une référence de classe vers une autre référence de classe est autorisé à la condition qu'il soit vérifié par la machine virtuelle.
Des travaux récents menés par Microsoft avec Singularity [10] montre aussi qu'il est plus performant d'utiliser l'isolation logicielle pour isoler un noyau que d'utiliser l'isolation matérielle.

3.4.2 Récupérateur mémoire

Le second nouveau mécanisme système offert par les machines virtuelles abstraites est le récupérateur mémoire, aussi appelé ramasse-miettes (ou glaneur de cellules – GC). Un ramasse-miettes va s'occuper de libérer automatiquement la mémoire qui n'est plus utilisée. Ainsi, ce n'est plus le développeur qui doit s'en occuper, ce qui offre deux avantages :

- Les fuites mémoire sont évitées. En effet, un développeur peut oublier de libérer des morceaux de mémoire. Cette erreur est très difficile à détecter car elle n'implique pas de fonctionnement anormal du programme.
- Tous les morceaux de mémoire peuvent être libérés. En effet, certains morceaux de la mémoire ne peuvent pas être récupérés manuellement par une application sans aide. Par exemple, il est impossible de savoir quand libérer une méthode car un programme n'a aucun moyen simple de savoir si un de ces threads n'est pas en train d'exécuter cette méthode.

Il existe deux grandes familles de récupérateur mémoire : le compteur de références et le parcours explicite du graphe des objets atteignables.
La première famille va compter le nombre de références vers un objet à chaque affectation. La figure Figure 12 - présente un exemple de fonctionnement. Avant l'affectation de la variable s, l'objet B référence l'objet chaîne de caractère "Coucou". Celui-ci a donc un compteur de référence égal à 1. Pendant l'affectation de la variable s, le compteur de références de l'ancien objet référencé est décrémenté. S'il tombe à zéro, il est libéré. Le compteur de références du nouvel objet est, quant à lui, incrémenté.

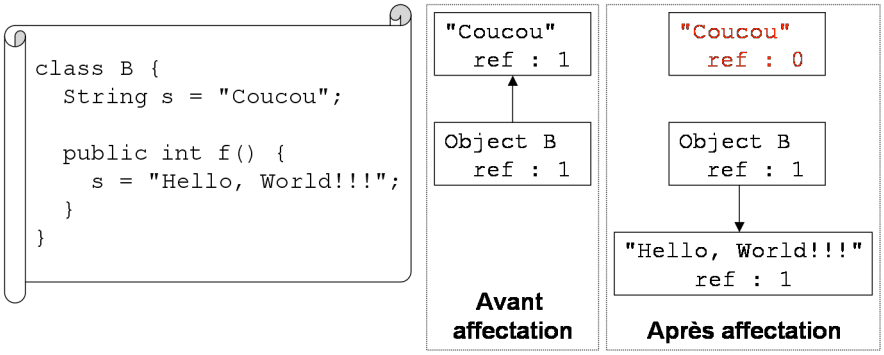

Figure 12 - Ramasse-miettes par compteur de références

Cette première famille de récupérateur mémoire possède un grand avantage : la libération se fait au plus tôt. Dès qu'un objet est récupérable, il est libéré. Si l'objet libéré est associé à une ressource limitée, comme un descripteur de fichier, celui-ci peut être libéré au plus tôt.
En revanche, le coût d'un tel algorithme devient rapidement prohibitif puisque chaque affectation (dans une variable locale, lors d'un retour de fonction ou lors d'une affectation dans une autre structure) entraîne l'exécution d'un petit peu de code. De plus, cet algorithme ne permet pas de détecter les cycles. Si deux objets a et b se référencent l'un l'autre, ils vont avoir un compteur de référence égal à 1. Ces deux objets ne sont donc pas libérés. Mais si ni a, ni b ne sont référencés ailleurs dans le programme, ils pourraient l'être.

Pour pallier les problèmes posés par les ramasse-miettes à compteur de références, on préfère souvent utiliser des algorithmes de parcours de graphe. Un certain nombre d'objets sont notés comme racines. Régulièrement, le graphe des objets atteignables à partir des racines est calculé. Tout objet qui n'est pas atteint à la fin du parcours est considéré comme libre : en effet, une application ne va plus pouvoir non plus atteindre cet objet. On distingue souvent deux sous-familles parmi les algorithme de parcours de graphe : le marque-et-balaye et le marque-et-copie. Dans le premier cas, le parcours de graphe note les objets atteints. Ensuite, un parcours des objets non atteints libère les objets. Dans le second cas, le parcours de graphe copie les objets dans une nouvelle zone de la mémoire. Ensuite, l'ancienne zone est globalement libérée. Cet algorithme évite un parcours des objets à libérer, mais entraîne une copie des objets atteints et impose donc la reconstruction des références si les références et les pointeurs vers les objets sont identifiés.

On classe aussi souvent les algorithmes de ramasse-miettes en deux catégorie : stoppe le monde et incrémentiel. Un algorithme stoppe le monde interrompt l'exécution de la machine virtuelle pendant la phase de collection et un algorithme incrémentiel continue l'exécution des applications. Un compteur de références est toujours incrémentiel. En revanche, un parcours de graphe peut être incrémentiel ou stoppe le monde.
Construire un algorithme de parcours de graphe incrémentiel n'est pas une tâche aisée. En effet, il faut avoir une tâche, appelé collecteur, qui s'occupe de balayer la mémoire pendant que les tâches de l'application, appelés mutateurs, continuent leur exécution. Comme le collecteur et les mutateurs accèdent en parallèle à la mémoire, des problèmes de synchronisation se posent. Une solution élégante à ce problème repose sur la notion de barrière en écriture. La barrière peut interrompre le mutateur pendant qu'il écrit dans la mémoire, mais dans ce cas, le ramasse-miettes est un stoppe le monde. La barrière peut aussi noter quel est l'emplacement mémoire modifié pendant la collection. À la fin de la collection, tous les emplacements mémoire qui ont été modifiés sont parcourus de nouveau.
Un algorithme efficace pour noter les objets mutés pendant une collection repose sur la notion de zones salies. La mémoire est découpée en zones. Lors d'une écriture dans un objet, un bit correspondant à la zone dans lequel se trouve l'objet est activé. À la fin de la collection, le parcours du graphe recommence sur toutes les zones modifiées. Activer le bit d'une zone ne demande que d'exécuter trois instructions machines (décalage de bit, addition et écriture). Il s'avère qu'une bonne taille de zone est de 256 octets.

On fait aussi souvent la différence entre des ramasse-miettes exacts et conservatifs. Un ramasse-miettes exact connaît la structure de la mémoire et sait exactement où se trouvent les références vers des objets. Un ramasse-miettes conservatif quant à lui ne connaît pas la structure de la mémoire. Il possède une table qui indique si un nombre est effectivement dans l'espace mémoire des objets. Un ramasse-miettes conservatif va parcourir l'ensemble de la mémoire. Quand il parcourt un nombre N, il va regarder dans cette table si ce nombre correspond à un objet. Si c'est le cas, l'objet est noté atteignable et parcouru. Ce type de ramasse-miettes peut faire des erreurs : il est tout à fait possible que le nombre N, bien qu'indiquant un objet, soit utilisé comme un nombre ou soit simplement une chaîne de caractères. Toutefois, cette erreur est conservative puisque dans le pire des cas, des objets qui auraient pu être libérés vont être maintenus en mémoire. Dans le cas d'un code natif, seuls des ramasse-miettes conservatifs sont possibles. C'est le cas du célèbre ramasse-miettes de Boehm. Dans le cas d'une représentation intermédiaire pour une machine virtuelle abstraite, les informations de typage sont présents et la structure des objets est connue : un ramasse-miettes exact peut donc être utilisé. Un ramasse-miettes conservatif est obligé de parcourir l'ensemble de la mémoire alors qu'un ramasse-miettes exact ne parcourt que les emplacements mémoires où se trouvent des références. Dans le premier cas, le coût en termes de performances est prohibitif alors que dans le second, le parcours du graphe des objets atteignables est largement diminué.

Pour finir, on observe qu'en général, les objets qui sont présents depuis longtemps en mémoire sont rarement libérés. En effet, un objet est soit un objet temporaire dans le sens où il va servir temporairement dans quelques méthodes pour effectuer une action, soit un objet du programme qui va avoir une durée de vie très longue. Un exemple typique d'objet temporaire est donné en Java par la classe StringBuffer. Un StringBuffer permet de construire des chaînes de caractère. Il va donc servir pendant cette phase de construction, mais ne sera plus utilisé une fois la chaîne de caractère construite. Un algorithme simple de parcours de graphe va parcourir l'ensemble du graphe des objets atteignables : parmi ces objets se trouvent des objets à durée de vie longue et d'autre à durée de vie courte. Parcourir systématiquement les objets à durée de vie longue n'est pas intéressant puisque ces objets sont probablement toujours atteignables. Un nouveau type d'algorithme de ramasse-miettes a donc été développé pour profiter de cette propriété : les ramasse-miettes générationnels. Un ramasse-miettes générationnel va associer une génération à chaque objet. Plus un objet est présent depuis longtemps en mémoire, plus sa génération augmente. Les objets de petites générations sont alors collectés plus souvent que les plus vieux objets. En général, seules deux générations sont définies (jeunes et vieux). Lorsqu'un objet jeune est atteint à la fin d'une collection des jeunes objets, il est promu dans la vieille génération. L'une des difficultés des ramasse-miettes générationnels est le problème des vieux objets qui référencent de plus jeunes objets. Pour éviter de parcourir la vieille génération pour les trouver, il faut être capable de noter si un objet d'une vieille génération référence un objet d'une jeune génération. De tels objets doivent alors être utilisés comme racines temporaires lors d'une collection d'une jeune génération. L'algorithme des zones salies utilisé pour rendre un ramasse-miettes incrémentiel permet de résoudre ce problème. Si une vieille zone est modifiée en dehors d'une collection, elle peut contenir des racines temporaires.

Pour conclure, un ramasse-miettes exact et générationnel a des performances tout à fait acceptables. Ce type de ramasse-miettes n'est possible que dans les machines virtuelles abstraites car il demande d'avoir une connaissance exacte de la mémoire et de la structure des objets qui s'y trouvent. La plupart des machines virtuelles abstraites implantent ce mécanisme, mais les systèmes d'exploitation n'offrent donc pas ce service.

4 Conclusion et perspectives

La virtualisation est principalement utilisée pour masquer l'hétérogénéité matérielle et système. Un système d'exploitation virtualise les périphériques et une machine virtuelle virtualise de plus le processeur. Les machines virtuelles concrètes virtualisent des processeurs existants alors que les machines virtuelles abstraites définissent de nouveaux processeurs abstraits. La virtualisation est un sujet essentiel et très étudié depuis ces dernières années avec d'une part l'émergence d'Internet et le besoin d'uniformisation des applications, et d'autre part la puissance des processeurs récents qui permettent de loger plusieurs systèmes sur la même machine.

Enfin la prolifération des matériels embarqués (carte à puce, téléphone, PDA) conduit à des spécialisations. En effet, nous avons considéré implicitement des ordinateurs standards, (un processeur assez performant, de la mémoire suffisante pour la machine virtuelle et ses applications et un accès à un réseau efficace), ainsi que des périphériques comme un écran graphique, un clavier et une souris. De plus, la notion de "qualité de service", que ce soit en terme de temps de réponse, de tolérance aux fautes ou autres, n'est aucunement garantie.

Il existe cependant des matériels qui ne correspondent pas à ces critères "standards" et/ou des applications ayant besoin d'une certaine qualité de service. Par exemple pour les cartes à puce, en raison de leur faible capacité (processeur et principalement mémoire), une version spéciale de la machine virtuelle Java a été développée, la JavaCard. Pour les téléphones portables, une autre version de la machine virtuelle Java a été développée d'abord la KVM, puis J2ME. En fonction de l'application et des différents terminaux cibles, le concepteur peut s'appuyer sur deux langages/configurations de l'embarqué (CDC) ou de la mobilité (CLDC). Ce raisonnement s'applique pour les assistants personnels (PDA), bien qu'actuellement leurs performances tendent a se rapprocher des ordinateurs "standards" et donc à supporter les mêmes machines virtuelles que ces derniers. C'est la prolifération de ces machines virtuelles qui a été à la base des travaux sur les machines virtuelles virtuelles [11].

Pour en savoir plus

Bibliographie

[1] Andrew Tanenbaum, Systèmes d'exploitation, Pearson Education France,

2003, 2e éd. (ISBN 2-7440-7002-5)
[2] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, November 2005, Third Ed. (ISBN 10: 0-596-00565-2 | ISBN 13: 9780596005658)
[3] Maurice J. Bach, Design of the UNIX Operating System, Prentice Hall Software Series, 1986, (ISBN-10: 0132017997 | ISBN-13: 978-0132017992)
[4] Marshall Kirk McKusick, George V. Neville-Neil, The Design and Implementation of the FreeBSD Operating System, Addison-Wesley Professional, 2004, First Ed. (ISBN-10: 0201702452 | ISBN-13: 978-0201702453)
[5] Tim Lindholm, Frank Yellin, The Java(TM) Virtual Machine Specification, Prentice Hall, 1999, Second Ed. (ISBN-10: 0201432943 | ISBN-13: 978-0201432947)
[6] Chris Lattner and Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, in Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society, 2004 (ISBN : 0-7695-2102-9)
[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools, Addison Wesley Publishing Company, 2006, Second Ed. (ISBN-10: 0321486811 | ISBN-13: 978-0321486813)
[8] Andrew W. Appel, Jens Palsberg, Modern Compiler Implementation in Java, Cambridge University Press, 2002 (ISBN-10: 052182060X | ISBN-13: 978-0521820608)
[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Xen and the art of virtualization, Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003, ACM (ISBN : 1-58113-757-5)
[10] Galen C. Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Microsoft Research Technical Report MSR-TR-2005-135, Microsoft Corporation, Redmond, WA, October 2005
[11] Gaël. Thomas, Nicolas Geoffray, Charles. Clément, Bertil Folliot. Designing highly flexible virtual machines: the jnjvm experience. In Software: Practice and Experience. John Wiley & Sons, Ltd., 2008

Normes

Réglementation

Organismes

Constructeur – Fournisseurs – Distributeurs

Normes

Thèses

Sites Internet