Virtualisation Logicielle

Gaël Thomas
gael.thomas@lip6.fr

Université Pierre et Marie Curie
Master Informatique
M2 – Spécialité SAR

Virtualisation

Principe d'un système d'exploitation

- ✓ Unifier l'accès au matériel
  - ☞ Diminuer le travail de développement des applications
  - ☞ Couche d'accès au matériel développée une unique fois
- ✓ Exécuter plusieurs applications en parallèle

⇒ **Première branche de la virtualisation : les systèmes d'exploitation**
(Linux, Unix, Windows, MacOS, Chorus, Mach, Solaris, OS/2, Android…)

Système d'exploitation = virtualisation des périphériques + accès au cpu

Virtualisation

Mais… Il faut encore unifier les systèmes d'exploitation

Une application tourne sur un unique OS et un unique matériel

Si en moyenne un processeur possède n os et qu'il existe m processeur, il faut n*m systèmes pour faire tourner toutes les applications (et m machines!)

Exemple : n = 4, m = 5 ⇒ 20 systèmes et 5 machines

⇒ **Seconde branche de la virtualisation : les émulateurs**
(Bochs, Mac 68K emulator, Rosetta…)

Émulateur = virtualisation d'une machine complète

Trop lent!

- ✓ Traduction processeur X vers processeur Y très difficile à optimiser
- ✓ Aucune aide du matériel

Virtualisation

Solution logicielle

- ✓ Construire un jeu d'instruction abstrait facile à optimiser par un compilateur

⇒ **Troisième branche de la virtualisation : les machines virtuelles applicative**
(JVM, CLI, VM Python…)

Machine virtuelle applicative = émulateur optimisé d'un machine abstraite

Solution matérielle

- ✓ Se servir du matériel pour optimiser les émulateurs (pagination à Intel VT/AMD V)

⇒ **Quatrième branche de la virtualisation : les moniteurs de machines virtuelles**
(Xen, VMWare, Parallels, MoL…)

Moniteur de machines virtuelles = émulateur optimisé d'une machine concrète

Ne peut émuler qu'un processeur X sur un processeur X…

Plan du cours

1. **Systèmes d'exploitation et virtualisation**
2. Émulation et performances
3. Moniteurs de machines virtuelles
4. Ramasse-miettes

Systèmes d'exploitation et virtualisation

Revisitons quatre mécanismes système classiques :

- ✓ Virtualisation de l'accès au processeur
- ✓ Virtualisation des fautes matérielles
- ✓ Virtualisation de la mémoire
- ✓ Virtualisation des entrées/sorties

Virtualisation de l'accès au processeur

Émuler l'accès au processeur pour

- ✓ Exécuter d'autres applications lorsqu'une application fait des E/S
- ✓ Permettre l'utilisation d'une machine par plusieurs utilisateurs
- ✓ Attendre plusieurs fin d'E/S en même temps dans la même application

Principes :

- ✓ Chaque processeur émulé est identique à l'original
  - ☞ Revient à multiplexer la machine
- ✓ Assurer une équité d'accès au processeur aux applications
  - ☞ Interdiction des instructions privilégiés
  - ☞ Plus d'accès direct au matériel (sauf mémoire d'entrée/sortie!)

Virtualisation de l'accès au processeur

Tâche (thread) : état + pointeur de code

- ✓ État : registres, signaux, fichiers ouverts, code, pile, donnée, tas
  = une machine complète!
- ✓ Pointeur de code : indique quelle est la prochaine instruction à exécuter

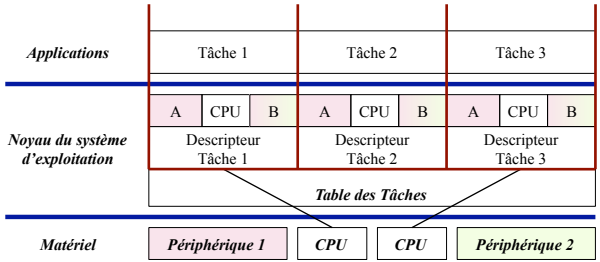
Processus : ensemble de tâches qui partagent une partie de leur état

- ✓ En général le tas, le code et les données

Système multi-tâches : exécute plusieurs tâches en parallèle

- ✓ Chaque tâche s'exécute comme si elle était la seule sur un matériel virtuel
- ✓ Le noyau gère l'équité d'accès et l'isolation entre tâches
- ✓ Change régulièrement la tâche qui s'exécute (préemption, ordonnancement)

Virtualisation de l'accès au processeur



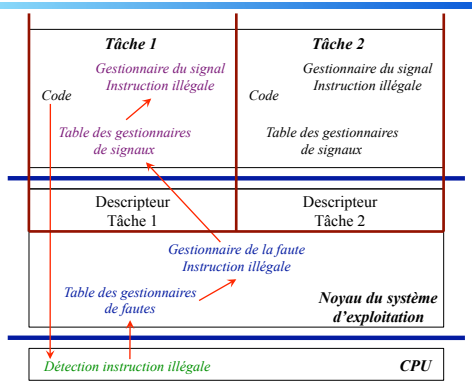
A : virtualise le périphérique 1
exemple : carte réseau virtualisé en socket
B : virtualise le périphérique 1 et périphérique 2
exemple : carte réseau + disque dur virtualisés en système de fichiers (nfs et ext3)

Virtualisation des fautes matérielles

Faute matérielle : faute due au logiciel/matériel détectée par le matériel, gérée par le système d'exploitation
- ✓ Division par zéro, accès mémoire illégal, instruction illégal
- ✓ Identifiée par un numéro
- ✓ Fonction de traitement associé

Virtualisation : le signal
- ✓ SIGFPE, SIGSEGV, SIGILL
- ✓ Identifié par un numéro
- ✓ Fonction de traitement associé

Virtualisation des fautes matérielles



Virtualisation de la mémoire

Rôle du système : assurer l'isolation entre processus
- ✓ Plusieurs processus dans la même mémoire physique
- ✓ Processus et système dans la même mémoire physique
- ⇒ Empêcher un processus d'accéder à la mémoire du système ou d'un autre processus

Trois solutions
- ✓ Solution logicielle : copier l'application sur disque lors d'une commutation
  Pas une solution pour isoler le système, multi-cœur impossible, **lent**
- ✓ Solution matérielle avec privilège : idem mais application n'accède pas à OS
  Multi-cœur impossible, **lent**
- ✓ Solution matérielle avec pagination et privilèges : virtualise la mémoire physique
  Multi-cœur possible, rapide

Virtualisation de la mémoire

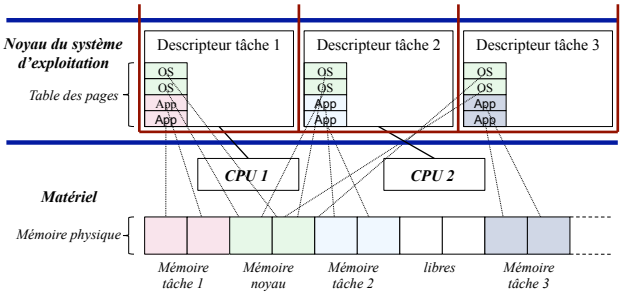
Table des pages : table de correspondance entre adresse physique et virtuelle

- ✓ Une table par processus
- ✓ Chaque processeur possède sa propre table de correspondance
- ✓ Privilège requis associé aux adresses

Page : grain de la table de correspondance

Taille définie par le processeur (et/ou son mode) : 4ko, 1Mo, 4Mo…

Virtualisation de la mémoire



- ✓ Possibilité de partager des pages entre tâches (shared memory, thread posix)
- ✓ Pages physique pas forcément contiguës

Virtualisation des entrées/sorties

Principe commun à (presque!) tous les périphériques :

Ensemble de données ordonnées consultables en lecture et/ou écriture

- ☞ Clavier : octets en lecture
- ☞ Fichier, carte réseau : octets en lecture/écriture
- ☞ Imprimante, terminal (écran) : octets en écriture

Idée directrice : tout périphérique peut être virtualisé en tant que fichier

Virtualisation des entrées/sorties

Vnode = ensemble ordonnée d'octets

- ✓ **Lecture** d'octets avec déplacement de la tête de lecture
- ✓ **Écriture** d'octets avec déplacement de la tête de lecture

Trois autres fonctions

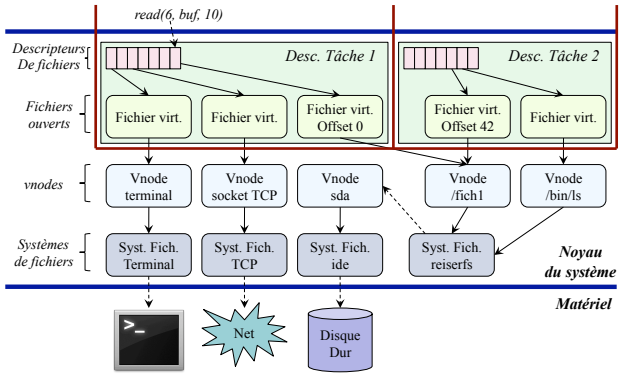
- ✓ **Ouverture** et **fermeture**
  - ☞ Certains périphériques nécessitent une connexion (socket de mode connecté)
  - ☞ Facilité du système pour mettre en cache des représentants des périphériques
- ✓ **Configuration** (ioctl)
  - ☞ Configuration spécifique du vnode

Mise en œuvre

- ✓ Structure de système de fichier : table de fonctions + données du périphérique
- ✓ Toute vnode possède un pointeur vers cette structure
- ✓ Le périphérique peut être lui-même un autre vnode

Reiserfs au dessus du vnode de sda1

Virtualisation des entrées/sorties



Virtualisation des entrées/sorties

Interface de VNode pas toujours suffisant

- ✓ Mémoire mappée
  - ☞ Exemple : mémoire vidéo
- ✓ Paramètre de fonction ne peuvent pas être toujours génériques
  - ☞ Exemple : ouverture de socket en mode flux, lecture d'une socket UDP

Systèmes d'exploitation et virtualisation

Système d'exploitation virtualise une machine

- ✓ Virtualise le matériel sous la forme de périphériques virtuels
- ✓ Multiplexe l'accès au processeur
- ⇒ Émule plusieurs machines virtuelles avec un processeur réel

Problème restant à résoudre : une application est écrite pour un OS et un CPU

- ✓ Applications non portables
- ✓ Plusieurs machines avec plusieurs OS pour pouvoir lancer en // des applications différentes

Plan du cours

1. Systèmes d'exploitation et virtualisation
2. **Émulation et performances**
3. Moniteurs de machines virtuelles
4. Ramasse-miettes

Émulation et performances

Émulation d'une machine X sur une machine Y pour rétro-compatibilité
(MacOS 9 -> MacOS 10, ppc -> ia32 -> ia64)

Trois techniques pour émuler une machine réelle X sur une machine réelle Y

- ✓ Interpréter le code binaire de X sur Y
  - ☞ Les registres de X sont des emplacement mémoire de Y
  - ☞ Chaque instruction de X est décodée et interprétée par Y
  - ☞ Beaucoup trop lent
- ✓ Convertir le code binaire de X vers Y (compilation)
  - ☞ Les registres de X sont des variables locales
  - ☞ Le code de X est dé-compilé vers une représentation de haut niveau, optimisé puis recompilé
  - ☞ Peut être assez rapide
- ✓ Exécuter en natif Y sur X si X est ou émule déjà un Y
  - ☞ Moniteur de machine virtuelle (voir suite du cours)

Émulation et performances

Conversion de code binaire difficile :

Un compilateur peut optimiser du code parce qu'il « comprend » le code :

- ✓ Flot de contrôle clair
  - ☞ Branchement et invocation de fonction explicite
- ✓ Flot de donnée clair
  - ☞ Variables typées, pointeur explicite
- ✓ Pas de notion de pile
  - ☞ Le code manipule des variables locales et des arguments uniquement

Le langage machine n'est pas adéquat pour de l'optimisation

```
                                this.f(this.y)
mov (%eax)+4, %ebx              %ebx <- this.y
mov %ebx, (%esp)-8              arg[1] <- %ebx
mov %eax, (%esp)-4              arg[0] <- %eax
mov (%eax), %eax                %eax <- this.vt
mov (%eax)+8, %eax              %eax <- this.vt[f] = addr de f
call %eax                       Appel f
```


Émulation et performances

Conséquence : peu ou pas d'optimisations possibles en traduisant du langage machine

- ✓ Impossibilité de comprendre où commencent et finissent les fonctions
- ✓ Saut indirect incompréhensibles

Solution actuelle : traduction binaire dynamique

- ✓ Génère le code pour les blocs de base (séquences d'instructions sans saut)
  Un bloc de base = une fonction
- ✓ A chaque saut, appel du traducteur qui génère/réutilise le bloc de base cible

Avantages :

- ✓ Un bloc de base natif peut être (un peu) optimisé
- ✓ Exemple : Rosetta 40% à 80% plus lent que natif (http://www.geekpatrol.ca/2006/02/rosetta-performance/)

Plan du cours

1. Systèmes d'exploitation et virtualisation
2. Émulation et performances
3. **Moniteurs de machines virtuelles**
4. Ramasse-miettes

Moniteurs de machines virtuelles

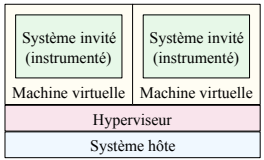
Principe : permettre à plusieurs OS de s'exécuter en même temps sur la même machine

- ✓ Émulation d'une machine physique A en machine physique A
- ✓ Éxécution native quand c'est possible (performances accrues)

Pourquoi faire :

- ✓ Isolation : OS A plante ⇒ ne compromet pas les autres OS
- ✓ Rétro-compatibilité : exécution d'applications pour d'anciens OS (msdos dans windows 95)
- ✓ Compatibilité : exécution simultanée d'applications pour différents OS
- ✓ Continuité de service : migration de l'OS pour maintenance (cloud computing)
- ✓ Tolérance aux pannes : points de reprise de l'OS (cloud computing)

Moniteurs de machines virtuelles



Moniteur de type I :
système hôte = machine nue
(Xen, ESX Server de VMware)

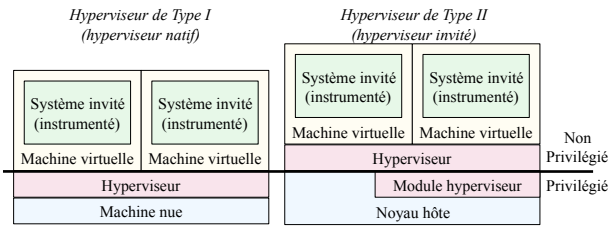
Moniteur de type II :
système hôte = système d'exploitation
(VMware workstation, Parallels)

Définitions :

- ✓ **Moniteur de machines virtuelles** (ou hyperviseur) : environnement d'exécution de machines virtuelles
- ✓ **Machine virtuelle** : machine émulée par un moniteur de machines virtuelles
- ✓ **Système invité** : système d'exploitation exécuté dans une machine virtuelle
- ✓ **Système hôte** : environnement dans lequel s'exécute l'hyperviseur

Moniteurs de machines virtuelles

Architecture logicielle des moniteurs de machines virtuelles



Moniteurs de machines virtuelles

Principe des moniteurs de machines virtuelles

- ✓ Éxécuter les processus invité nativement
  - ☞ Performances natives, sauf quand appel au système
- ✓ Instrumenter le noyau du système pour le faire collaborer avec l'hyperviseur
  - ☞ Performances vont de native à émulée

Performance générale des moniteurs de machines virtuelles :
de 5 à 10% de perte de performances par rapport au natif

Moniteurs de machines virtuelles

Rôles d'un moniteur de machine virtuelle

- ✓ **Gestion des processeurs** : chaque machine virtuelle doit « voir » des processeurs et « croire » qu'elle s'exécute en mode privilégié
- ✓ **Gestion de la mémoire** : chaque machine virtuelle doit avoir un espace d'adressage continu et « croire » que c'est la mémoire physique
- ✓ **Gestion des périphériques** : chaque machine virtuelle doit « voir » des périphériques qui ne coïncident par forcément avec les périphériques du système hôte

Moniteurs de machines virtuelles

Un système invité doit s'exécuter dans un mode moins privilégié que l'hyperviseur

⇒ Le système invité ne peut et ne doit pas s'exécuter en mode privilégié!

Principe de **l'hyper-appel** : attraper toutes les instructions privilégiées du système invité et les renvoyer vers l'hyperviseur

Trois techniques :

- ✓ Virtualisation complète **: émulation** complète du noyau
  (VMware fusion, WMware workstation, Parallels)
  - N'empêche pas les applications invitées de s'exécuter nativement
- ✓ Para-virtualisation **: réécriture** de certaines parties des sources du noyau
  (Xen, VMWare ESX, User Mode Linux, Co-Linux)
- ✓ Utilisation du **matériel**
  (Pratiquement tous les hyperviseurs aujourd'hui)
  - ☞ Certaines instructions n'ont pas la même sémantique en mode U et S (popf par exp)
  - ☞ Nécessite Intel-VT ou AMD-V

Moniteurs de machines virtuelles

Exécution du noyau

Émulation complète du noyau

- ✓ Rapide
- ✓ Pas d'accès au source du noyau ⇒ maintenable

Réécriture des sources

- ✓ Performances proche du natif (hyper-appel plus lent que instruction machine)
- ✓ Nécessite de modifier les sources du noyaux ⇒ non maintenable

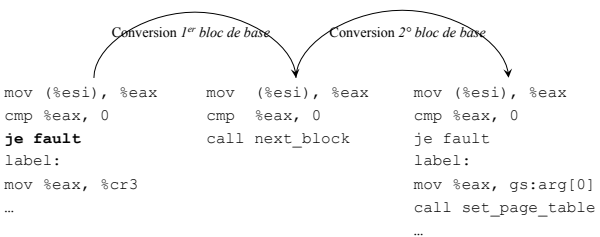
Utilisation du matériel

- ✓ Rapide
- ✓ Pas d'accès au source du noyau ⇒ maintenable

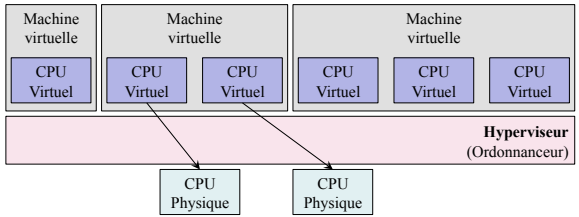
Moniteurs de machines virtuelles

Émulation du noyau : réécriture binaire du noyau

- ✓ Même principe que conversion binaire : réécriture de **blocs de base**
- ✓ Copie des instructions non privilégiées
- ✓ Insertion d'hyper-appels pour instructions privilégiées



Moniteurs de machines virtuelles



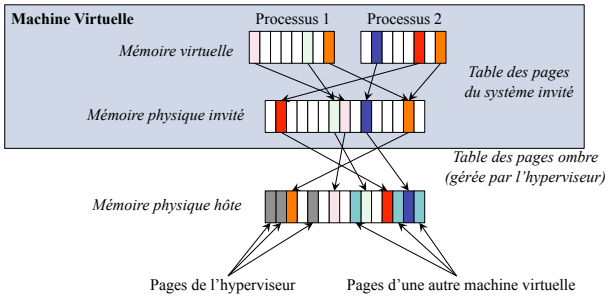
Gestion des processeurs

- ✓ N processeurs hôtes : processeurs physiques
- ✓ M processeurs invités : processeurs virtuels

État d'un processeur virtuel : `idle` ou `run`

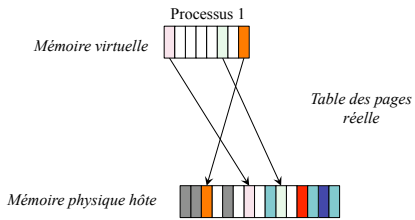
Moniteurs de machines virtuelles

Gestion de la mémoire : vu par le logicielle



Moniteurs de machines virtuelles

Gestion de la mémoire : vu par le processeur hôte



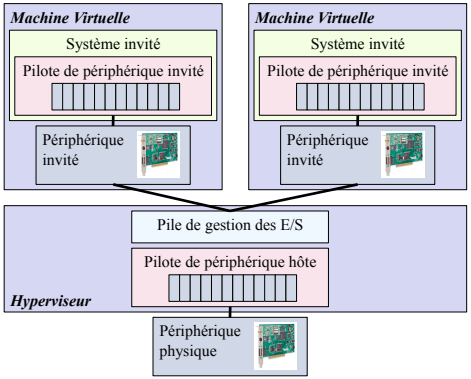
Moniteurs de machines virtuelles

Mise en œuvre :

- ✓ Par le logiciel : solution active
  - ☞ Attrape toutes les modifications de la table invité
    En enlevant les droits lecture/écriture aux pages de la table invité
  - ☞ Répercute les changements dans la table réelle via la table ombre
  - ☞ Gestion des bits accessed et dirty lors d'une faute en lecture dans la table hôte
- ✓ Par le logiciel : solution paresseuse
  - ☞ Construit une table des pages réelle vide
  - ☞ A chaque faute mémoire, remplit la table des pages réelle
  - ☞ Gestion des bits accessed/dirty beaucoup plus couteuse
- ✓ Par le matériel depuis 2006
  - ☞ Extended page table pour Intel VT et Nested page table pour AMD-V
  - ☞ Gère le double chaînage de façon matériel (correspondance + bits accessed/dirty)
  - ☞ Le moniteur n'est plus impliqué lorsque la table hôte change

Moniteurs de machines virtuelles

Gestion des périphériques



Moniteurs de machines virtuelles

Problème : obligation de copier du buffer du périphérique hôte à celui du périphérique invité si DMA accède à la mémoire physique

- ✓ Mémoire physique invité non continue
- ✓ Ouvre la porte à la mémoire des autres machines virtuelles!

Aide matériel avec Intel VT-d ou AMD I/O MMU
DMA en passant par adresses virtuelles et non physiques

- ✓ Assure qu'une VM ne peut pas attaquer l'hyperviseur
- ✓ Permet à l'OS hôte de gérer lui même ses entrées/sorties sur le matériel virtuel

⇒ DMA entre périphérique et mémoire pilote invité (si périphérique associé à une unique machine virtuelle)
(hyperviseur n'intervient plus, processeur non sollicité)

⇒ DMA entre mémoire pilote hôte et mémoire pilote invité sinon
(libère le processeur pendant ce temps)

Moniteurs de machines virtuelles

Pour conclure

- ✓ Éxécution de plusieurs OS en // avec un moniteur de machines virtuelles
- ✓ Technologie en pleine extension
  - ☞ Intel-VT et AMD-V sont en productions depuis 2006
  - ☞ IOMMU aussi, exploités depuis 2007
- ✓ Performances proches du natif : 5 à 10% de dégradation
- ✓ Ouvre la voie au cloud computing
  - ☞ Ferme de serveur, chaque serveur gère des VMs
  - ☞ Continuité de service, informatique omniprésente

☞ Un des buzzword de 2011 ☺

Plan du cours

1. Systèmes d'exploitation et virtualisation
2. Émulation et performances
3. Moniteurs de machines virtuelles
4. **Ramasse-miettes**

Ramasse-miettes

Ramasse-miettes : récupérateur automatique de la mémoire

But premier : éviter les bugs les plus difficiles à trouver!

- ✓ Double libération ⇒ bug arrive plus tard
- ✓ Llibération avant utilisation d'une référence ⇒ bug arrive plus tard
- ✓ Fuite mémoire ⇒ pas vraiment un bug mais quasi-impossible de trouver la cause

But second : décharger le développeur de cette gestion

- ✓ Libération d'objets utilisés en multi-threads très difficile
- ✓ Le développeur ne se préoccupe plus de la libération

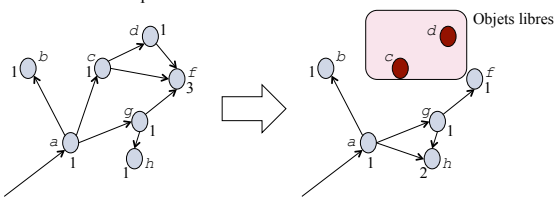
Ramasse-miettes

Deux techniques de base

- ✓ Le compteur de références
  - ☞ Compte le nombre de références vers un objet
  - ☞ Si ce nombre tombe à zéro, libère l'objet
- ✓ Le parcours de graphe
  - ☞ Parcours le graphe des objets atteignables
  - ☞ Tout objet non atteint via ce parcours peut être libéré

Ramasse-miettes

Ramasse-miettes à compteur de références



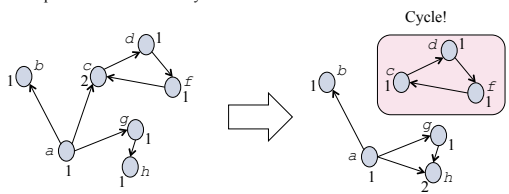
Remplacement de c par h dans a (i.e. exécution de a.field = h)

- ✓ Incrémente le compteur de h ⇒ 2
- ✓ Décrémente le compteur de c ⇒ 0 ⇒ libère c
  ⇒ décrémente les compteurs de d et f ⇒ libère d ⇒ décrémente le compteur de f

Ramasse-miettes

Ramasse-miettes à compteur de références peu utilisés :

- ✓ Exécution d'un peu de code à chaque affectation de référence ⇒ très couteux
  - ☞ Ecriture référence dans objet
  - ☞ Ecriture d'une référence dans la pile, dans un registres
- ✓ Impossible de libérer les cycles

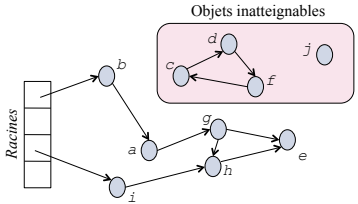


Efficace si pas de cycle et libération contrôlée : exemple : compteur d'ouverture d'inodes dans un noyau (libération dans close)

Ramasse-miettes

Les ramasse-miettes à parcours de graphe : exécuté pendant des périodes de collection

- ✓ Ensemble de références racines
  Variable locales, arguments de fonction, variables globales (static en Java)
- ✓ Parcourt du graphe des objet atteignables à partir des racines
- ✓ Tout objet non atteint par le graphe ne peut pas non plus l'être par l'application!



Ramasse-miettes

L'algorithme marque et balaye (mark and sweep)

- ✓ Phase de collection
  - ☞ Identifie les racines
  - ☞ Parcours le graphe
- ✓ Phase de récupération
  - ☞ Parcours l'ensemble des objets qui n'ont pas été atteint pendant le parcours et les libère

Complexité :

- ✓ N est le nombre d'objet = A + L (Atteignable + Libre)
- ✓ Phase de collection : complexité A
- ✓ Phase de récupération : complexité L
- ⇒ Complexité N

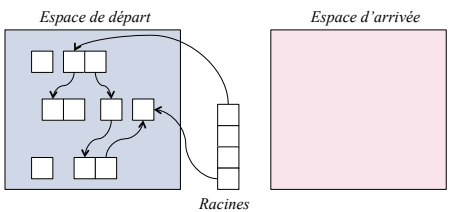
Autres problèmes :

- ✓ Fragmentation de la mémoire ⇒ localité ↘ et temps d'allocation ↗
- ✓ Bloque l'exécution de l'application (voir stop-le-monde versus incrémental)

Ramasse-miettes

L'algorithme copiant

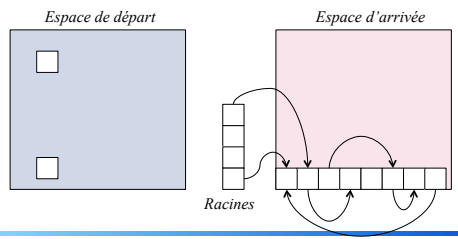
- ✓ Mémoire séparée en deux : espace de départ et espace d'arrivée
- ✓ Pendant la phase de parcours les objets sont copiés dans l'espace d'arrivée



Ramasse-miettes

L'algorithme copiant

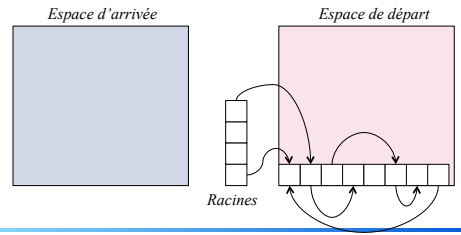
- ✓ Mémoire séparée en deux
- ✓ Pendant la phase de parcours les objets sont copiés dans l'espace d'arrivée
- ✓ A la fin, les deux espaces sont permutés et l'ancien espace de départ est considéré vide



Ramasse-miettes

L'algorithme copiant

- ✓ Mémoire séparée en deux
- ✓ Pendant la phase de parcours les objets sont copiés dans l'espace d'arrivée
- ✓ A la fin, les deux espaces sont permutés et l'ancien espace de départ est considéré vide



Ramasse-miettes

Algorithme copiant

- ✓ Complexité = A où A est le nombre d'objets atteignable
- ✓ Très bonne localité après copie
- ✓ Allocation très rapide : pointeur sur le dernier objet alloué dans l'espace courant, incrémenté à chaque allocation
- ✓ Mais copie couteuse, surtout si A est proche de N où N est le nombre total d'objets
- ✓ Moitié de la mémoire perdue

Variante : l'algorithme compactant

- ✓ Pendant la phase de collection, les objets qui peuvent être déplacé dans des trous sont notés
- ✓ Les objets inatteignables sont libérés et le maximum de trou est bouché
- ✓ Complexité = N (nombre total d'objet)
- ✓ Mauvaise localité
- ✓ Allocation très facile
- ✓ Copie beaucoup moins couteuse

Ramasse-miettes

Deux catégories de ramasse-miettes

Ramasse-miettes exacts

- ✓ Les racines sont connues de façon précise
- ✓ La structure des objets est connue

Ramasse-miettes conservatifs

- ✓ Structure de la pile d'exécution inconnue
- ✓ Structure des objets inconnues (seule la taille est connue)

- ✓ Allocateur spécial permettant de savoir si un nombre peut correspondre à une référence
- ✓ Possibilité de collision entre un entier et une référence
  - ☞ Décision conservative : l'objet référencé est considéré atteint
  - ☞ Impossible de faire un copiant ou un compactant

Ramasse-miettes

Algorithmes exacts

Nécessite aide du compilateur

- ✓ Carte de la pile (StackMap) : indique pour chaque fonction et à chaque PC où se trouvent les références
  - ☞ En pile et où, en registre et où
  - ☞ Hypothèse simplificatrice : on ne lance une collection qu'en des points connus des fonctions (voir algorithmes incrémentaux)
    (les branchements arrières des boucles non bornées, les retour de fonction)
- ✓ Structures exactes des objets connues à l'exécution

- ✓ Facile à faire dans une machine virtuelle car typage et compilateur à la volée
- ✓ Impossible à faire en C, C++, assembleur

Ramasse-miettes

Algorithmes conservatifs

Nécessite une technique pour savoir si un nombre correspond à une référence ou un pointeur à l'intérieur d'un objet

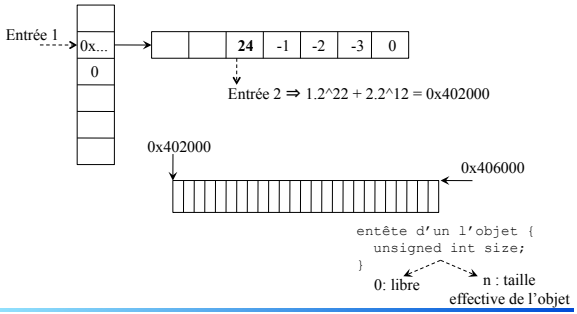
Table de hash de Boehm

- ✓ Tous les objets sont mis dans une table de hash
- ✓ Pour accélérer la recherche, une page est constituée d'objets de même taille
- ✓ Les bits de poids forts sont utilisés comme clé de hash

Ramasse-miettes

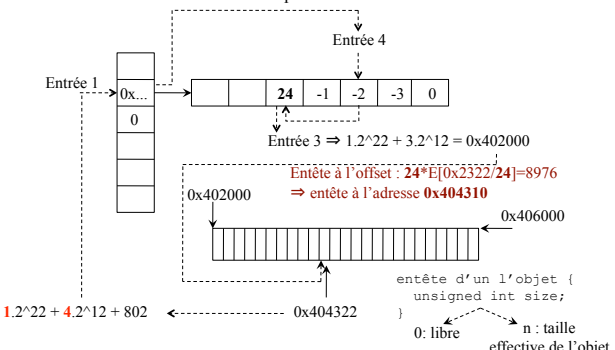
Table de hash de Boehm :

Exemple : la zone de 0x402000 à 0x406000 contient des objets de taille 24



Ramasse-miettes

Table de hash de Boehm : recherche rapide



Ramasse-miettes

Avantages et inconvénients de l'algorithme de Boehm

- \+ Permet de construire des ramasse-miettes conservatifs (pour le C et le C++)
- \+ Allocation relativement rapide (moins qu'avec algorithmes copiant)
- – Très mauvaise localité
- – Beaucoup de gaspillage si peu d'objets de certaines tailles

Ramasse-miettes

Problème : entre deux parcours du graphe, seul un petit sous-ensemble du graphe a évolué

Idée : ne parcourir que ce sous-ensemble

Mais : mise en œuvre difficile

Toutefois : plus un objet est vieux, moins il a de chance d'être inatteignable

- ✓ Variable globale restent pendant toute la durée du programme
- ✓ Objets threads + graphe atteignable vit très longtemps
- ✓ La plupart des objets sont alloués pendant un court lapse de temps (connexions réseaux rapides, tampons temporaires…)

Seconde idée :

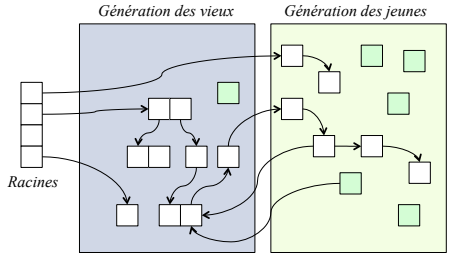
Parcourir souvent le sous-graphe des objets les plus jeunes

Parcourir moins souvent l'ensemble du graphe

Ramasse-miettes

Algorithme générationnel :

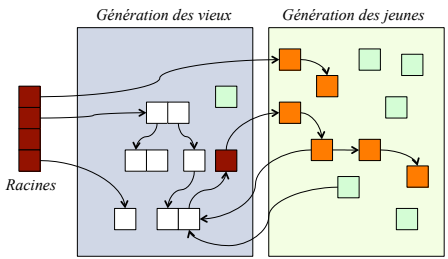
- ✓ Un objet commence sa vie jeune
- ✓ Si il résiste à quelques collections, il vieillit



Ramasse-miettes

Problème : trouver les racines du graphe des objets jeunes

⇒ Exécution d'une barrière en écriture à chaque affectation de référence vers référence et mémorisation systématique de tout objet vieux qui référence un jeune



Ramasse-miettes

Amélioration de l'algorithme générationnel : le générationnel copiant

Un paradis + un purgatoire + un enfer

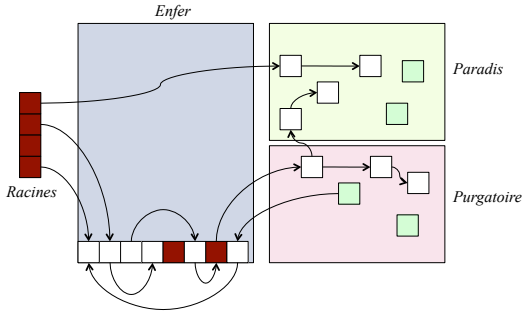
- ✓ La génération jeune est constituée de paradis + purgatoire
- ✓ Allocation dans le paradis
- ✓ Lors d'une collection de la jeune génération
  - ☞ Si objet au purgatoire, alors copié dans l'enfer et marqué racine vieille
  - ☞ Si objet dans paradis, alors ne fait rien
  - ☞ Vide purgatoire + inverse purgatoire et paradis
- ✓ Oublie des racines vieilles au bout de deux collections

Pourquoi obliger les jeunes objets à résister à deux collections

- ✓ Pour empêcher les objets alloués juste avant la collection de vieillir prématurément

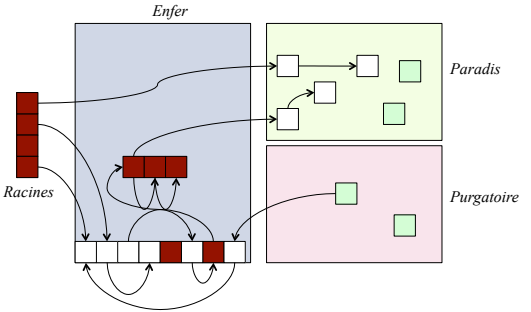
Ramasse-miettes

Illustration de l'algorithme générationnel copiant : avant une collection



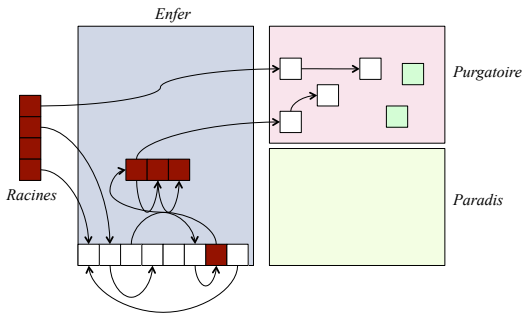
Ramasse-miettes

Phase de parcours et de copie



Ramasse-miettes

Inversion du paradis et du purgatoire + oublie des vieilles racines



Ramasse-miettes

Allocation et concurrence

Problème : éviter la prise de verrou lors d'une allocation

Principe : autant d'allocateurs que de threads

Application :

- ✓ Ramasse-miettes conservatif : chaque threads a ses propres zones
- ✓ Ramasse-miettes générationnels avancés : un couple paradis/purgatoire par thread

Actuellement, les ramasse-miettes performants suivent ce schéma

Ramasse-miettes

Ramasse-miettes et concurrence : deux types de ramasse-miettes

- ✓ Ramasse-miettes stoppe-le-mondes :
  Les threads applicatifs sont interrompus le temps d'une collection
  Si temps collection long, temps de réponse de l'application fortement dégradé
- ✓ Ramasse-miettes incrémentaux :
  Les threads applicatifs continuent à s'exécuter pendant une collection
  Temps de réponse de l'application non dégradé
  Beaucoup plus difficile à mettre en œuvre!

Aujourd'hui : machines multi-cœurs ⇒ incrémental s'exécute vraiment en //

Ramasse-miettes

Ramasse-miettes incrémentaux : le problème

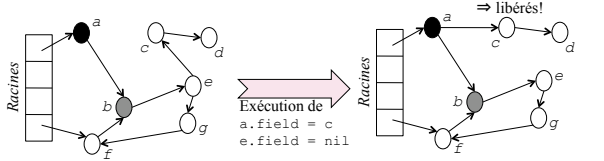
- ✓ L'application est constituée de threads appelés threads mutateurs
- ✓ Le thread qui parcours le graphe des objets atteignables est appelé thread collecteur

Problème classique d'accès concurrent lecteur écrivain

- ✓ Les mutateurs écrivent dans le graphe
- ✓ Le collecteur lit le graphe

Noir = visité – gris = en cours – blanc = non visité

Ratés par collecteur
⇒ libérés!



Ramasse-miettes

L'invariant tri-coleurs de Dijkstra

Un objet noir ne doit jamais référencer un objet blanc

Mise en œuvre :

- ✓ Tout objet noir lu est grisé
- ✓ Ou tout objet blanc lu est grisé

} Nécessite une barrière en lecture

- ✓ Ou lors d'une affectation d'un blanc dans un noir, le noir est grisé (Dijkstra)
- ✓ Ou lors d'une affectation d'un blanc dans un noir, le blanc est grisé (Steel)

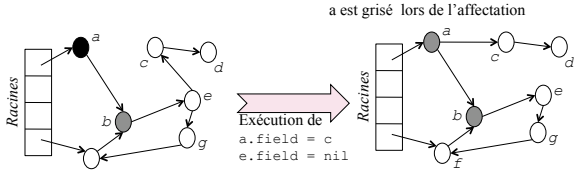
} Nécessite une barrière en écriture

Utilisation exclusivement de barrières en écriture pour question de performances

Ramasse-miettes

Mise en œuvre de l'algorithme de Dijkstra

a est grisé lors de l'affectation



f

Ramasse-miettes

Ramasse-miettes et concurrence : synchronisation des threads

- ✓ Dans stoppe-le-monde : pour bloquer les threads
- ✓ Dans incrémental : pour trouver les racines dans la pile

Mise en œuvre :

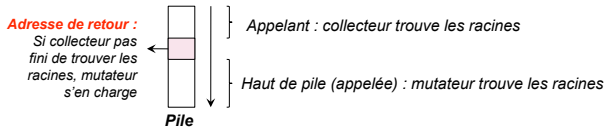
- ✓ Signal posix :
  - ☞ Interruption n'importe quand
  - ☞ Réveille les appels systèmes (read sur socket par exp)
  - ☞ État de la pile difficile à connaître
- ✓ Point de contrôle (handshake) : technique utilisée actuellement
  - ☞ Consultation d'une variable « demande de collection » à des points de contrôles connus (retour arrière de boucle non bornée, retour de fonction)
  - ☞ État de la pile connu à ces endroits avec l'aide du compilateur à la volée (StackMap)
  - ☞ Léger surcoût à l'exécution

Ramasse-miettes

Ramasse-miettes et concurrence

Incrémental pour trouver les racines dans la pile :
ce n'est pas le mutateur qui trouve les racines mais le collecteur

- ✓ Utilisation de point de contrôle
- ✓ Le mutateur trouve les racines dans son cadre de fonction courant
- ✓ Le collecteur trouve les racines dans les appelants
- ✓ Remplacement de l'adresse de retour de fonction dans la pile pour synchronisation
- ✓ Marche bien si le collecteur est légèrement en avance sur le mutateur



Ramasse-miettes

Pour conclure : ramasse-miettes reste un sujet chaud de recherche
(best paper VEE 2009, la conférence de référence sur les machines virtuelles, conférence ISMM chaque année uniquement sur le sujet)

Mécanisme de moins en moins couteux : 40 ans de recherche derrière!

Multi-cœurs ⇒ change radicalement la façon d'appréhender la concurrence dans les ramasse-miettes

Conclusion

Virtualisation à tous les niveaux

- ✓ La technique de référence pour l'isolation
- ✓ Le technique de référence pour masquer l'hétérogénéité

Avancées très significatives ces 15 dernières années

- ✓ Xen : 2003
- ✓ JVM : 1996

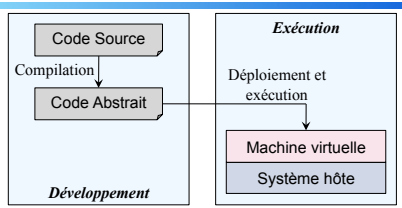
⇒ Performance de la virtualisation proches de performances natives

Ouvre la voix au Cloud computing

Plan du cours

1. Systèmes d'exploitation et virtualisation
2. Émulation et performances
3. Moniteurs de machines virtuelles
4. **Machines virtuelles applicatives**
5. Ramasse-miettes

Machines virtuelles applicatives



Principe : définition d'un **code abstrait** indépendant du matériel et de l'OS

- ✓ Pour masquer l'**hétérogénéité** matériel et système
  - ☞ Le code doit être développé et compilé une unique fois
  - ☞ Le code abstrait s'exécute sur n'importe quelle paire matériel/système d'exploitation
- ✓ Pour assurer l'**intégrité** de l'environnement d'exécution
  - ☞ Le code abstrait est exécuté dans une machine virtuelle qui vérifie que le code ne corrompt pas le reste du système

Machines virtuelles applicatives

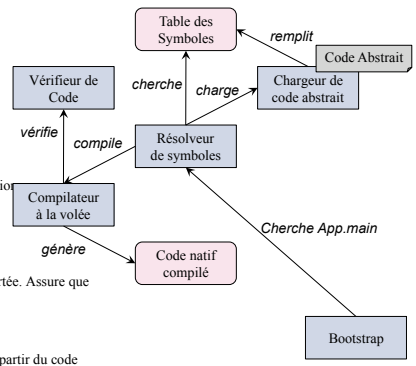
Fonctionnement

Résolveur de symboles :
Renvoie un offset, une méthode compilée ou une classe à partir d'un symbole

Chargeur de code abstrait :
Construit la représentation d'une classe à partir d'un fichier

Vérifieur de code :
Vérifie le typage, la portée. Assure que le code n'est pas malicieux

Compilateur à la volée
Génère du code natif à partir du code abstrait



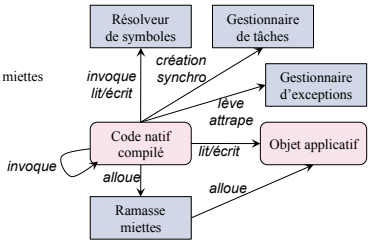
Machines virtuelles applicatives

Fonctionnement

Gestionnaire de tâches
Création de tâche, synchronisation

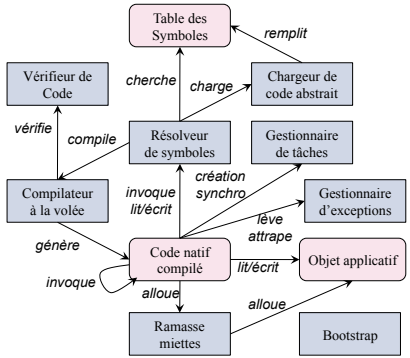
Gestionnaire d'exceptions
Lève ou attrape des exceptions

Ramasse-miettes
Alloue la mémoire, collecte les miettes



Machines virtuelles applicatives

Vue d'ensemble
d'une machine virtuelle



Machines virtuelles applicatives

Représentation du code abstrait

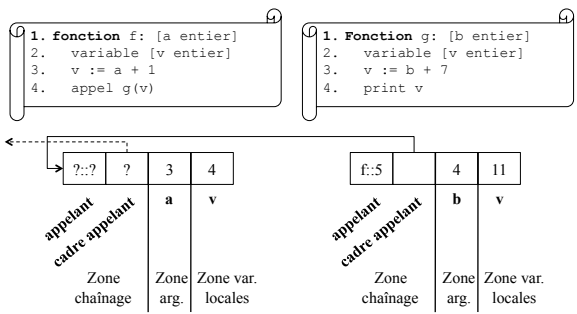
Code source
- ✓ Le source est lexé/parsé puis linéaralisé vers une machine à pile ou à registre pure

Machine à pile pure
- ✓ Les instructions manipulent une pile infinie

Machine à registres pures
- ✓ Les instructions manipulent un ensemble infini de registres

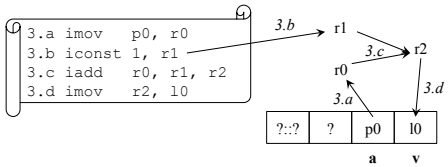
Machines virtuelles applicatives

Rappel : le cadre de fonction



Machines virtuelles applicatives

Machine à registres pure (LLVM) : représentation du code v := a + 1

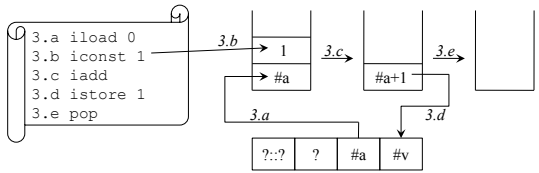


Techniques de compilation :

- ✓ Tous les registres vers pile native + choisit les registres les + utilisés en natif
  ⇒ Performances moyenne : pas d'optimisation particulière
- ✓ Techniques classiques de compilation (SSA, SCCP, GVN, GCM, ABCE, linear scan register allocation…)
  ⇒ Même performances que compilateur statique mais lent à la compilation

Machines virtuelles applicatives

Machine à pile pure (JVM, .NET) : représentation du code v := a + 1



Techniques de compilation :

- ✓ Pile, variables locales et paramètres projetés vers pile native + haut de pile en registres natifs
  ⇒ Performances moyenne : ne profite pas bien des registres natifs
- ✓ Pile, variables locales et paramètres projetés vers registres pour machine à registres pure (ce qui est fait dans toutes les machines virtuelles modernes)
  ⇒ Très bonne performances car représentation facile à optimiser

Machines virtuelles applicatives

Sûreté de fonctionnement : quatre techniques

- ✓ Typage : impossible d'échapper aux règles de typage
  - ☞ Pas de manipulation directe de pointeur
  - ☞ Tout emplacement mémoire possède un type et seul une référence du type valide permet d'y accéder
- ✓ Portée des classes, champs et méthodes
  - ☞ Vérification à la compilation/exécution des droits d'accès aux champs, classes et méthodes
- ✓ Gestion de la mémoire : interdiction de libérer directement la mémoire
  - ☞ Pas de double libération
- ✓ Vérification de débordement
  - ☞ Tout accès à un tableau vérifié (dépassement de bornes)
  - ☞ Tout accès à un objet est vérifié (référence nulle)

Machines virtuelles applicatives

Vérification de typage : exemple avec Java

- ✓ Toutes les méthodes sont signées (types d'entrée et de sortie)
- ✓ Les champs de classes possèdent des types
- ✓ Les champs, les méthodes et les classes possèdent un opérateur de portée
- ✓ Tous les bytecodes sont signés (addition entière, flottante etc…)
- ✓ Transtypages entier vers référence impossible
- ✓ Transtypages référence A vers référence B vérifié
  - ☞ À la compilation si B est un parent de A
  - ☞ A l'exécution si impossible de vérifier que le transtypage est valide à la compilation

⇒ Impossible d'échapper aux règles de typage et de forger une référence