KoinKoin

Système d'exploitation à architecture basée micro-noyau

Antoine Castaing
Nicolas Clermont
Damien Laniel

30 juillet 2006

Free Documentation License

Table des matières

Première partie

Rapport

Chapitre 1

Introduction

1.1 Définition d'un système d'exploitation

Un système d'exploitation fournit une interface entre le matériel et les logiciels utilisateurs.

Il remplit deux fonctions distinctes :

1. la gestion du matériel
2. la gestion des ressources

1.2 Le noyau

1.2.1 Définition

Il s'agit du coeur du système d'exploitation, comparable au cerveau chez l'être humain. C'est le premier programme qui est chargé au démarrage, hors le chargeur de démarrage.

Le noyau est le seul composant du système à être en contact direct avec la couche matérielle. Il dispose donc des privilèges les plus élevés vis-à-vis de l'architecture, afin de pouvoir effectuer les actions demandées par les processus utilisateurs. Ainsi les processus utilisateurs n'ont-ils besoin d'aucun droit particulier vis-à-vis du matériel, permettant entre autres d'isoler ces processus du reste du système. Ainsi le noyau joue-t-il un rôle crucial dans la sécurité du système.

Il va de soit que plus le noyau rend de services, plus le danger est grand qu'une faille existe, permettant à un code malicieux d'utiliser les privilèges du noyau pour anéantir la sécurité du système. Ce qui nous amène à la conception des noyaux et des "modèles" existants.

1.2.2 Différents types de noyaux

Le noyau monolithique

Un noyau monolithique fournit un grand nombre de fonctionnalités qui sont incluses directement dans le code du noyau. Ces fonctionnalités comprennent par exemple la gestion de la mémoire, les pilotes de périphériques, la gestion de processus et divers algorithme d'ordonnancement de ces processus, des systèmes de fichiers, et bien d'autres choses encore. C'est le cas de la plupart des noyaux les plus utilisés aujourd'hui, notamment des noyaux Linux et xBSD.

La quasi-totalité du système est donc constitué par le noyau, tous les différents composants systèmes se partageant les mêmes ressources, rendant difficile le contrôle de flux et la vérification d'un modèle de sécurité. Néanmoins cette approche a pour avantage de très bonnes performances, minimisant le nombre de changements de contexte, et leur coût.

Le micro-noyau

La philosophie des micro-noyaux est de faire monter le code ne nécessitant pas de privilèges matériels particulier dans les couches plus élevées du système. Ainsi les micro-noyaux ont pour avantage de réduire grandement la taille du noyau, en externalisant la plupart des services traditionnellement fournis par les noyaux dans de petits serveurs indépendants, cloisonnés, appelés services, qui tournent en espace utilisateur. Cela nécessite alors un système de communication sûr (IPC basé message) pour que les services puissent communiquer entre eux et avec le noyau.

Ils offrent une plus grande modularité, et donc une meilleure évolutivité. La séparation du code en petits modules permet aussi une plus grande robustesse et une meilleure sécurité. En effet un bug ou une faille dans l'un des services systèmes restera cantonné à ce service, ne propageant pas l'erreur aux autres éléments du système.

De plus les architectures à base de micro-noyau facilite grandement le travail de certification de par leur faible taille, avec peu de lignes de code (LoC, L4 : entre 7000 et 8000, minix3 moins de 4000, à comparer avec les 3 millions de LoC de Linux 2.6).

Mach, L4 et Chorus (C5) sont des exemples de micro-noyaux.

L'exo-noyau

L'approche exo-noyau est encore plus radicale. La philosophie est si un programme utilisateur peut effectuer une action, pourquoi faire intervenir le

noyau ou même des services systèmes ? Ainsi l'exo-noyau noyau ne fait que mettre à disposition des processus utilisateurs les ressources, par exemple de l'espace disque, que le processus utilisateur gère lui-même.

La notion de système d'exploitation s'en trouve radicalement transformée, les services du système n'existant plus en tant que fonctionnalités noyaux ou serveurs utilisateurs, mais en tant que librairies. Par exemple le système de fichiers est mis à disposition uniquement par des fonctions de librairie pour utiliser de façon pratique l'espace disque alloué par l'exo-noyau.

1.3 Historique des systèmes d'exploitation

- Années 1960 : UNIX, noyau monolithique
- Années 1980 : MS-DOS, noyau monolithique
- 1985 : début du développement de Mach, micro-noyau
- 1990 : début du développement de Hurd, système d'exploitation à base de micro-noyau
- 1991 : début du développement de Linux, noyau monolithique
- 2005 : début du développement de KoinKoin, système d'exploitation à base de micro-noyau !

1.4 Présentation sommaire de KoinKoin

Le but du projet KoinKoin est l'obtention d'un système d'exploitation à base de micro-noyau. Il fournit d'une part le micro-noyau (KoinKoin) qui gère entre autres la mémoire, les interruptions, la création de tâches et de fils d'exécutions et l'ordonnancement de ces fils d'exécution, ainsi qu'un système de messages. KoinKoin met aussi à disposition un système de mémoire partagée et des tâches multi-threadées.

D'autre part, il fournit également un ensemble de services qui tournent en espace utilisateur, comme un gestionnaire de modules ou encore des pilotes de périphériques.

L'ensemble formé par KoinKoin et les services utilisateurs est dénommé KoinKoin OS.

1.4.1 Portabilité

Pour le moment, KoinKoin ne fonctionne que sur les architectures Intel 32 bits (IA32).

Cependant, certaines parties du code ont été séparées en une partie dépendante de l'architecture et une partie indépendante afin de faciliter le portage futur de KoinKoin vers d'autres architectures.

Chapitre 2

Le chargeur de démarrage

2.1 Présentation

KoinKoin possède son chargeur de démarrage ou bootloader spécifique. Son rôle est d'installer un environnement d'exécution propice pour KoinKoin, notre micro-noyau. En particulier il est chargé de :

- Installer le mode d'adressage protégé ("protected mode").
- Installer la pagination.
- Jumper sur KoinKoin pour lui donner la main en lui passant les informations nécessaires.

Pour toute information concernant la création de l'image système et la "procédure d'installation" de KoinKoin OS, référez-vous à la section "Comment utiliser KoinKoin" des annexes.

2.2 Utilisation de GRUB pour l'amorce

Nous utilisons GRUB pour effectuer l'amorce de notre système d'exploitation (bootstrap). Celui-ci est utilisé pour démarrer notre bootloader et charger les binaires de KoinKoin et des services de base.

Voici les points forts qui ont justifié ce choix :

- Grub gère lui-même des systèmes de fichiers, ce qui est très utile pour créer une arborescence sur le périphérique utilisé pour démarrer, dans notre cas une disquette. La plupart des systèmes de fichiers courants peuvent être utilisés.

- Grub gère plusieurs périphériques : disquette, disque dur ; réseau, ... Il est donc très facile de lui spécifier où aller chercher les “modules”.
- Lorsque le premier programme est lancé (nommé kernel dans le fichier de configuration), Grub lui transmet des informations sur le système via une structure prédéfinie.
- Lorsque Grub lance le programme, le mode protégé est déjà activé.

Après avoir chargé les modules, il donne la main à notre chargeur de démarrage.

2.2.1 Le fichier ”multiboot.h”

GRUB nous fournit un certain nombre d'informations, certaines étant propres à lui, et d'autres qu'il a pu détecter lors du démarrage. Le fichier ”multiboot.h” est un fichier en-tête qui contient des définitions de constantes et de structures qui nous permettent d'utiliser les informations qu'il nous transmet.

Nous utilisons ensuite ces constantes et ces structures dans notre chargeur de démarrage puis dans notre noyau. GRUB appelle en fait notre chargeur de démarrage en lui passant deux arguments : le nombre magique (magic number) identifiant propre à l'exécutable de notre chargeur de démarrage, et l'adresse de la structure ”multiboot info” qui permet d'accéder à toutes les structures de GRUB.

Ce nombre magique nous permet de vérifier que notre exécutable est bien conforme à la norme ”multiboot” définie par GRUB.

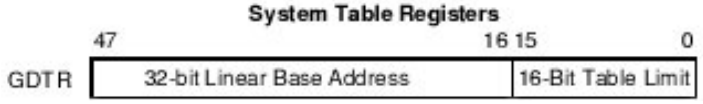
La structure ”multiboot info” nous permet quant à elle d'accéder notamment aux informations suivantes :

- Les adresses haute et basse de la mémoire vive.
- L'adresse du début de l'endroit en mémoire vive où il écrit les structures ”module” donnant les informations sur les modules qu'il a chargé.
- Le nombre de modules chargés.
- Les structures ”module” suscitées contenant les adresses de début et de fin de chaque module, c'est-à-dire où GRUB a chargé les modules en mémoire vive.

2.3 GDT et segmentation

Une table de descripteurs de segment est un "tableau" de descripteur de segments. Cette "structure" a une taille variable et peut contenir jusqu'à 8192 descripteur. La GDT (Global Descriptor Table) est l'un des deux types de table de descripteurs.

Chaque système doit avoir une GDT de définie, qui sera alors utilisée par tous les programmes et tâches du système. Mais la GDT n'est pas elle-même un segment, juste une structure de données. Son adresse de base (adresse de début) et sa limite (sa taille, exprimée en octets) doivent être chargées dans le registre GDTR.

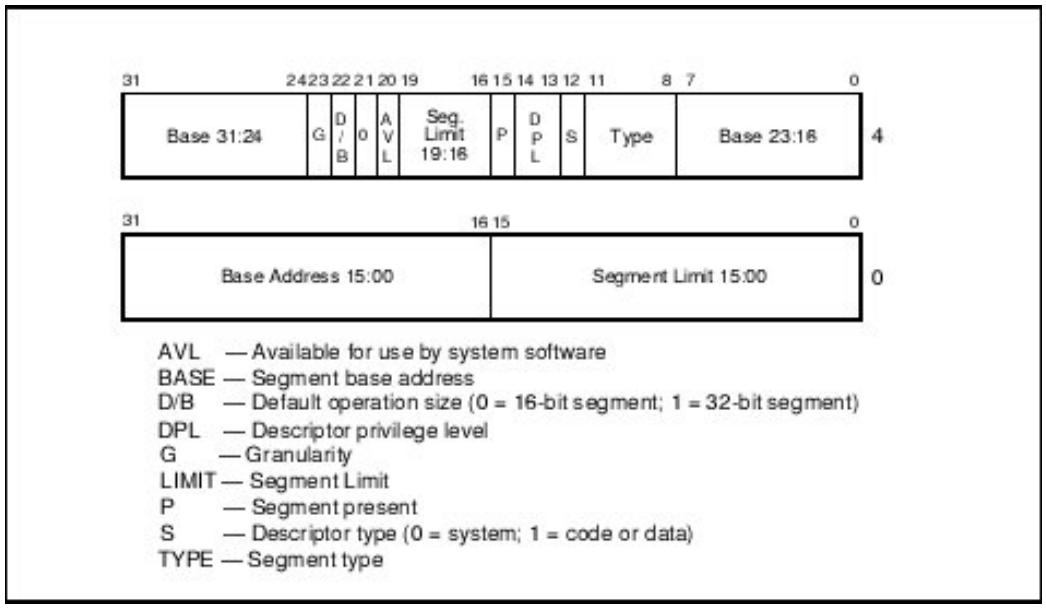


Le registre GDTR[1]

On utilise pour cela l'instruction assembleur **lgdt**, qui prend en paramètre une structure GDTR correctement remplie. Le processeur est alors informé de l'emplacement et de la taille de la GDT, et pourra alors l'utiliser. A noter que le premier descripteur de la GDT n'est pas utilisé par le processeur, et est donc mis à NULL.

Intéressons-nous désormais à la structure d'un descripteur de segment pour comprendre les différences entre segment de code et de données, user et kernel, tss... Voici la structure générale d'un descripteur de segment, et donc d'une entrée de la GDT :

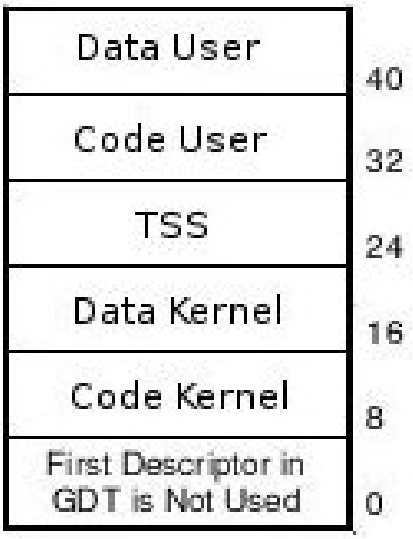
[1]Documentation Intel Volume 3



Un descripteur de segment[2]

Ainsi, pour fixer le type et la nature d'un segment, il suffit de renseigner les différents champs aux valeurs désirées. Par exemple, pour des segments de privilège **Kernel**, il faut fixer le champ **DPL** à la valeur 0x0, tandis que pour des segments de privilège **User**, la valeur est de 0x3.

La GDT mise en place dans Koinkoin est la suivante :



La GDT de Koinkoin

[2]Documentation Intel Volume 3

Chacun des segments recouvre l'ensemble de la mémoire disponible. En effet dans KoinKoin, les propriétés de partitionnement mémoire de la segmentation ne sont pas utilisé, puisque le cloisonnement mémoire est assuré par la pagination. La segmentation de la mémoire s'avérerai donc plus contraignante et limitante qu'utile.

Vous aurez sans doute remarqué que nous utilisons un unique TSS pour le système. En effet compte tenu que nous avons mis en place un changement de contexte manuel, nous n'avons besoin que d'un unique TSS global.

2.4 Passage en mode 32 bits

2.4.1 Mode réel

Lorsqu'un ordinateur démarre, celui se trouve en **mode réel**, c'est-à-dire en mode d'adressage 16 bits. L'adressage s'effectue de la façon suivante :

seg : offset

avec :
- **seg** : Sélecteur de segment sur 16 bits
- **offset** : Offset (décalage) de segment, sur 16 bits

Lorsque le processeur recoit une adresse de cette forme, il multiplie la valeur de **seg** par 16 et y ajoute la valeur de **offset**, obtenant ainsi une adresse sur 20 bits. La limite maximale de mémoire adressable est donc de seulement 1 Mo, ce qui extrêmement limité, réellement insuffisant pour les systèmes modernes.

2.4.2 Mode protégé

Le mode protégé a donc été inventé pour pallier à cette extrême limitation. Le mode protégé se sert de la GDT, qui doit donc impérativement existée. En effet l'adressage s'effectue de la façon suivante :

seg : offset

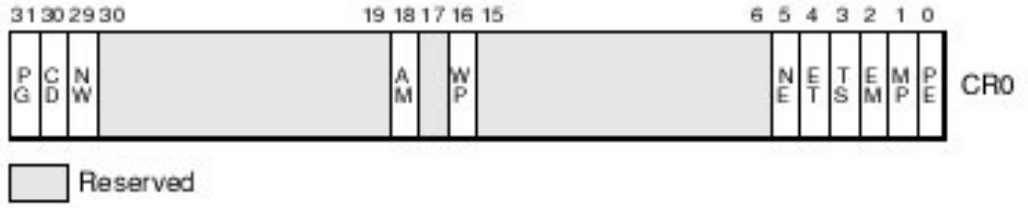
avec :
- **seg** : Sélecteur de segment sur 16 bits ; le segment sélectionné est présent dans la GDT.

– **offset** : Offset (décalage) de segment, sur 32 bits

Ainsi, lorsque le processeur recoit une telle adresse, il obtient une adresse sur 32 bits, permettant ainsi d'adresser jusqu'à 4 Go de mémoire, ce qui est tout de même nettement plus confortable que l'unique Mo du mode réel.

2.4.3 Activation du mode protégé

L'activation du mode protégé s'effectue en changeant la valeur d'un bit partculier dans le registre processeur CR0. Voici à quoi ressemble ce registre :



Le registre processeur CR0[3]

Le bit qui nous intéresse ici est **PE (Protection Enable)**. Quand il est fixé à la valeur 1, il active le mode protégé (valeur 0 = mode réel).

A noter que Grub active lui-même le mode protégé, mais nous avons refait la manipulation, au cas où nous utiliserions un jour un bootstrap ne s'en chargeant pas.

Une fois le mode protégé enclenché, nous pouvons mettre en place la pagination [4].

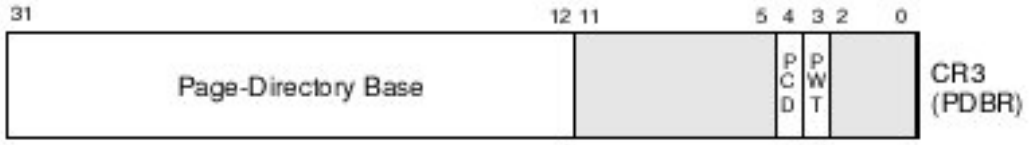
2.5 Chargement de l'espace d'adressage du noyau

Il nous faut à présent charger l'espace d'adressage du noyau, afin d'accéder aux adresses linéaires mises en place lors de la pagination[5]. Ceci est impératif pour pouvoir exécuter le code du noyau, "mappé" en mémoire haute. Le chargement d'un espace d'adressage s'effectue en modifiant la valeur du registre processeur CR3. Voici le registre CR3 :

[3]Documentation Intel Volume 3

[4]Cf. chapitre suivant pour explications détaillées

[5]Cf. chapitre pagination



Le registre processeur CR3[6]

Ce registre contient l'adresse physique du début de la page directory, ainsi que deux flags (PCD et PWT). Ce registre est également appelé PDBR (Page Directory Base Register). Seul les 20 bits les plus significatifs de l'adresse physique de la page directory sont spécifiés, les 12 autres bits étant fixé à 0. En effet l'adresse de la page directory est une adresse de page, et est donc un multiple de 0x1000.

Les flags PCD (Page-level Cache Disable) et PWT (Page-level Writes Transparent) représentent des options de cache interne au processeur. Pour écrire l'adresse de la page directory noyau, il suffit de faire un **mov** en assembleur.

2.6 Activation de la mémoire virtuelle

La dernière étape avant d'exécuter le noyau est l'activation de la mémoire virtuelle. En effet l'environnement est désormais prêt pour l'activation de la mémoire virtuelle (segmentation, pagination, ...), et il faut désormais informer le processeur qu'il doit utiliser ce mode d'adressage afin qu'il réalise les traductions des adresses utilisées (utilisation de la MMU, Memory Management Unit).

Pour ce faire, il faut fixer un bit de plus dans le registre CR0. Cette fois-ci, le bit qui nous intéresse est le bit **PG (Paging)** (bit 31 du registre[7]). Cette opération s'effectue encore une fois en langage assembleur. Nous pouvons à présent "jumper" sur le code du noyau et poursuivre l'exécution.

2.7 "Jump" sur le noyau

Il s'agit de la toute dernière étape de la phase de boot. Elle consiste en quelque lignes d'assembleur, très simples. Mais il reste cependant un point à expliquer. Comme nous l'expliquons au chapitre suivant, le code du noyau a été mappé en adresse haute, à l'adresse 0xc0000000. Cette adresse correspond donc au début du code de l'exécutable du noyau. Cependant il ne s'agit pas de l'adresse sur laquelle nous souhaitons jumper.

[6]Documentation Intel Volume 3
[7]Cf. schéma de CR0

En effet il ne s'agit que de l'adresse de début du header Elf de l'exécutable. Or le header Elf est composé de différents champs donnant des renseignements importants sur l'exécutable. L'un de ces champs, **Entry point**, fournit l'adresse de la première instruction de l'exécutable. C'est cette valeur qui nous intéresse.

Pour la récupérer, il suffit de lire la valeur du champ, situé à l'offset 0x18 du header. On déréférence donc l'adresse **0xc0000000 + 0x18**, et il suffit alors de jumper sur la valeur lue.

Et nous voici enfin dans l'exécution de notre noyau à proprement dit !

Chapitre 3

La pagination

3.1 Définition

La pagination est utilisée au-dessus du mécanisme de segmentation précédemment décrit. Le mécanisme de pagination du processeur divise l'espace d'adressage linéaire (obtenu par la segmentation) en pages. Ces pages de l'espace d'adressage linéaire sont alors "mappées" dans des pages de l'espace d'adressage physique. La pagination offre des mécanismes de protection de pages, pouvant être utilisés à la place ou en complément des mécanismes de protection de la segmentation.

La pagination permet à chaque espace d'adressage de disposer de 4Go de mémoire virtuelle.

3.2 Les structures de données utilisées

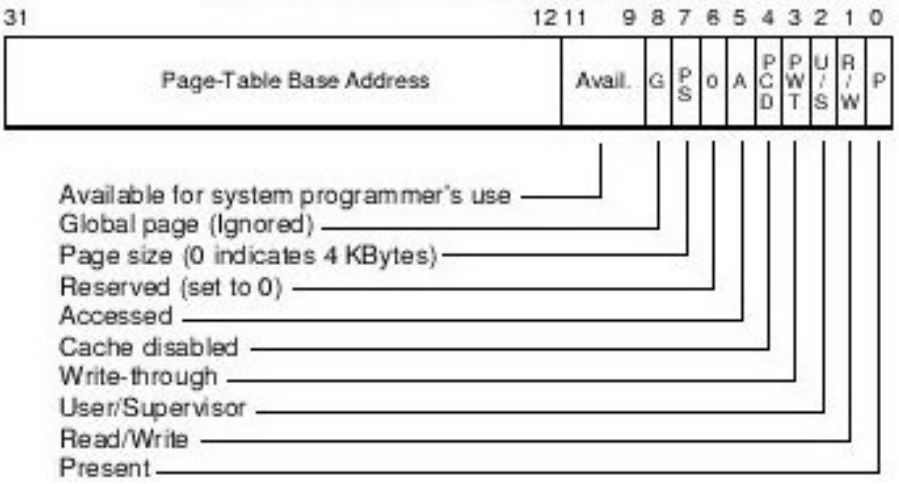
Les informations que le processeur utilise pour traduire une adresse linéaire en adresse physique sont contenues dans trois structures de données :

- Page Directory
- Page Table
- Page

3.2.1 Page Directory

Il s'agit d'une page de 4ko, contenant des PDEs (Page Directory Entrie). Chaque PDE a une taille de 32 bits, soit 4 octets. Une Page Directory peut donc contenir 1024 PDE. Chaque PDE référence une Page Table.

On trouve une et une seule Page Directory par espace d'adressage.



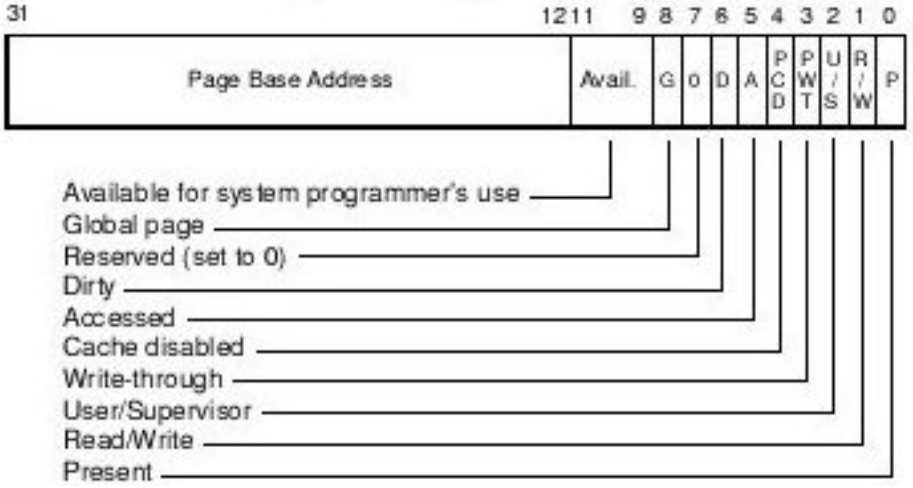
Format d'une entrée de Page Directory [1]

3.2.2 Page Table

Il s'agit d'une page de 4ko, contenant des PTEs (Page Table Entrie). Chaque PTE a une taille de 32 bits, soit 4 octets. Une Page Table peut donc contenir 1024 PTE. Chaque PTE référence une page physique.

A noter qu'une Page Table peut être référencée par une ou plusieurs PDE. Egalement les Page Tables ne sont pas utilisées avec des pages de 4Mo.

Il y a plusieurs Page Tables par espace d'adressage.



Format d'une entrée de Page Table [2]

[1] Documentation Intel Volume 3
[2] Documentation Intel Volume 3

3.2.3 Page

Une page est un espace d'adressage de 4Ko. Les adresses de page sont toujours des multiples de 0x1000.

A noter qu'il est possible d'utiliser des pages de 4Mo, en modifiant le bit PSE du registre CR4. Elles ne sont pas utlisées dans KoinKoin, nous n'en éprouvons pas le besoin actuellement.

3.3 Traduction d'une adresse virtuelle en adresse physique

Quand un programme ou une tâche utilise une adresse virtuelle, le processeur traduit d'abord cette adresse en adresse linéaire, puis utilise le mécanisme de pagination pour traduire cette adresse linéaire en l'adresse physique correspondante.

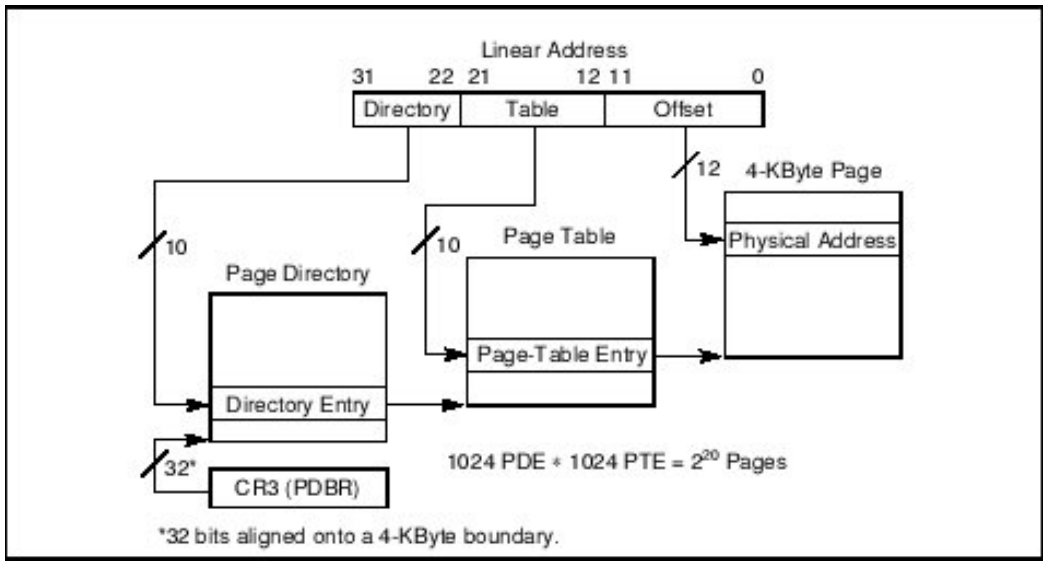
Si cette page n'est pas mappé dans l'espace d'adressage de l'appelant, le processeur génère une exception de type **Page Fault** (exception 14). La valeur de l'adresse inaccessible est alors stocké dans le registre CR2. Un handler a été mis en place dans KoinKoin, renseignant sur la tâche ayant généré l'exception.

Pour minimiser le nombre de cycles bus requis pour réaliser cette traduction, les entrées de Page Directory et de Page table récemment accédées sont stockées dans le cache TLB (Translation Lookaside Buffers). Ainsi le cache TLB permet au processeur de traduire des adresses virtuelles directement.

La traduction de l'adresse linéaire en adresse physique en elle-même s'effectue de la manière suivante :

- Les 10 premiers bits de l'adresse linéaire fournissent au processeur l'entrée de la Page Directory à utiliser. Il y lit l'adresse physique de la Page Table à utiliser.
- Les 10 bits suivants fournissent au processeur l'entrée de la Page Table à utiliser. Il y lit l'adresse physique de la page à utiliser.
- Les 12 derniers bits correspondent au décalage dans la page, ou offset. Le processeur peut alors lire la valeur correspondante.

Voici un schéma résumant ces différentes étapes :



Transcription d'une adresse virtuelle[3]

3.4 Mise en place de la pagination dans KoinKoin

La mise en place de la pagination dans KoinKoin consiste en premier en la mise en place des structures de données (Page Directory et Page Tables) de l'espace d'adressage noyau.

Pour cela une interface a été mise en place, facilitant la création de Page Tables, mais également le référencement.

3.4.1 L'identity mapping ou référencement à l'identique

Après activation de la pagination et donc de la mémoire virtuelle, plus aucune adresse physique n'est utilisable. Or le noyau aura forcément besoin d'accéder à sa propre Page Directory, ainsi qu'à certaines Page Tables, qui ne peuvent pas avoir d'adresse virtuelle. En effet, qui pourrait donc bien mapper la Page Directory du noyau ? Personne, bien évidemment.

Pour résoudre ce problème, on se sert du concept d'identity mapping, ou référencement à l'dentique. Le principe est le suivant : les adresses virtuelles et

[3]Documentation Intel Volume 3

leur adresse physique correspondante sont identiques. Par exemple, l'adresse virtuelle 0x3000 sera transcrit en adressage physique en 0x3000.

Ceci implique donc (en accord avec le mécanisme de traduction d'adresse virtuelle en adresse physique) que la première entrée de la première page table référence la première page physique, la deuxième entrée la deuxième page physique, ... A noter que nous avons limité l'identity mapping aux 1024 premières pages, soit jusqu'à l'adresse 0x400000, afin de n'utiliser qu'une seule Page Table. En effet nous n'avons pas besoin de référencer à l'identique l'ensemble de la mémoire, comme cela a été mis en place sous Linux (alors que cela pose des problèmes de sécurité et masque les erreurs de référencement).

Ainsi le noyau pourra toujours accéder à ses structures non référencées, en les plaçant dans l'identity mapping.

3.4.2 Les Page Tables du noyau

Voici les différentes Page Tables du noyau au moment de l'initialisation de la pagination :

- PTI : permet le référencement à l'identique (Identity Mapping). Cette page table est référencée par l'entrée 0 de la page directory noyau.
- PTM : à l'origine son rôle est surtout de référencer la page table suivante, la PTT. Puis nous avons décidé qu'elle référencerait également toutes les pages servant au gestionnaire de mémoire physique. Cette page table est référencée par l'entrée 1 de la page directory noyau.
- PTT : son rôle est de référencer toutes les autres page tables, y compris la PTM. A noter que la première entrée de cette page table référence la GDT. Cette page table est référencée par l'entrée 2 de la page directory noyau.
- PTK : le rôle de cette page table est de mapper le code de l'exécutable du noyau. Ici, un détail important est à expliquer. En effet, le code du noyau est mappé en adresse haute, à l'adresse 0xc000000.
  Si vous avez bien compris le mécanisme de traduction d'adresse linéaire en adresse physique, vous conviendrez aisément que cette page table ne peut être référencée par une entrée quelconque de la page directory. Après découpage de l'adresse virtuelle du noyau, il apparait que l'entrée de page directory correspondant à l'adresse virtuelle 0xc000000 est l'entrée 768.
- PTAS : cette page table est utilisée par le noyau pour stocker diverses informations correspondant aux espaces d'adressage utilisateurs.
- PTPD : cette page table est utilisée par le noyau pour stocker les adresses virtuelles (ayant une signification dans son propre espace d'adressage uniquement) des Page Directory, ainsi que des PTM et PTT, des

autres espaces d'adressage.

3.4.3 Les Page Tables utilisateur

Voici les Page Tables mises en place lors de la création d'un espace d'adressage utilisateur :

- PTI : ici aussi son rôle est de permettre le référencement à l'identique. Mais contrairement à la PTI du noyau, elle ne va pas référencer l'intégralité des 1024 premières pages. En fait son but est de référencer la GDT, le TSS et l'IDT.
- PTM : Sert à référencer la PTT. Elle ne référence bien évidemment pas les structures de gestion mémoire, comme le noyau.
- PTT : Sert à référencer toutes les Page Tables de l'espace d'adressage.

Ces Page Tables sont référencées au même index dans la Page Directory que dans l'espace noyau.

3.4.4 Utilisation des mécanismes de protection

Les mécanismes de protection fournis par la pagination ont été mis en place. Ces mécanismes apportent une notion de droits sur les pages virtuelles. Ils se définissent en affectant les valeurs correspodantes aux champs "READ/WRITE" et "USER/SUPERVISOR" des PTE référençant la page.

Tout d'abord, elle permet de dissocier deux niveaux d'accès :

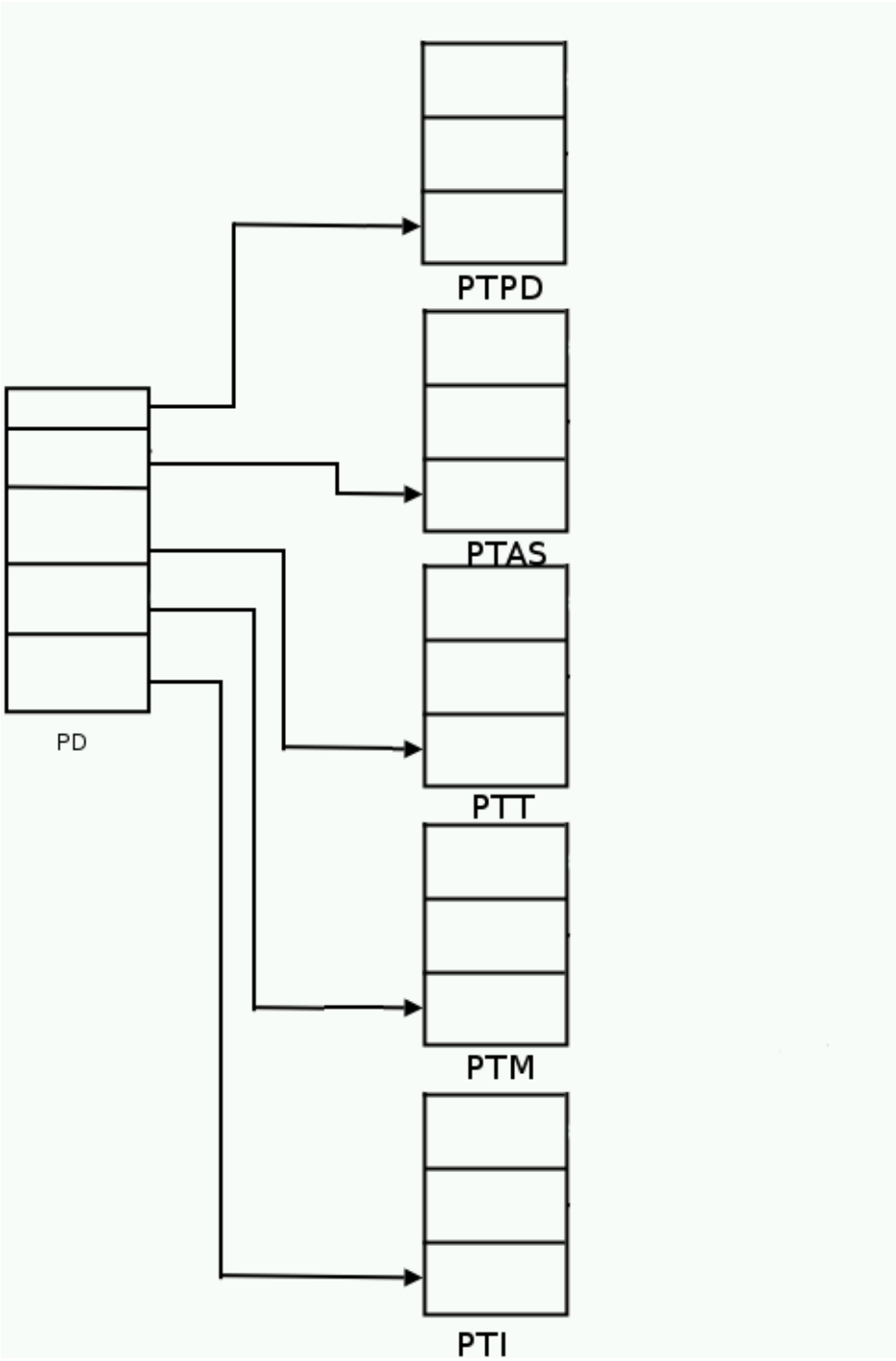
- **SUPERUSER** : La page n'est accessible qu'en mode "supervisor", c'est-à-dire lorsque le processeur fonctionne en CPL[4] 0, 1 et 2 . Or dans KoinKoin seul le mode CPL0 est utlisé, et seul le noyau l'utilise. Seul le noyau peut donc accéder à la page, que ce soit en lecture ou en écriture. Toutes les pages du noyau, quelque soit leur utilisation, sont dans ce mode.
- **USER** : La page est accessible en mode "user" (CPL3), c'est-à-dire par toutes les applications.

De plus il est possible d'appliquer les permissions suivantes :

- **RW** : La page est accessible en lecture et en écriture.
- **RDONLY** : La page n'est accessible qu'en lecture. Une tentative d'écriture dans la page déclenchera une erreur de page (page fault, exception 14). Le code du noyau, qui est mappé à la fois par les espaces d'adressages utilisateur et noyau, possède ces permissions.

[4]Current Privilege Level

3.4.5 Visualisation des structures



Visualisation des structures de pagination

Chapitre 4

La mémoire

4.1 Mémoire physique

4.1.1 Le gestionnaire de mémoire physique

"Son rôle est de conserver la trace de la partie de la mémoire qui est en cours d'utilisation et de celle qui ne l'est pas, d'allouer cette mémoire aux processus qui en ont besoin, de la libérer quand ils ont fait leur travail" [1].

4.1.2 Choix de l'algorithme

Pour le choix de notre algorithme de gestion de la mémoire physique, nous nous sommes basés sur les recherches théoriques menées en la matière par Andrew Tannenbaum dans son livre "Operating System Version 2". Les solutions présentées pouvant être utilisées dans Koinkoin étaient les suivantes :

- Gestion par tableaux de bits
- Gestion par une liste chaînée unique
- Gestion par deux listes chaînée distinctes

La gestion par tableaux de bits utilise un tableau de n bits pour décrire l'état d'une mémoire de 32n bits, un bit à '1' marquant la mémoire comme occupée, '0' comme libre. Cette solution est la plus simple à implémenter, mais "lorsqu'un processus de k unités est chargé en mémoire, le gestionnaire de mémoire doit parcourir le tableau de bits pour trouver une séquence de k bits consécutifs dont la valeur est 0. Or cette recherche est une opération lente (parce que cette zone peut enjamber des mots frontières dans la table), et cela constitue un argument contre les tables de bits"[2]. Nous n'avons donc

[1] Andrew Tanenbaum, "Systèmes d'exploitation 2ème édition"
[2] Andrew Tanenbaum, "Systèmes d'exploitation 2ème édition"

pas opté pour cet algorithme.

La deuxième solution consiste en une liste chaînée unique décrivant l'intégralité de la mémoire. Chaque élément de la liste décrit une zone mémoire (adresse de début, nombre de pages), en la marquant comme libre ou non. Cette solution est plus performante et plus intéressante que la précédente, mais une allocation mémoire requiert de parcourir tous les éléments de la liste, qu'ils soient libres ou non, ce qui représentent des opérations inutiles.

La dernière solution permet une optimisation sur ce point. On utilise alors deux listes distinctes, l'une contenant les éléments décrivant les zones libres, l'autre les zones occupées, permet ainsi une accélération dans l'allocation. De plus les éléments de la liste libre sont triés par taille, du plus petit au plus grand. Ainsi le gestionnaire de mémoire parcourt la liste libre jusqu'à trouver une zone suffisament grande, qui sera par conséquence la plus petite possible, rendant la recherche optimale.

Nous avons donc opté fort logiquement pour la gestion à l'aide de deux listes chaînées distinctes, malgré sa forte complexité.

4.1.3 Définition d'une area

Une "area" est une structure décrivant une zone de mémoire physique ainsi que ses attributs. Ainsi une area possède une adresse de départ, un nombre de pages, des attributs, et un compteur de référence servant à comptabiliser le nombre de personnes se partageant cette zone mémoire.

Les attributs d'une area peuvent être les suivants :
- PM_ATTR_NONE : l'area ne possède aucun attribut particulier.
- PM_ATTR_SHARE : l'area décrit une zone de mémoire partagée entre plusieurs espaces d'adressage.
- PM_ATTR_RESIDENT : l'area décrit une zone de mémoire qui réside en permanence en mémoire.

Les areas sont donc utilisées pour gérer des zones de mémoire physique, qu'elles soient libres ou utilisées, et consituent donc l'élément unitaire du gestionnaire de mémoire.

4.1.4 Les structures de gestion

La gestion de la mémoire physique est une gestion par listes chaînées. La structure du gestionnnaire (t_pm) consiste donc en deux listes chaînées

d'areas :

- la liste "used" contenant les areas utilisées
- la liste "free" contenant les areas libres. A noter que les areas de cette liste sont triées par ordre croissant de taille, optimisant ainsi la recherche d'espace libre.

Lorsqu'une area est libérée (passage de la liste used à la liste free), si la zone mémoire qu'elle décrit est contigue avec une autre area libre, alors les deux areas libres sont fusionnées, afin d'éviter que la mémoire se fragmente en de petites zones libres difficilement utilisables.

4.2 Mémoire virtuelle

4.2.1 Définition

"La mémoire virtuelle repose sur le principe suivant : la taille de l'ensemble formé par le programme, les données et la pile peut dépasser la capacité disponible de mémoire physique. Le système d'exploitation conserve les parties de programme en cours d'utilisation dans la mémoire principale, et le reste sur le disque" [3].

4.2.2 Le gestionnaire de mémoire virtuelle

"Le gestionnaire de mémoire virtuelle fournit un certain nombre de fonctionnalités permettant d'allouer, de libérer des pages virtuelles mais également de manipuler des données en virtuel"[4].

4.2.3 Initialisation

Avant de pouvoir allouer, libérer ou manipuler des données en virtuel dans un espace d'adressage, il convient d'initialiser la mémoire virtuelle de l'espace d'adressage en question. Cette initialisation consiste en la création de la liste d'espace virtuel libre de l'as. Dans koinkoin, cela revient a créer une zone virtuelle (vmarea) débutant à l'adresse 0x1000000.

En effet les adresses inférieures à cette valeur doivent être protégées, puisqu'elles ont étées utilisées par les structures systèmes dépendantes de l'architecture (cf. "Pagination").

[3] Andrew Tanenbaum, "Systèmes d'exploitation 2ème édition"

[4] Sujet k3

4.3 Les espaces d'adressage

"Un espace d'adressage est composé d'une structure de données destinée à contenir suffisamment d'informations pour décrire la mémoire utilisée par un processus, que cette mémoire soit physique, virtuelle, mappée, non mappée, etc..."[5].

Concrêtement, un espace d'adressage est donc composé d'un espace d'adressage physique, et d'un espace d'adressage virtuel :

- Espace d'adressage physique : Regroupe l'ensemble des areas décrivant les zones physiques utilisées par l'espace d'adressage, quelque soit leur nature ou fonction (pages contenant les structures allouées par malloc, le code du module chargé dans l'espace d'adressage, les structures de données utilisées liées à la pagination, ...). Concrêtement il s'agit d'une liste de pointeur sur les areas du système utilisées (liste "used" du gestionnaire de mémoire physique).
- Espace d'adressage virtuel : "Ensemble d'éléments décrivant la mémoire virtuelle en mettant en relation des zones virtuelles avec des zones physiques ou bien simplement des zones virtuelles n'ayant aucune correspondance avec des zones physiques" [6]. Au niveau implémentation, il s'agit d'une liste de vmarea.

Les espaces d'adressage sont décrits pat le type t_as, et sont identifiés par un identifiant unique : l'asid (Adress Space Identifier).

4.4 Allocation de mémoire

Voici les étapes effectuées lors de l'allocation d'espace mémoire par un programme ou un service (en fait par malloc lorsque celui-ci réserve une page supplémentaire ou plus) :

1. **Réservation d'espace physique** : Le gestionnaire de mémoire physique parcourt la liste d'areas libres de la mémoire principale, et réserve l'espace demandé.
2. **Réservation d'espace virtuel** : Le gestionnaire de mémoire virtuelle parcourt la liste d'areas virtuelles de l'espace d'adressage dans lequel on effectue l'allocation, et réserve l'espace demandé.

[5]Sujet k2
[6]Sujet k3

3. **Référencement ou mapping** : Cette dernière étape consiste à faire correspondre les zones mémoires physiques et virtuelles précédemmment réservées. En d'autres termes, on veut que l'adresse virtuelle réservée référence l'adresse physique réservée.

   Pour cela, on découpe l'adresse virtuelle afin d'obtenir les index correspondant dans la Page Directory et la Page Table, et on écrit l'adresse physique réservée dans l'entrée de Page Table calculée.

A noter que ces actions sont bien évidemment intégralement effectuées par le noyau.

4.5 Partage de mémoire

La conception adoptée pour la gestion de la mémoire, à savoir listes chaînées d'areas, nous a permit de mettre en place facilement une fonctionnalité de mémoire partagée, entre des espaces d'adressage différents.

En effet, comme décrit précédemment[7], chaque area possède ses propres attributs, parmi lesquels PM_ATTR_SHARE, qui marque la zone mémoire physique comme partageable.

Entre alors en jeu le champ "compt_ref" de la structure area, qui comptabilise le nombre d'espace d'adressage référençant la zone physique. En effet, si une area est marquée comme partageable, ce champ est incrémenté à chaque référencement supplémantaire de la zone (c'est-à-dire à chaque appel de vm_map() sur cette zone). Ainsi on connaît en permanence le nombre d'espace d'adressage référençant la zone.

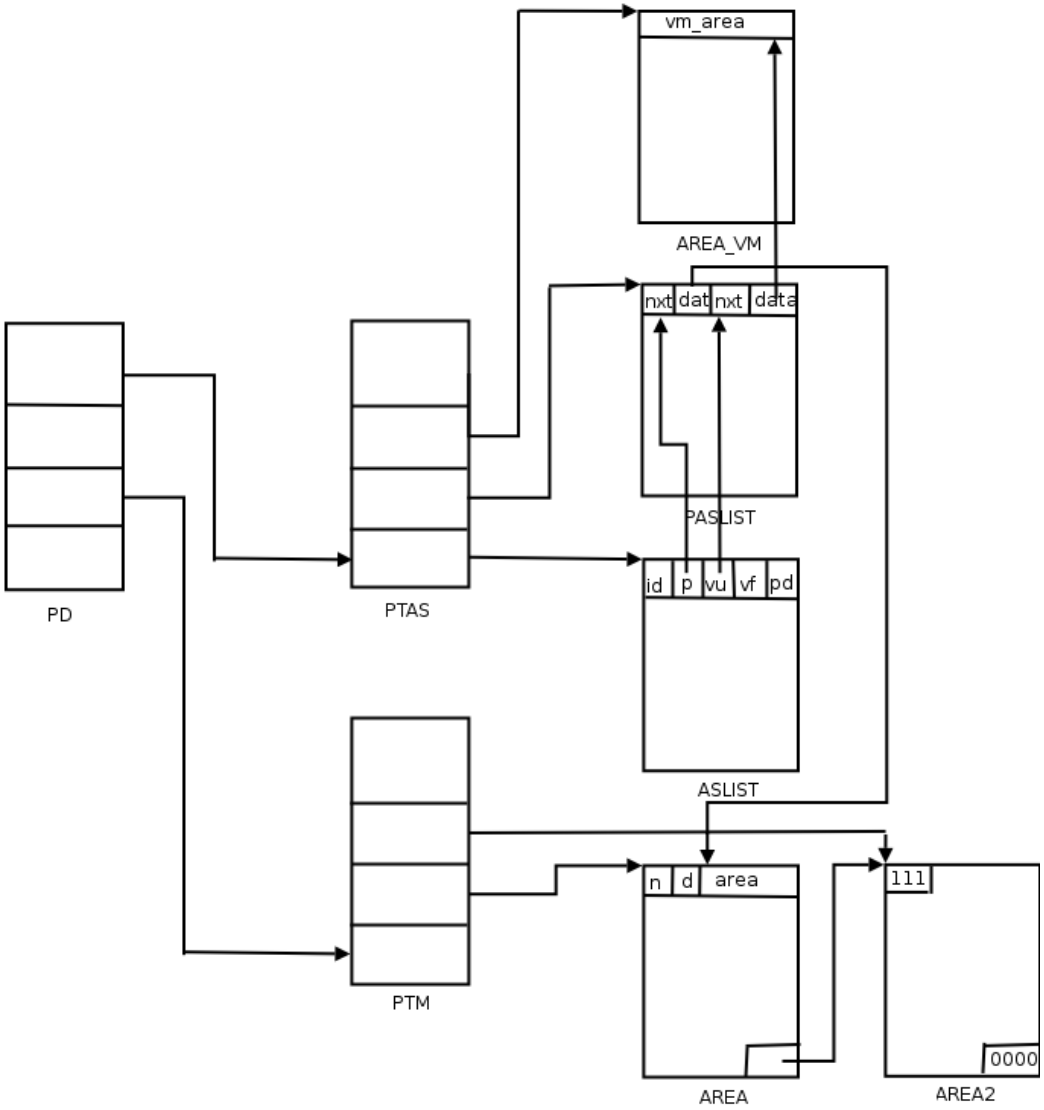
Lorsqu'un espace d'adressage souhaite libérer la zone partagée, le gestionnaire de mémoire vérifie la valeur du champ "compt_ref". Se présente deux cas possibles :

1. La valeur est de 1, ce qui signifie que l'espace d'adressage était seul à référencer la zone physique. Elle est peut donc être libérée, et marquée comme libre (donc utilisable). Les gestionnaires de mémoire libère donc les areas physiques et virtuelles.
2. La valeur est supérieure à 1, ce qui signifie que d'autres espaces d'adressage utilisent encore la zone. Il ne faut donc pas la supprimer. Le gestionnaire de mémoire libère alors seulement la vmarea de l'espace d'adressage référençant la zone, et décrémente la valeur de "compt_ref".

[7] Cf. partie 5.1.3

4.6 Implémentation

Voici une représentation des différentes structures servant à la gestion de la mémoire (physique et virtuelle) et des espaces d'adressage, au niveau des structures de données :



Chapitre 5

Le gestionnaire d'ensembles

5.1 Présentation

Nous avons vu dans le chapitre concernant la mémoire qu'il est possible d'implémenter une gestion des listes chaînées avant même d'avoir une réelle gestion de la mémoire. Cependant, cette implémentation dépend fortement de la gestion de la mémoire et ne peut donc pas être utilisée pour d'autres choses.

Maintenant que nous disposons d'un gestion de mémoire évoluée fournissant des fonctions de haut niveau facilement utilisables, nous pouvons envisager d'implémenter un gestionnaire pour ce type de structure d'une façon plus générique et réutilisable. De plus, nous aimerions pouvoir gérer plusieurs types de structures de données, qui nous serviront tour à tour en fonction des besoins spécifiques des autres services ou programmes utilisateurs. Enfin, nous aimerions disposer d'une interface unifiée pour la gestion de ces différents types d'ensembles, pour faciliter leur utilisation et également pour pouvoir changer très facilement de type d'ensemble si nous nous apercevions qu'un autre type d'ensemble est plus approprié pour un traitement donné. C'est tout cela que propose le gestionnaire d'ensemble.

En effet, celui-ci encapsule des gestionnaires de chaque type d'ensemble proposé, en permettant de stocker des objets ou encore de les organiser à l'intérieur de l'ensemble, tout cela très simplement et de façon transparente pour l'utilisateur. Il suffit à l'utilisateur de préciser le type d'ensemble qu'il souhaite utiliser et la méthode de tri et le gestionnaire d'ensemble fera tout le reste tout seul.

Le gestionnaire d'ensemble est une bibliothèque. Actuellement, elle est utilisée uniquement par le noyau, mais il sera très facile de l'en rendre indépendante afin de pouvoir la lier dynamiquement à n'importe quel service ou

programme utilisateur. Ainsi, les programmes n'auront pas besoin de gérer des structures de données en interne, tous ce qu'ils auront a faire est d'appeler des fonctions du gestionnaire d'ensemble.

Nous verrons que l'interface fournie par notre gestionnaire d'ensemble est assez proche de la gestion des ensembles fournie par la bibliothèque standard du C++.

5.2 Implémentation et interface

5.2.1 Structure définissant un ensemble

La structure définissant les ensembles, de type t_set, regroupe les informations suivantes :

- setid : Identifiant de l'ensemble, que nous utiliserons dans toutes les fonctions pour désigner l'ensemble sur lequel on souhaite travailler.
- type : Définit le type d'ensemble utilisé. Nous verrons dans la section suivante quelles valeurs il peut prendre.
- sort : Définit comment doivent être triés (ou pas) les nouveaux éléments ajoutés à l'ensemble. Nous verrons dans une section suivante les modes de tri possibles.
- objectsz : Taille d'un objet de l'ensemble. Cette information est utile pour certains types d'ensembles seulement.
- initsz : Taille initiale de l'ensemble, c'est à dire le nombre d'éléments qu'il contiendra à sa création.
- data : Pointeur permettant de retrouver les objets de l'ensemble. Par exemple, dans le cas d'une liste chaînée, ce sera un pointeur vers le premier noeud de la liste.

5.2.2 Types d'ensembles

Tous les types d'ensembles qu'on peut imaginer peuvent être encapsulés dans le gestionnaire d'ensemble. Nous avons défini quelques types qui peuvent ou pourront être utilisés en passant par le gestionnaire d'ensemble. Ces types sont définis dans une énumération nommée t_type_set qui contient actuellement les valeurs suivantes :

- SET_TYPE_ARRAY : Tableaux
- SET_TYPE_LIST : Listes chaînées
- SET_TYPE_DLIST : Listes doublement chaînées
- SET_TYPE_BTREE : Arbres équilibrés
- SET_TYPE_STACK : Piles (LIFO)

– SET_TYPE_QUEUE : Files (FIFO)

Cette liste n'est bien sûr pas exhaustive, la seule limite de types d'ensembles étant notre imagination ou nos besoins.

Cependant, deux types d'ensembles sont actuellement réellement implémentés et utilisables : les listes chaînées et les listes doublement chaînées.

5.2.3 Normalisation des objets contenus dans les ensembles

Le gestionnaire d'ensemble est capable aussi bien de manipuler des objets simples comme des identifiants que des objets complexes comme des structures. La seule contrainte pour que ces objets puissent être utilisés par le gestionnaire d'ensemble est qu'il doivent fournir un identifiant, unique pour chaque objet dans un ensemble donné. Pour une structure, le premier champ devra être l'identifiant de l'objet. Cela permet au gestionnaire d'ensemble d'ignorer le type réel de l'objet. Dans le cas contraire, cela complexifierait son implémentation. La seule connaissance d'un identifiant lui permet donc de manipuler tous les objets de la même façon, sans connaître leur type réel.

5.2.4 Modes de tri

Il existe trois modes de tri possibles, que l'utilisateur choisit à la création de l'ensemble :

– SET_SORT_ENABLE : Tri automatique en fonction de l'identifiant des objets.
– SET_SORT_DISABLE : Aucun tri.
– SET_SORT_MANUAL : Le tri des objets sera fait par le programme utilisant les ensembles, selon des critères que le gestionnaire d'ensembles ne peut pas connaître.

5.2.5 Les itérateurs

De même qu'en C++, nous avons une structure d'itérateur, nommée t_iterator, qui permet d'accéder facilement aux objets d'un ensemble, encore une fois de façon unifiée, et qui en particulier facilite les parcours d'ensembles. Cette structure contient simplement un identifiant d'objet et un pointeur vers l'objet lui-même.

Cela nous permet notamment d'avoir une macro, nommée SET_FOREACH, qui fonctionne de la même façon que le mot clé foreach en C++.

5.2.6 Fonctions

- set_init : Initialise la gestion des ensembles.
- set_clear : Libère toutes les ressources utilisées par les ensembles.
- set_rsv : Réservation d'un ensemble en lui spécifiant tous les paramètres nécessaires à remplir sa structure t_set. Elle nous donne l'identifiant de l'ensemble ainsi créé.
- set_rel : Supprime un ensemble.
- set_get : Permet de trouver un objet dans l'ensemble grâce à l'identifiant de cet objet. Elle nous donne un itérateur sur cet objet.
- set_head, set_tail, set_prev, set_next : Permet d'obtenir un objet d'un ensemble, respectivement le premier objet de l'ensemble, le dernier objet de l'ensemble, l'objet qui est avant ou après un autre objet. Ces fonctions nous donnent également un itérateur sur l'objet souhaité.
- set_insert, set_insert_head, set_insert_tail, set_insert_before, set_insert_after : Insère un objet dans l'ensemble, respectivement de manière automatique, au début ou à la fin de l'ensemble, avant ou après un autre objet.
- set_delete : Supprimer un objet de l'ensemble.

5.3 Évolutions futures

Comme vous l'avez déviné, les évolutions les plus évidentes du gestionnaire d'ensemble constiteront à implémenter un gestionnaire pour d'autres types d'ensembles que ceux actuellement proposés.

De plus, nous pourrons également envisager de transformer le gestionnaire d'ensemble en un service, auquel il suffira d'envoyer des messages pour qu'il gère les ensembles nécessaires aux programmes. Cela permettra alors de partager très facilement des données entre plusieurs programmes distincts, encore une fois grâce une interface très simple et unifiée qui sera fournie par ce gestionnaire.

Bien sûr, une gestion de droits devra accompagner cette possibilité de partages, pour des raisons évidentes de sécurité.

Chapitre 6

Tâches et threads

6.1 Les tâches

6.1.1 Définition

KoinKoin ne dispose pas de processus au sens UNIX, néanmoins nous utiliserons parfois ce terme pour désigner une tâche. Une tâche est en fait un conteneur, une entité composée de fils d'exécution et de ressources (mémoire, cpu, ...).

6.1.2 Processus de création d'une tâche

Le processus de création d'une tâche se découpe en trois phases :

1. Réservation d'une structure t_task
2. Attachement d'un espace d'adressage à la tâche
3. Attachement d'un thread à la tâche

La dernière étape peut être répétée autant de fois que voulue.
A noter que lors de la création d'une tâche, celle-ci est marquée comme arrêtée.

6.1.3 Avantages

La particularité d'un tel système est son extrême modularité. Avec celui-ci il est possible de créer tous les espaces d'adressages puis de les rattacher à la toute fin.

De plus il est possible de détacher un espace d'adressage ou mêmes des threads pour les ré-attacher à d'autres tâches, tout cela facilitant énormément le recyclage de structures internes au noyau : **t_as**, **t_thread**, **t_task**, etc...

A noter qu'il existe une structure tâche noyau, la **ktask**. Cela n'offre en théorie aucun intérêt, mais nous permettra de créer ultérieurement, si nous le désirons, des threads noyau.

6.1.4 Priorité de tâche

Les tâches peuvent différenciées par leurs comportements. Pour cette raison chaque tâche se voit associer un comportement **behav**. Ce comportement introduit un interval de priorités pour la tâche, celle-ci pouvant évoluer librement dans celui-ci.

Comme nous venons de le dire la tâche peut donc retoucher sa priorité. Pour résumer, une tâche est toujours associée à :

- un comportement **behav** identifiant le degré de priorité désirée.
- une priorité courante **prior**, celle-ci étant constamment située dans l'interval associé au comportement.

Voici les différents comportements qu'une tâche peut prendre :

behaviour	**interval**	**default priority**
KERNEL	[210, 250]	230
REALTIME	[170, 210]	190
INTERACTIVE	[130, 170]	150
TIMESHARING	[50, 130]	90
BACKGROUND	[10, 50]	30

Le type **behav** ne définit pas la priorité de la tâche vis-à-vis de l'ordonnanceur mais bien l'interval dans lequel la tâche va pouvoir évoluer.

6.1.5 Classe de tâche

En plus d'avoir un comportement et une priorité courante, chaque tâche se voit attribuer une **classe** lors de sa création.

La **classe** définit les droits généraux de la tâche vis-à-vis du système, c'est-à-dire les opérations autorisées sur le système : le processeur comme les périphériques.

Les différentes classes de tâches sont les suivantes :

- CLASS_KERNEL
- CLASS_DRIVER

– CLASS_SERVICE
– CLASS_USER

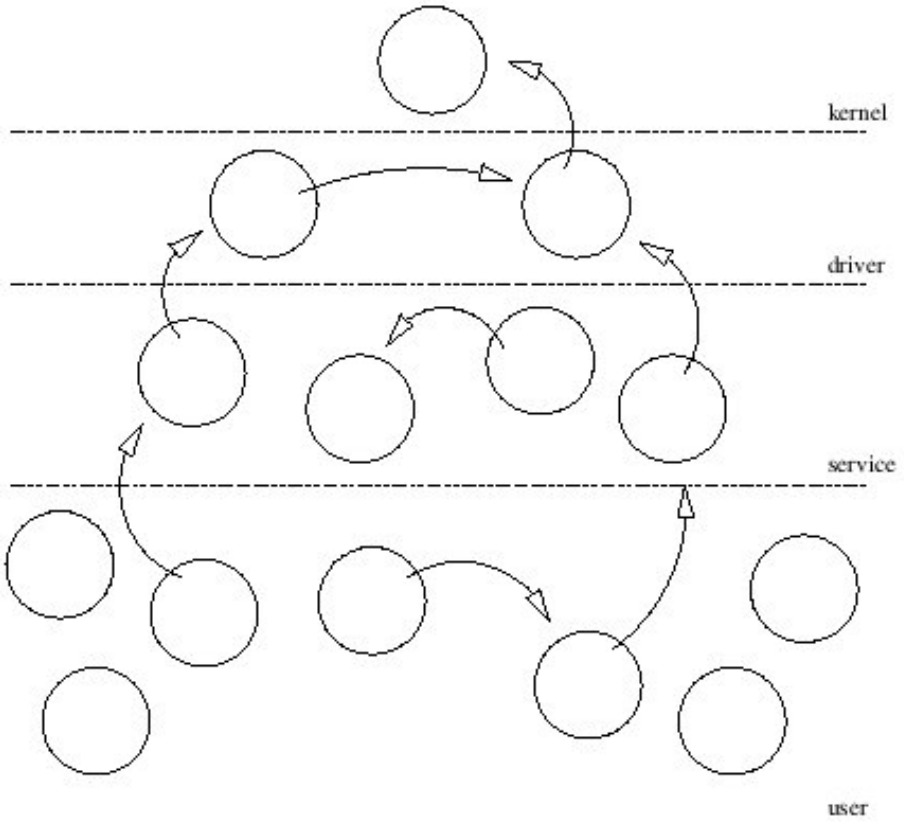
Le fait d'appartenir à telle ou telle classe ne signifie rien mis à part que dans certaines classes il est possible d'effectuer certaines opérations privilégiées.

De plus, ces classes permettent de "trier" les tâches, ceci étant obligatoire pour pouvoir établir un système hiérarchique.

Visualisation de la hiérarchie

Comme nous venons de le voir, les tâches sont toutes distinguées par rapport à leur classe. Grâce à cela le système peut être divisé en couches, chaque couche ayant des droits différents vis-à-vis du système.

Voici une visualisation des couches de classes de KoinKoin :



A noter que les appels entre les couches sont normalisés pour coller au système hiérarchique. Les tâches peuvent appeller une tâche disposant de privilèges supérieurs ou égaux aux siens.

Par exemple un processus peut demander à un service ou au noyau de faire quelque chose pour lui. En revanche il n'arrivera jamais que le noyau demande à un pilote de faire quelque chose pour lui.

6.1.6 Le gestionnaire de tâches

Le noyau a la responsabilité de la gestion des tâches. Il est donc le seul à pouvoir rendre les services de création, suppression, changement détat, etc... concernant les tâches.

Le gestionnaire de tâches dispose d'une variable globale représentant l'ensemble des tâches du systèmes (ensemble de type liste chaînée).

L'ordonnanceur, le gestionnaire de tâches et le gestionnaire de threads, évoluant ensemble, peuvent accéder à la liste des tâches en demandant au gestionnaire de tâche l'idenyifiant de l'ensemble représentant les tâches du système : **task_setid()**.

6.2 Les threads

6.2.1 Définition

Un thread est un fil d'exécution, permettant de séparer les ressources et l'exécution.

"Le thread inclut : un compteur ordinal qui effectue le suivi des instructions à exécuter ; des registres qui détiennent ses variables de travail en cours ; et une pile qui contient l'historique de l'exécution, avec une frame pour chaque procédure appelée mais n'ayant encore rien retourné. Bien qu'un thread doive s'exécuter dans un processus (une tâche), ils représentent deux concepts distincts qui peuvent être traités séparemment. Les processus (les tâches) servent à regrouper les ressources ; les threads sont les entités planifiées pour leur exécution par le processeur."[1]

6.2.2 Implémentation dans KoinKoin

L'implémentation des threads dans KoinKoin respecte la définition énoncée par Andrew Tanenbaum, à savoir :

- Chaque thread possède sa propre pile (taille par défaut : 1 page)
- Tous les threads d'une même tâche se partage l'ensemble des ressources de la tâche (espace d'adressage, variable globale, etc...)
- Chaque thread possède son propre compteur ordinal

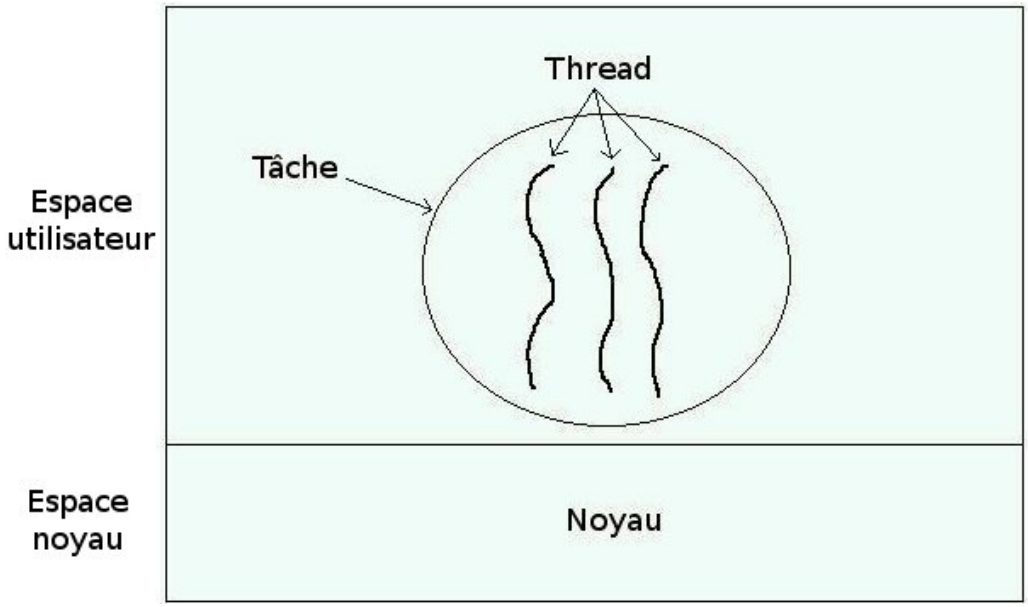
[1] Andrew Tanenbaum

- Chaque thread possède sa propre copie des registres du système ; cette copie se trouve dans la structure **t_arch_dep_cpu**, qui stocke les valeurs des registres EAX, EBX, ECX, EDX, ESI, EDI, SS, CS, EIP, ...
- L'exécution de chacun des threads d'une tâche est indépendant, dans le sens où chacun d'entre eux possède son propre statut d'ordonnancement (RUN, STOP, WAIT, ...)
- Chaque thread possède sa propre priorité[2].

De plus, chaque thread possède une liste des threads en attente de sa fin d'exécution. Ainsi, au moment de sa mort, les threads en attente pourront facilement être réveillés.

6.3 Visualisation

Voici une représentation graphique du concept de tâche et de thread :



Représentation d'une tâche multithreadée

[2]Cf. Priorité de tâche

Chapitre 7

La notion de module

7.1 Définition

Un module est une entité particulière car celle-ci est passive. En effet un module en soit ne vit pas, ne s'exécute pas. Un module est un exécutable stocké en mémoire principale plus ou moins longuement dans le temps. Par exemple les exécutables des services ou des pilotes, bref des tâches fondamentales, seront sauvegarder constamment en mémoire principale afin que celles-ci puissent être lancées et relancées très rapidement et sans aide particulière.

7.2 Explication de la structure t_module

7.2.1 Les différents champs

Chaque module possède son propre numéro identifiant unique, le **modid**, attribué par le gestionnaire de modules. La structure indique aussi l'adresse physique à laquelle le module est stocké en mémoire principale, ainsi que le nombre de pages mémoire qu'il occupe, permettant de le mapper aisément dans un espace d'adressage (c'est-à-dire le charger). A noter qu'un module peut bien évidemment être chargé dans plusieurs espaces d'adressage simultanément, grâce à la fonctionnalité de mémoire partagée.

Chaque module possède également un nom permettant de l'identifier de faqn unique, ce nom représentant le chemin de l'exécutable sur le système.

7.2.2 La durée de vie

Chaque module se voit attribué une durée de vie nommée **lifetime**.

Grâce à cette durée de vie, certains modules resteront chargés en mémoire jusqu'à l'arrêt du système alors que d'autres se verront supprimés dès qu'ils ne seront plus utilisés.

Bien entendu il est logique de penser que les modules correspondants aux services fondamentaux auront une durée de vie infinie.

Imaginons que le pilote IDE crashe à cause d'un bug. Il est clair que les systèmes de fichiers ne pourront fonctionner sans pilote IDE. Il sera donc difficile de prendre l'exécutable du pilote IDE sur le système pour le charger et le relancer puisque la gestion des fichiers sera immobilisée.

Grâce au gestionnaire de modules[1] et au mécanisme de durée de vie, le pilote IDE pourra être relancé sans difficulté car l'exécutable sera déjà chargé en mémoire principale.

Les différentes durée de vie sont :
- LIFETIME_INFINITE
- LIFETIME_FINITE

[1] décrit dans la partie "Services"

Chapitre 8

Interruptions et traps

8.1 Introduction

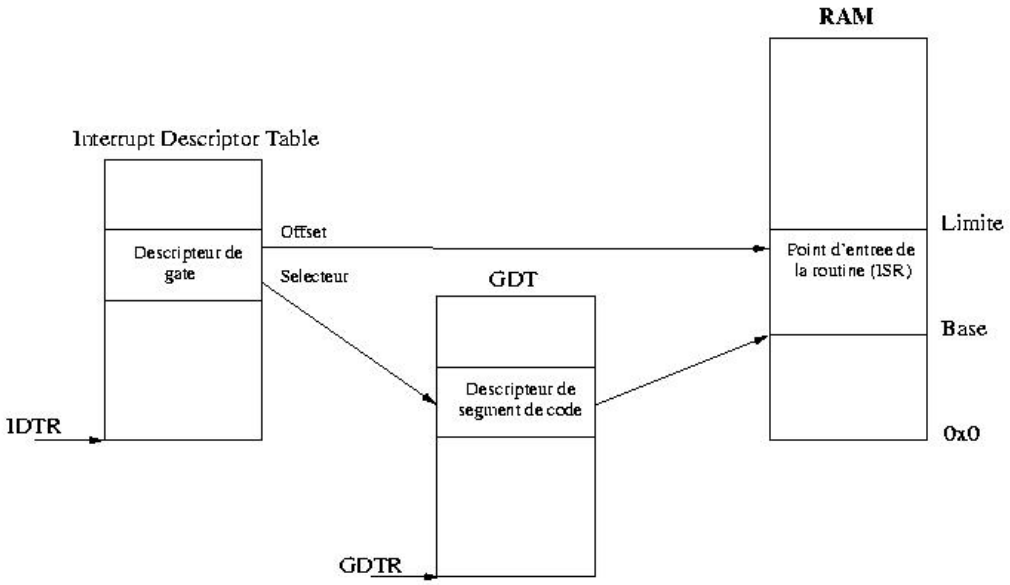
Les interruptions (de manière générique) est une partie très dépendantes de l'architecture matérielle. Nous nous sommes attachés à l'architecture intel, cependant grâce à nos méthodes de généricité il est simple d'ajouter de nouvelles architectures.

Les interruptions sont les seuls fonctions vraiment dynamiques. En effet, le processeur jump sur les fonctions spécifiés dans l'idt au moment où un évenement intervient. Les interruptions, qui sont au nombre de 256, peuvent être séparer en 3 groupes :

- Les exceptions : ce sont les évenements processeurs, ils sont au nombre de 32
- Les interruptions matérielles ou IRQ : ce sont des évenements liés aux matériels connectés à la carte mère. Il y en a 16.
- Les interruptions logicielles : ce sont des interruptions définis par le système d'exploitation, il y en a 256 - 48 donc 208.

8.1.1 l'idt

Une table permet d'associer à chaque interruption une routine à executer, c'est la table IDT (Interrupt Descriptor Table). A chaque interruption est associe un "vecteur" qui est un index dans cette table afin de déclencher la bonne procédure.



Représentation du fonctionnement de l'idt et des interruption

L'idt est un tableau contenant des entrées spécifiant l'adresse mémoire où se trouve la fonction [1] liée à l'interruption.

La table IDT contient des descripteurs systèmes appeles 'gates'. Il y a plusieurs types de gates, mais pour traiter les interruptions materielles, les exceptions et les interruptions logicielles dans notre cas, on utilise classiquement les 'interrupt gate'. Un descripteur est une structure qui pointe sur une routine.
Un descripteur a une longueur fixe de 64 bits.
Il y a 3 types de descripteurs :

- **trap gates** : Type le plus simple. On donne l'adresse de la routine (32 bits), le segment de code à utiliser, et le privilège requis pour exécuter l'interruption.

- **interrupt gates** : Similaire au trap gate, mais un bit change, signifiant que la routine associée est non interruptible. Les interruptions vont être désactivées avant l'appel à la routine.

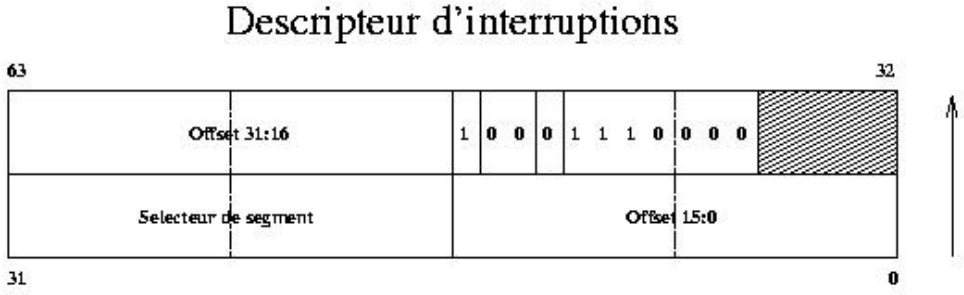
- **task gates** : Les "task gate" référence un « TSS descriptor » de la GDT. C'est à dire que lorsque le processeur recevra l'interruption, il effectuera une commutation de tâche c'est à dire le mécanisme de préemption, habituel des systèmes multitâches.

[1] qu'on appellera par la suite les low handler

Nous utilisons uniquement les interrupt gate pour plusieurs raisons :

1. Notre modélisation à plusieurs couches nous permet de nous passer des task gates. Ainsi nous sauvegardons les registres sans avoir besoins du tss. Si nous utilisions les tss, il nous en faudrait 1 par tâche ce qui encombrerait la mémoire inutilement[2]
2. Nous n'utilisons pas les trap gates car nous souhaitons que nos interruptions ne soient pas interuptible. Cela impliquerait la mise en place de système d'accès concurrent sur les appels systèmes, ce qui est fastidieux et pas obligatoirement plus efficace.[3]

Voici la description des entrées de l'idt :



Représentation d'une entrée de l'idt

Le sélecteur de segment va permettre de trouver le bon descripteur de segment et l'offset sur 32 bits et en fait une adresse linéaire pointant sur le handler. Le champ DPL permet de déterminer à quel degré de privilège l'interruption peut être exécuté. Nous remplissons ainsi l'idt avec les handler de bas niveau (les wrappers)[4] Voici les étapes de la vérification des droits l'ors des interruptions par l'unité de controle :

1. Elle détermine le vecteur i associée à l'interruption ou exception.
2. Elle détermine le descripteur de segment de code où se trouve le handler.
3. Etape sécurité :
   – Vérification que le CPL[5] >= DPL[6] sinon General protection.
4. Changement éventuel de pile si CPL <> DPL.

[2]Cette méthode est la manière générique de gérer les interruptions mais elle n'est plus apprécié

[3]attention ce n'est pas parce que nos interruptions ne sont pas interruptible que nous en perdons elles sont sauvegardées et levées juste après

[4]cf section Les wrappers

[5]CPL : current privilege level

[6]privilege level défini dans la GDT

5. Sauvegarde eflags, cs, eip.
6. Elle charge dans cs et eip le segment et l'offset contenu dans le "gate desciptor".
7. Lorsque le handler est terminé l'exécution iret va procéder à la restauration de contexte.
8. Dépilement de cs, eip, eflags.
9. Vérifie si le CPL <> DPL du handler si c'est le cas on restaure ss et sp qui ont été sauvegardé lors de l'étape 4.
10. fin

8.1.2 l'idtr

Nous avons créé la table IDT et l'avons rempli. Il est donc important de pouvoir trouver son adresse. Pour cela, il y a un registre IDTR qui contient l'adresse de l'IDT. Le registre IDTR, de longueur 48 bits, se compose de deux champs. Le premier champ contient l'adresse du début de l'IDT en mémoire. Le second champ correspond à la limite de l'IDT en mémoire, et est calculé de la manière suivante : IDT_limit = taille IDT - 1. Grâce à ce registre, on peut placer l'IDT dans n'importe quel emplacement mémoire.
2 instructions assembleurs sont nécessaires aussi :
- sidt : pour stocker l'adresse de la table
- lidt : pour récupérer l'adresse de la table

8.2 Les différents types d'interruptions

Comme nous l'avons présenté précédement il existe différents types d'interruptions. Nous allons les détailler dans les sections suivantes.

8.2.1 Les exceptions

Les exceptions sont les interruptions déclenchées par le processeur.
Pour les développeurs intel une exception est :

```
An exception is an event that typically occurs when an instruction causes
 an error.
For example, an attempt to divide by zero generates an exception.
However, some exceptions, such as break-points, occur under other
conditions. Some types of exceptions may provide error codes. An
error code reports additional information about the error.
```

`An example of the notation used to show an exception and error code is shown below.`

La liste des exceptions est la suivante :

- **Division by Zero** : Tentative d'exécution de l'instruction DIV ou IDIV avec pour diviseur 0.
- **Debug / Single Step** : Utilisé en parallèle du registre de debug, l'interruption est levé lorsque l'execution arrive sur un "breakpoint hit". Breakpoint
- **Nmi interruption** : Cette interruption est utilisée pour surveiller l'alimentation du système. Dès que le proxesseur détecte une baisse de tension (qui peut annoncer une coupure de l'alimentation), cette interruption est levé.
- **Breakpoint** : Cette interruption attend une exception breakpoint.
- **Overflow** : Cette interruption est levé lorsqu'une intruction arithmétique est executé avec un nombre signed puis que l'instruction INTO est executé (En fait, le processeur met le OF flags à 1 lors de l'opération arithmétique, ce flag est testé lors de l'execution INTO).
- **Bounds Check** : L'instruction BOUND est utilisé pour vérifier la table de l'index des tableaux. Si la limite est exceder cette exception est levée.
- **Invalid Opcode** : Cette exception est levé si le processeur exécute un opcode réservé ou que le préfixe LOCK est mal utilisé.
- **Device Not Available** : Cette interruption intervient lorsqu'on exécute une "floating-point" instruction alors que l'on a pas un biprocesseur.
- **Double Fault** : Cette interruption intervient lorsque deux exceptions sont levés exactement au même moment ou quand il n'y a pas d'entrée dans l'IDT qui correspond à l'exception qui intervient.
- **coprocesseur segment overrun** : Pour les coprocesseurs, attention cette exception ne peut être masquer.
- **Invallid TSS** : Cette exception est levé lorsque le TSS (dont on spécifie l'adresse dans la gdt) est mal formé[7]
- **Segment not found** : Cette interruption intervient lorsque l'on utilise une adresse linéaire et que le numéro du segment ne correspond à aucune entrée de la gdt.
- **Stack Segment Fault** : Cette exception est levée lorsque une operation de pile dépasse l'offset (de 0xFFFF) ou qu'il n'y a pas de segment dans le registre SS.
- **General protection** : Cette exception correspond au dépassement des registres CS, DS, ES, FS, GS. Elle est levé pour d'autres raisons de

[7] cf contexte switch -> TSS

droits (changement de CPL) sur les segments.
- **Page Fault** : Cette erreur intervient lors de problème de pagination. C'est à dire lorsque l'on essaie d'accèder à une adresse non mappé ou lorsque l'on ne respecte pas les droits R/W sur les segments.
- **Les interruptions réservés par intel** : Les interruptions 15 et de 19 à 31 sont des entrées que Intel a réservé pour leur futur développement.
- **Floating point error** : Cette exception intervient lorsque l'un des opérandes devant être représenter sous forme à virgule flottante (écriture exponentielle des nombres[8]). Elle peut aussi être levée par les multiprocesseur avec Divide-by-Zero, Underflow, Overflow...
- **Alignement check** : Problème d'alignement mémoire.
- **Machine Check** : Problème de RAM ou problème matériel.

8.2.2 Les interrutions matériels IRQ

Les interruptions matérielles interviennent à la demande du matériel ayant une IRQ (usb, clavier, ...). Mais comment le processeur est prévenu ?

Le Pic

Les interruptions matérielles sont générées par les différents composants matériels d'un ordinateur et parviennent au processeur par l'intermédiaire du contrôleur d'interruption 8259, appelé aussi "Programmable interrupt controler" (PIC).

Ces interruptions sont les sonnettes que tirent les périphériques pour dire au processeur que quelque chose se passe. Mais si chaque peripherique pouvait envoyer directement un signal au processeur, il faudrait sur celui-ci autant de broches (jouant le role de cordons de sonnette) que de périphériques. Pour éviter cela, les périphériques envoient leur requête d'interruption à une puce à laquelle ils sont connectés, le contrôleur d'interruption. C'est ce contrôleur qui va envoyer à leur place une interruption au processeur. Les requêtes faites par le contrôleur sont appelées IRQ (Interrupt Request).

[8]Cf cours d'architecture, exponentiel tend vers 0 en $-\infty$ donc si le nombre est trop proche de 0 l'exposant de l'exponentiel doit être trop grand

Les interruptions

8.2.3 Les interruptions logicielles

8.3 Level 0 : Les wrappers

8.4 Level 1 : Les handlers

8.5 Level 2 : Les traps

8.5.1 L'intéret des traps

8.5.2 Le fonctionnement des traps

Chapitre 9

L'ordonnanceur

9.1 Définition

"Lorsqu'un ordinateur est multiprogrammé, il possède fréquemment plusieurs processus en concurrence pour l'obtention de temps processeur. Cette situation se produit chaque fois que deux processus ou plus sont en état prêt en même temps. S'il n'y a qu'un seul processeur, un choix doit être fait quant au prochain processus à exécuter. La partie du système d'exploitation qui effectue ce choix se nomme l'**ordonnanceur** (scheduler) et l'algorithme qu'il emploie s'appelle **algorithme d'ordonnancement**."[1]

9.2 Fonctionnement dans KoinKoin

L'interface de l'ordonnanceur est simple mais permet néanmoins la gestion des priorités sur les tâches comme sur les threads.

A noter que le gestionnaire de tâches, le gestionnaire de threads et l'ordonnanceur évoluent ensemble et modifient tous la structure principale de l'ordonnanceur pour maintenir l'ordonnancement à jour en ce qui concerne les priorités par exemple.

Une chose importante : l'ordonnanceur s'occupe d'ordonnancer des threads et non des tâches. En effet dans KoinKoin, une tâche est considérée comme un conteneur alors qu'un thread est une entité active.

L'ordonnanceur, pour pouvoir correctement décider du temps à attribuer à chaque tâche suivant sa priorité, doit garder un certain nombre de valeurs constamment à jour, notamment : le nombre de tâches actuellement sur le système et le quantum de temps à attribuer à une tâche.

[1] Andrew Tanenbaum

Avec ces deux paramètres l'ordonanceur sera capable de calculer le temps processeur à attribuer à chaque tâche en fonction de sa priorité.

Une amélioration importante devra cependant être apportée au scheduler de KoinKoin : calculer le temps processeur de chaque thread en fonction du nombre total de threads dans la tâche et du temps processeur de celle-ci. Et ce afin déviter qu'une tâche créant sans cesse de nouveaux threads ne monopolise le processeur, bloquant de fait l'exécution des autres tâches.

9.3 L'algorithme utilisé

Nous avons choisi d'utiliser un algorithme d'ordonnancement dit préemptif. Andrew Tanenbaum en donne la définition suivante : "Un algorithme d'ordonnancement préemptif sélectionne un processus et le laisse s'exécuter pendant un délai déterminé. Si le processus est toujours en cours à l'issue de ce délai, il est suspendu, et l'ordonnancement sélectionne un autre processus à exécuter (s'il y en a un de disponible). L'ordonnancement préemptif nécessite une interruption à la fin du délai afin de redonner le contrôle du processeur à l'ordonnanceur."

9.4 Structure de l'ordonnanceur

La structure de l'ordonnanceur (type **t_sched**) possède les champs suivants :

- **quantum** : Interval de temps pendant lequel un thread est autorisé à s'exécuter.
- **run** : Ensemble des threads en état d'exécution.
- **thrid** : Identifie le thread actuellement en cours d'exécution.

Chapitre 10

Mécanisme de communication : IPC

10.1 Définition d'un message

Dans KoinKoin OS, les communications inter-processus se font par le biais de messages qui sont, nous le rappelons, une évolution des appels systèmes traditionnels.

Un message est tout simplement un packet logiciel structuré par un header donnant certaines informations concernant le message ainsi que les données ayant un sens pour le destinaire du message.

10.2 Structure d'un message

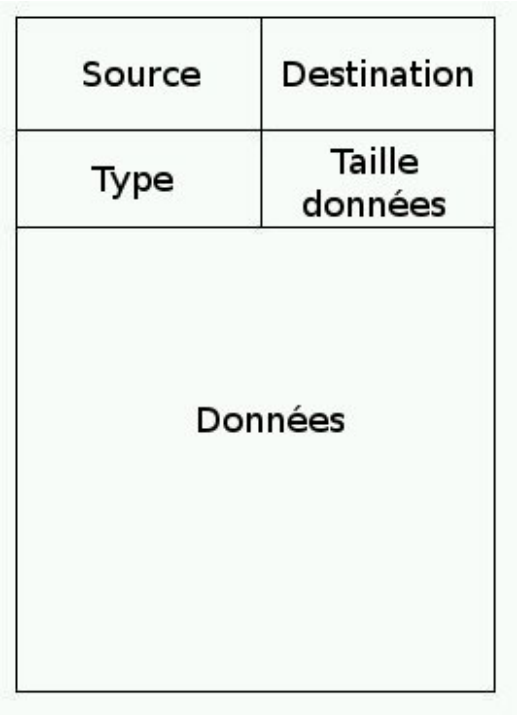
Avant d'aller plus loin, présentons la structure du header d'un message, qui ressemble très fortement à une trame UDP.

Nous trouvons bien évidement les deux champs correspondant à la source et la destination, représenté par des **t_tskid**.

Vient ensuite le type du message (unicast, multicast, ...).

Finalement le dernier champ correspond à la taille des données, ces données ayant une structure propre au serveur réceptionnant le message.

Voici une représentation d'un message en mémoire :



Structure d'un message

10.3 Création d'un message

La création d'un message s'effectue en utilisant la fonction de la libc système :

```
int create_msg(t_tskid dest, int data_size);
```

.

Il suffit à l'utilisateur de passer en paramètre le destinataire du message, ainsi que la taille des données à transporter.

Cette fonction est en fait un appel système, et la création du message est donc effectuée par le noyau. Ce dernier commence par réaliser une allocation mémoire (physique et virtuelle) de pages suffisante pour contenir le message en entier, c'est-à-dire le header et les données. Cette allocation, effectuée dans l'espace d'adressage du créateur du message, n'est pas effectuée en utilisant **malloc**, par mesure de sécurité. En effet, comme nous le verrons par la suite, la zone physique contenant le message sera référencé par l'émetteur et le destinataire. Il ne faut donc pas que le destinataire puisse modifier les structures de gestion du **malloc** de l'émetteur, présentes en début de page.

Il s'agit donc simplement d'une allocation de pages.

Le noyau écrit alors le header du message, en utilisant les paramètres fournis par l'émetteur (destination et taille des données). L'adresse des données est déterminée en ajoutant la taille du header à l'adresse de début du message. La source (c'est-à-dire l'identifiant de tâche de l'émetteur) est fournie par l'ordonnanceur, excluant la possibilité d'attaque par spoofing sur la source (usurpation d'identité de tâche).

Le noyau retourne enfin à l'émetteur une structure de message pointant sur la zone mémoire remplie. Le programme appelant peut alors utiliser la structure retournée et remplir les données.

10.4 Envoi d'un message

L'envoi d'un message s'effectue en utilisant la fonction de la libc système :

```
int send_msg(t_msg *msg_hdr);
```

.

Il suffit à l'utilisateur de passer en paramètre la structure de message précédemment crée et remplie.

Encore une fois, cette fonction est un appel système et c'est donc le noyau qui se charge d'envoyer le message. Ce dernier commence par lire dans le header du message l'identifiant de tâche du destinataire (en se plaçant dans l'espace d'adressage de l'emetteur). Puis il initialise et remplie une autre structure de message, dit "message noyau". Cette structure est composée des champs suivants :

- **src** : ce champs est rempli avec l'identifiant de tâche de l'émetteur.
- **msg** : ce champs est rempli avec l'adresse de la structure de message passée en paramètre par l'émetteur.

Il ne reste alors plus qu'à délivrer ce message à la tâche destinatrice. Tout d'abord un petit complément sur les tâches : chacune d'elles possède un ensemble de messages, en fait une liste chaînée dans laquelle sont stockées les messages à destination de la tâche. Le noyau ajoute donc simplement la structure message noyau créée dans la file de messages du destinataire et rend la main à la tâche émettrice.

10.5 Récupération d'un message

La récupération d'un message, en mode asynchrone, s'effectue en utilisant les fonctions libc suivantes :

- `t_msg *get_msg(void);`
  Permet de récupérer un message, quelque soit l'émetteur.

- `t_msg *get_msg_from(t_tskid src);`
  Permet de récupérer un message en provenance de la tâche **src** uniquement.

Là encore ces fonctions sont des appels systèmes, et la récupération des messages est donc effectuée par le noyau. Nous n'expliquerons ici que le traitement effectué lors de l'appel de **get_msg()**, puisque le traitement de **get_msg_from(t_tskid src)** ne diffère que par un test entre le champs **src** de la structure message noyau et le paramètre **src** fourni par la tâche appelante.

Le noyau commence par vérifier si un message est présent dans la file de messages de la tâche appelante. Si c'est le cas, il récupère l'adresse physique du message, en utilisant l'adresse virtuelle du message dans la tâche émettrice (identifiée par le champs **src** du message noyau). Cette adresse virtuelle est fournie, rappelons le, par le champs **msg** du message noyau.

Le noyau effectue alors une réservation d'espace mémoire virtuel dans l'espace d'adressage de la tâche appelante (c'est-à-dire le destinataire du message), suffisamment grande pour contenir le message en entier (header et données). Puis il référence la zone physique du message en utilisant l'adresse virtuelle réservée. Enfin, il change la valeur de l'adresse (virtuelle) des données dans le header, pour qu'elle corresponde avec l'adresse virtuelle réservée à laquelle est ajoutée la taille du header. Ainsi les données seront accessible par la tâche appelante.

Le noyau termine le traitement en renvoyant l'adresse virtuelle du message, et rend la main à la tâche appelante, qui peut alors lire le message.

10.6 Attente d'un message

Une tâche peut se mettre en attente d'un message, c'est-à-dire bloquer son exécution jusqu'à la réception d'un message (mode synchrone). Cette fonctionnalité est fournie par l'utilisation de deux fonction libc :

- `t_msg *wait_msg(void);`
  Met la tâche en attente d'un message, quelque soit l'émetteur.

- `t_msg *wait_msg_from(t_tskid src);`
  Met la tâche en attente d'un message en provenance de la tâche **src** uniquement.

Comme d'habitude, ces deux fonctions sont des appels systèmes, déléguant ainsi le traitement au noyau. De même, nous n'expliquerons que le traitement de **wait_msg()**. En effet, celle-ci utilise la fonction **get_msg()**, alors que **wait_msg_from(t_tskid src)** utilise **get_msg_from(t_tskid src)**. Il n'y a pas d'autres différences entre ces deux fonctions.

Le noyau commence par vérifier si un message est présent dans la file de messages de la tâche, par un appel à **get_msg()**. Si tel est le cas, le noyau retourne directement l'adresse du message et rend la main à la tâche appellante. Cette dernière continue donc son exécution, sans avoir été mise en pause.

Si aucun message n'était présent, le thread ayant effectué l'appel se voit affecté un statut d'ordonnancement particulier : STATUS_MSG. Puis le noyau demande à l'ordonnanceur de passer au thread suivant (bien sûr en réalisant la sauvegarde du contexte d'exécution et toutes les opérations usuelles). Le thread appelant est alors enlevé de l'ensemble des threads en cours d'exécution, et se retrouve donc en attente ou bloqué. Le noyau rend alors la main, et le prochain thread s'exécute.

Intéressons-nous pour finir au réveil du thread en attente. Cette opération s'effectue en fait au moment de l'envoi d'un message à cette destination. En effet, lors du traitement de **send_msg(t_msg *msg_hdr)**, comme nous l'avons dit précédemment, le noyau récupère l'identifiant de tâche du destinataire. Il vérifie alors le statut du destinataire, et si celui-ci se trouve en statut STATUS_MSG. Si c'est le cas, le thread est réveillé, c'est-à-dire remis en statut d'ordonnancement STATUS_RUN, et replacé dans l'ensemble des threads en éxécution (c'est-à-dire l'ensemble de l'ordonnanceur). A sa pro-

chaine exécution, c'est-à-dire quand l'ordonnanceur lui redonne la main, il récupère le message et poursuit son exécution.

Chapitre 11

Les appels systèmes

11.1 Définition

Les appels systèmes permettent aux programmes utilisateurs de demander des services au noyau, et donc d'interagir avec lui. Un appel système fait donc basculer le système en mode noyau (CPL0), puis, une fois que le noyau a effectué le traitement demandé, le système rebascule en mode utilisateur (CPL3) et le programme utilisateur reprend son exécution, juste après l'invocation de l'appel système.

11.2 Fonctionnement

Chaque appel système possède un numéro unique. L'appel système est déclenché par une interruption logicielle (instruction assembleur **int**). La plupart des appels systèmes possèdent des arguments, qu'il faut également transmettre au noyau. Pour cela, on les place dans des registres, et ce bien sûr avant de déclencher l'interruption.

Nous avons choisi comme convention que le code de l'appel système soit placé dans le registre EAX, et ses arguments dans les registres EBX, ECX et EDX (si nécessaire). Les appels systèmes sont codés en assembleur, seul langage permettant de changer la valeur d'un registre.

Les appels systèmes utilisés par les programmes fonctionnant en espace utilisateur sont présents dans la libc.

11.3 Les appels systèmes présents dans Koin-Koin

11.3.1 Les appels systèmes non-privilégiés

Ces appels systèmes sont librement utilisables par tous services ou programmes. Ils utilisent l'interruption 0x30 (48ième entrée de l'IDT).

Les appels systèmes suivant sont temporaires, ils seront bientôt implémenter sous forme de fonctions utilisateurs de la librairie C, avec une implémentation thread-safe :

Code syscall	Fonction libc	Description
SYSCALL_MALLOC	malloc(unsigned int size)	Alloue de la mémoire
SYSCALL_FREE	free(void **data)	Libère de la mémoire précédemment allouée par malloc

Les appels systèmes suivants sont totalement définitif, et concerne les messages :

Code syscall	Fonction libc	Description
SYSCALL_GET_MSG	get_msg()	Récupère le prochain message de la tâche
SYSCALL_WAIT_MSG	wait_msg()	Endort l'appelant jusqu'à réception d'un message
SYSCALL_WAIT_MSG _FROM	wait_msg_from(...)	
SYSCALL_GET_MSG _FROM	get_msg_from(...)	
SYSCALL_SEND_MSG	send_msg(t_msg *msg_hdr)	envoie le message msg_hdr
SYSCALL_CREATE_MSG	create_msg(t_tskid dest, int data_size)	Crée un message

Les appels systèmes suivants sont également définitif, mais leur implémentation changera (Cf. partie "Evolutions futures") :

– Appels systèmes portant sur les threads. Ils sont librement utilisables à condition que l'opération demandée porte sur un thread de la même tâche que le thread appelant. Seul Mod peut effectuer des opérations de thread sur toutes les tâches.

Code syscall	Fonction libc	Description
SYSCALL_THREAD_RSV	thread_rsv(...)	Création d'un nouveau fil d'exécution
SYSCALL_THREAD_ ATTACH	thread_attach(...)	Attache un thread à une tâche
SYSCALL_THREAD_STACK	thread_stack(...)	Réservation d'une pile pour un thread
SYSCALL_THREAD_LOAD	thread_load(...)	Charge le contexte d'exécution d'un thread
SYSCALL_THREAD_STORE	thread_store(...)	Récupère le contexte d'exécution d'un thread
SYSCALL_THREAD_CLONE	thread_clone(...)	Clone un thread
SYSCALL_THREAD_ARGS	thread_arg()	Place les arguments du programme sur la pile
SYSCALL_THREAD_RUN	thread_run(...)	Met un thread en état d'exécution
SYSCALL_THREAD_WAIT	thread_wait(...)	Met le thread en attente de la mort d'un thread
SYSCALL_THREAD_EXIT	thread_exit(...)	Termine l'exécution d'un thread

– Appels systèmes portant sur les tâches :

Code syscall	Fonction libc	Description
SYSCALL_TASK_WAIT	task_wait(...)	Met une tâche en attente de la mort d'une tâche
SYSCALL_TASK_EXIT	task_exit(...)	Termine l'exécution de la tâche

– Autres :

Code syscall	Fonction libc	Description
SYSCALL_EXIT	exit(int value)	Termine l'exécution de l'appelant
SYSCALL_SUBSCRIBE_ TRAP	subscribe_trap(...)	Inscription à une trap
SYSCALL_ UNSUBSCRIBE_TRAP	unsubscribe_trap(...)	Désinscription d'une trap
SYSCALL_TIME	system_time()	Retourne la date et l'heure en secondes

11.3.2 Les appels systèmes privilégiés

Ces appels systèmes sont utilisés et réservés au service Mod, et concernent la création de tâches, threads, ainsi que l'exécution. Ils utilisent l'interruption 0x31 (49ième entrée de l'IDT).

Code syscall	Nom fonction	Description
SYSCALL_VM_RSV	vm_rsv(...)	Allocation de mémoire virtuelle
SYSCALL_VM_MAP	vm_map(...)	Mappe un espace mémoire physique dans un espace d'adressage
SYSCALL_MM_REL	mm_rel(...)	Libère des pages physiques et virtuelles
SYSCALL_TASK_RSV	task_rsv(...)	Création d'une tâche
SYSCALL_TASK_RUN	task_run(...)	Mettre une tâche en exécution
SYSCALL_AS_RSV	as_rsv(...)	Réservation d'un espace d'addressage
SYSCALL_AS_ATTACH	as_attach(...)	Attache un espace d'adressage à une tâche
SYSCALL_AS_SET_MODID	as_set_modid()	Affecte un module à un espace d'adressage
SYSCALL_AS_GET_MODID	as_get_modid(...)	Récupère l'identifiant du module chargé dans un espace d'adressage

11.3.3 Evolutions des appels systèmes

Il est prévu de changer la façon dont sont implémentés la majorité des appels systèmes de KoinKoin. En effet à terme seul les appels systèmes concernant la gestion des messages subsiteront dans leur forme actuelle.

Les autres appels systèmes prendront en fait la forme d'un envoi de message à KoinKoin, les données duquel décriront l'appel système (code identifiant + arguments). Ainsi lors de l'envoi d'un message, KoinKoin vérifiera si le message lui est destiné. Si tel est le cas, il parsera le message et effectuera l'appel système pour le thread appelant.

Le premier avantage de cette approche concerne la portabilité du code. En effet, seuls les appels systèmes concernant la gestion des messages seront codés en assembleur, réduisant le nombre de lignes de code dépendant de l'architecture. Un autre avantage est la facilité d'écriture d'appels systèmes, en particulier le passage d'arguments.

Enfin cette approche nous permettrait de mettre en place une optimisation intéressante pour les performances, nommé "mode rafale". En effet un message pourrai être utilisé pour demander en un envoi la réalisation de plusieurs appels systèmes, ne nécessitant qu'un seul changement de contexte. La permission serai toujours vérifiée pour chaque appel système de la rafale, et une limite au nombre d'appels systèmes que peut contenir une rafale serai

imposée. Ainsi la rafale permettrai d'améliorer les performances lors de l'enchainement d'appels systèmes sans pour autant porter atteinte à la sécurité de KoinKoin.

Chapitre 12

Les pilotes de périphériques

12.1 Définition

Un pilote est un service particulier dans le sens ou celui-ci intéragit avec le matériel. Cela ne signifie pas qu'il a des droits accrus. En effet, généralement, le noyau, les pilotes et les services auront concrètement les mêmes droits systèmes.

Bien sûr tous les pilotes de périphériques ou drivers de KoinKoin sont donc implémentés sous la forme de petits services ou serveurs indépendants et cloisonnés fonctionnant en espace utilisateur, appartenant tous à la couche Drivers de KoinKoin OS.

12.2 Fonctionnement général des périphériques

12.2.1 Les ports d'entrée / sortie

La plupart des périphériques possèdent des registres permettant de contrôler leur fonctionnement. On accède à ces registres grâce à des ports d'entrée / sortie (ports E/S) qui leur sont associés afin de permettre de lire ou d'écrire dans ces registres. Ces ports sont identifiés par une adresse unique. Ainsi, nous parlerons de registres quand nous nous intéresserons au contenu de ces registres et nous parlerons de port E/S quand il s'agira d'envoyer (ou de recevoir) des données aux (ou des) périphériques.

Les périphériques disposant de ports ont un port de base unique. Les adresses des ports du périphérique sont calculées à partir de ce port de base.

Les ports E/S peuvent généralement être classés en 4 catégories :

- Les ports de contrôle permettent d'envoyer des commandes au périphérique ou de modifier son comportement
- Les ports de status permettent de connaître l'état du périphérique
- Les ports de données en lecture permettent de lire des données depuis le périphérique, typiquement le résultat d'une commande
- Les ports de données en écriture permettent d'écrire des données sur le périphérique

Pour envoyer (et recevoir) des informations sur (et depuis) ces ports, nous utilisons les fonctions C outb ou outw (et inb ou inw) que nous avons créé et qui encapsulent simplement les fonctions éponymes en assembleur.

Les fonctions outb et inb envoient ou reçoivent un octect tandis que les fonctions outw et inw envoient ou reçoivent des mots (un mot vaut deux octects).

Après avoir envoyé des informations sur un port (avec outb ou outw), il peut être nécessaire de vérifier si le périphérique est occupé, ce qui signifie qu'il est en train de prendre acte de notre information et de faire les changements ou les traitements nécessaires. Si c'est le cas, il est préférable d'attendre qu'il ne soit plus occupé avant de lui envoyer de nouvelles informations, que ce soit sur le même port ou un autre.

12.2.2 Interruptions matérielles

Comme nous l'avons vu dans le chapitre sur les interruptions, on peut associer à certains périphériques un numéro d'IRQ et une fonction "handler" qui sera appelée par le gestionnaire d'interruptions lorsqu'il recevra une interruption sur cette IRQ.

12.2.3 Mode caractère et mode bloc

La plupart des périphériques peuvent être classés en deux catégories : les périphériques en mode caractère et les périphériques en mode bloc.

Les périphériques en mode caractère écrivent ou lisent des caractères un par un. Les périphériques en mode bloc, au contraire, ne peuvent écrire ou lire que des blocs, c'est à dire 512 octects (256 mots), ou un multiple de 512 octects, à la fois.

12.3 Pilote de la CMOS

12.3.1 Définitions

La CMOS, pour Complementary Metal Oxide Semiconductor, prend la forme d'un petit composant mémoire sur la carte mère, alimenté en permanence par une petite pile. De fait les informations qu'elle contient sont conservées lors d'un redémarrage de l'ordinateur.

La CMOS est utilisée comme mémoire permanente du BIOS, lui servant "notamment à garder en mémoire des informations dont il a besoin lors du boot"[1]. Citons pour exemple le nombre et type de lecteurs disquettes, des informations sur les disques durs, sur la date et l'heure, ... Chaque information est identifiée par la CMOS par son offset de décalage par rapport à l'adresse de début.

12.3.2 Intéragir avec la CMOS

La communication avec la CMOS s'effectue au moyen d'I/O ports. En effet la CMOS a deux ports associés :

1. Le port de requêtes (port 0x70) : ce port permet de demander une requête à la CMOS. On y écrit l'information que l'on souhaite lire ou écrire (représentée par son offset), et la CMOS dépose alors l'information dans le deuxième port.
2. Le port d'action (port 0x71) : ce port permet de réaliser une opération en lecture ou écriture sur l'information demandée précédemment (soit IN soit OUT).

12.3.3 Implémentation du pilote

Sous KoinKoin, le pilote de la CMOS fait partie intégrante du noyau. Il constiste en une fonction de lecture d'informations CMOS, prenant en paramètre l'information recherchée et en effectuant un OUT sur le port de requêtes, puis retourne la valeur lue sur le port d'action. Une liste de DEFINE des informations CMOS est présente dans le header du pilote[2].

Un pilote CMOS implémenté sous la forme d'un service utilisateur a récemment été ajouté à KoinKoin OS. Ce pilote peut être utilisé par les autres processus utilisateurs pour obtenir des informations CMOS. L'interface exportée par ce pilote consiste en la fonction de librairie C suivante :

[1] Jean-Pascal Billau, LSE

[2] src/kaneton/drivers/cmos/cmos.h

```
int cmos_information(unsigned char information,
                     unsigned char *value);
```

Avec *information* identifiant l'information CMOS demandée, une liste de define étant fourni dans le fichier en-tête *include/drivers/cmos.h*. La valeur de l'information est retournée dans le paramètre résultat *value*. Cette fonction renvoie 0 en cas de succès, 1 si une erreur a eu lieu (mauvais code information, ...).

12.4 Pilote de l'horloge

12.4.1 Fonctionnement

L'horloge ou timer tient un rôle crucial dans le système d'exploitation. En effet, l'horloge permet de connaître l'heure courante, mais également de savoir à quel moment l'ordonnanceur doit passer d'un thread à un autre. Compte tenu du rôle de ce pilote, il réside dans le micro-noyau.

L'horloge est liée à une interruption, la numéro 0x0, et à un composant matériel, qui fournit une impulsion à une fréquence donnée, ou tic. Cette impulsion prend alors la forme d'une interruption 0x0, qui appelle donc le handler mis en place à l'entrée 0 de l'IDT. Ce dernier comtabilise les tics, et effectue alors le traitement associé (passer au thread suivant, mettre l'heure à jour, ...).

//FIXME LES PORTS TYPE MACHIN

12.4.2 Gestion de l'heure

Après informer l'ordonnanceur d'un basculement de thread, l'horloge met également à disposition du système l'heure et la date actuelle. Cette information est stockée par le pilote de l'horloge sous la forme d'une structure **t_clock**, composée des champs suivants :

1. initial_time : Date et heure initiale, lors du démarrage du système, exprimée en secondes.
2. elapsed_time : Temps écoulé depuis le démarrage du système, en secondes.

Lors de son initialisation, le pilote de l'horloge initialise une variable **t_clock**, la **system_clock**. Pour cela, l'horloge demande au pilote du CMOS la date et l'heure actuelle, la transforme en secondes, et l'affecte au champ "initial_time" de la **system_clock**. Le pilote affecte également la valeur 0 au champ "elapsed_time" de la **system_clock**.

La mise à jour de ce champ se passe de la manière suivante : l'horloge sait parfaitement que le quantum de temps alloué à chaque thread est de "NB_TICK", soit 10ms. L'horloge utilise donc un compteur temporaire, celui de la seconde en cours, qu'elle incrémente de 10 à chaque changement de thread. Ainsi lorsque le compteur arrive à 60, elle incrémente le champ "elapsed_time" de la **system_clock** et remet le compteur à 0.

Lorsque l'on souhaite connaître l'heure actuelle, l'horloge renvoie la somme des deux champs de la **system_clock**. On obtient alors l'heure et la date actuelle du système exprimée en secondes. On utilise pour cela l'appel système exporté par l'horloge :

```
unsigned int system_time(void);
```

12.5 Pilote du clavier

Dans cette section, nous présenterons le coeur du fonctionnement du pilote de clavier, qui est un périphérique en mode caractère.

12.5.1 Fonctionnement matériel

Nous allons commencer par décrire les interactions et évènements matériels lors de la saisie d'une touche au clavier. A chaque fois que l'on appuie sur une touche du clavier, ce dernier génère un code identifiant la touche, appelé scan code. Il y a deux types de scan code :

- make code : indique un enfoncement de touche. Le bit 7 est à la valeur 0. Par conséquent tous les make codes ont une valeur inférieure à 128.
- break code : indique un relâchement de touche. Ici le bit 7 vaut 1, et les break codes sont donc supérieurs ou égaux à 128.

Certaines touches sont composées de plusieurs scan codes. On parle de touches étendues, et sont prefixées du scan code 0xE0 ou 0xE1. C'est le cas des touches du pavé numérique, ou encore de PageUp et PageDown.

Pour communiquer tous ces scan codes, le clavier déclenche une interruption 0x1 (IRQ1). Le driver est alors prévenu qu'un scan code va être transmis, via les I/O Ports du clavier (ports 0x60 et 0x64).

Le port 0x64, appelé registre d'état, convient d'être lu en premier. C'est en effet lui qui dit si le scan code est disponible dans le tampon de sortie (port 0x60). Il suffit pour cela de tester la valeur du bit 1 du registre d'état. Si il est activé, le pilote peut récupérer la valeur du scan code sur le port 0x60.

12.5.2 Conversion des scan codes en ASCII

Les scan codes n'ont aucune signification en code ASCII. Ce sont des codes purement liés au matériel. Il est donc nécessaire de les transcrire en caractère ascii avant de les transmettre à la tâche demandante. On utilise pour cela une table de map, qui fait correspondre le caractère ascii et le scan code. Cette table permet également la résolution de certaines combinaisons de caractères, comme Shift + a = 'A'.

Il est nécessaire de mettre en place une table de map par langue ; en effet, la signification des scan codes du clavier américain et français diffèrent.

12.5.3 Implémentation dans KoinKoin

Un petit rappel d'importance avant de continuer : dans KoinKoin le driver clavier, comme la quasi totalité des drivers, est externalisé du noyau. Il d'agit d'une tâche autonome, s'écutant en espace utilisateur, et communiquant par messages (c'est-à-dire un service).

Ainsi le pilote du clavier ne reçoit pas directement les interruptions 0x1. Elles lui sont en effet relayées par le handler associé à l'entrée 0x1 de l'IDT, sous la forme d'un message.

A réception de ce message, le pilote va alors lire le ou les scan codes composant la touche, selon qu'il s'agisse d'une touche étendue ou non (fonction **int Irq1_FillKeyboardBuffer(void)**). Avant de lire le scan code sur le port 0x60, on vérifie bien que la donnée est disponible en testant la valeur du registre de contrôle. L'intégralité des scan codes composant la touche est récupéré et stockée dans le tampon : **gl_key_buf**. Ne sont bufferisés que les make codes, le traitement associé à la touche ayant déjà été effectué lorsque survient le break code.

L'étape suivante consiste, à partir de l'ensemble des scan codes de la touche, de déterminer le code ascii associé, si il y en a. Le pilote commence par "poper" le buffer, afin de récupérer dans la variable globale **gl_kbd** l'ensemble des scan codes composant la touche. Cette structure possède également la valeur ascii de la touche si elle existe, ainsi qu'un ensemble de variables précisant si une touche spéciale est activée (c'est-à-dire enfoncée).

Les touches spéciales ainsi mémorisées sont : Shift, Ctrl, Windows, Alt et Altgr. Lors de la réception du make code d'une de ces touches, on affecte 1 à la variable, 0 lors de la réception de son make code.

Lorsqu'une touche possède une équivalence ascii, on utilise la table de correspondance. Le pilote sait quelles touches spéciales sont activées, et il n'y a donc pas de problème pour déterminer le code ascii (par exemple 'a' et 'A'.).

De plus certaines combinaisons de touches sont prises en compte, telles que :

- Shift + PageUp : Scrolling de 10 lignes vers le haut dans le tty.
- Shift + PageDown : Scrolling de 10 lignes vers le bas dans le tty.
- Alt + Fnum : Changement de tty.

12.6 Le pilote de tty

Le driver TTY est le gestionnaire d'affichage, ou console. Il met à la disposition du système un certain nombre de ttys, nombre défini par MAX_TTY.

Chaque tty possède son propre buffer, servant à stocker l'affichage.

Les fonctionnalités offertes par ce driver sont les suivantes :

- Affichage d'un caractère ou d'une chaîne de caractères.
- Affichage en couleur
- Le scrolling
- L'affichage du backspace, en d'autres termes l'effacement d'un caractère.
- L'effacement complet de la console.
- La gestion du curseur, celui-ci étant géré via I/O ports.
- Le changement d'affichage de tty (passage d'un tty à un autre).

12.7 Pilote de disque dur ATA

Dans cette section, nous présenterons le coeur du fonctionnement du pilote de disque dur ATA.

ATA (Advanced Technology Attachment), plus connu sous le nom commercial de IDE (Integrated Drive Electronics), est le plus répandu des standards de connexion de disque durs à un ordinateur.

Nous avons donc programmé un pilote capable d'interfacer les disques durs suivants le standard ATA.

Il existe aussi une extension à ce standard qui est le standard ATAPI (ATA with Packet Interface). Celle-ci permet quant à elle d'interfacer d'autres périphériques de stockage tels que les lecteurs de CDROM. Mais nous ne parlerons pas plus de ce standard car nous n'avons pas encore implémenté de pilote pour ces périphériques.

12.7.1 Fonctionnement général des disques durs

Afin de comprendre comment fonctionne le pilote de disque dur, il est tout d'abord nécéssaire d'expliquer les bases du fonctionnement matériel des disques durs. [3]

Les périphériques de stockage comme les disques durs ou les lecteurs de CDROM sont branchés sur des nappes, qui les relient à des contrôleurs électroniques. Sur chacune de ces nappes, il est possible de brancher deux périphériques de stockage, l'un étant appelé le maître et l'autre l'esclave. Ces périphériques de stockage peuvent donc être des disques durs, mais aussi d'autres types de périphériques de stockage. Nous utiliserons le terme générique de "lecteur" pour désigner un de ces périphériques.

Un disque dur est en réalité constitué de plusieurs disques empilés, que l'on appelle des plateaux.

Un cylindre représente une zone verticale sur l'ensemble des plateaux.

Des têtes de lecture sont situées de part et d'autre des plateaux. Ce sont elles qui réalisent les opérations de lecture et d'écriture sur les disques. Une seule de ces têtes est active à la fois.

Chacun des plateaux est divisé (virtuellement) en cercles concentriques appelés pistes. Ces pistes sont elles-mêmes divisées en secteurs.

12.7.2 Les ports d'entrée / sortie

Les disques durs eux-mêmes n'ont pas de ports E/S. Ce sont les contrôleurs électroniques qui en disposent. L'adresse des ports de base des contrôleurs sont 0x1f0 pour le premier et 0x170 pour le deuxième.

[3] Pour comprendre tous les détails du fonctionnement des disques durs, voir http ://www.commentcamarche.net/pc/disque.php3

A partir de ces ports de base, il faut ajouter un des codes suivants afin d'obtenir l'adresse du port souhaité.

Nous allons maintenant décrire les principaux ports des disques durs que nous avons utilisé.[4]

- Lecteur et mode d'adressage (0x06) : Une partie du registre associé à ce port est utilisée pour spécifier le lecteur utilisé, c'est à dire le maître ou l'esclave du contrôleur. Il permet également de spécifier le mode d'adressage (CHS ou LBA).
- Contrôle (0x206) : Ce port permet d'initialiser le contrôleur, d'activer ou désactiver les interruptions, et de préciser si le lecteur utilisé dispose de plus ou moins de 8 têtes de lecture.
- Commande (0x07) : Ce port permet, en écriture, d'envoyer des commandes au disque dur.
- État (0x07) : Ce port permet, en lecture, de connaître l'état actuel du disque dur (occupé, en attente de données, ...).
- Erreur (0x01) : Ce port permet, en lecture, de connaître, le cas échéant, le code d'erreur de la dernière commande executée.
- Adressage (0x03, 0x04, 0x05, 0x06) : Lorsqu'on utilise le mode CHS, ces ports permettent de spécifier le cylindre, la tête de lecture et le secteur de début. Lorsqu'on utilise le mode LBA, une partie de l'adresse est envoyée sur chacun de ces ports.
- Nombre de secteurs (0x02) : Ce port permet de spécifier le nombre de secteurs concernés par une opération de lecture ou d'écriture.
- Données (0x00) : Ce port permet d'envoyer ou de recevoir des données du disque dur.

12.7.3 Interruptions matérielles

Les contrôleurs IDE peuvent recevoir des interruptions matérielles, sur l'IRQ 14 pour le premier contrôleur et sur l'IRQ 15 pour le deuxième. Une telle interruption est émise à chaque fois qu'un disque dur a terminé la lecture ou l'écriture d'un secteur. Nous n'effectuons aucun traitement particulier à la réception de ces interruptions.

12.7.4 Modes d'adressage

Il existe 2 modes d'adressage possibles pour les disques durs.

[4]Pour une description complète des tous les ports des disques durs, voir http ://fr.wikipedia.org/wiki/Integrated_drive_electronics

Le plus répandu est le mode CHS, acronyme de Cylinder/Head/Sector ou en français Cylindre/Tête/Secteur. Pour l'utiliser, il faut donc préciser sur quel cylindre et quel secteur on veut lire ou écrire et avec quelle tête de lecture. Ce mode est donc très proche dans sa conception de la représentation physique des disques durs. C'est pourquoi il est nécessaire de bien comprendre le fonctionnement des disques durs pour utiliser ce mode d'adressage de façon optimale. La principale limite de ce mode est qu'il ne peut adresser que 8 Go sur le disque dur. Pour les disques de plus grande taille, il était donc nécessaire d'inventer un nouveau mode d'adressage pour pouvoir utiliser la totalité du disque.

C'est ainsi que naquit le mode LBA (Logical Block Addressing). Ce mode se distingue tout d'abord par un adressage linéaire, c'est à dire que les adresses sont spécifiées simplement par un nombre entre 0 et la taille totale du disque dur. Contrairement au mode CHS, le mode LBA fournit donc une abstraction par rapport à la réalité matérielle d'un disque dur. Il permet d'adresser jusqu'à 128 Go. Ce mode d'adressage étant plus récent, il n'est pas supporté par tous les disques durs, bien que la plupart des disques récents le supportent.

12.7.5 Informations d'état

Le registre d'état

La lecture du registre d'état d'un contrôleur permet de connaître l'état actuel du lecteur actif. Chaque bit de ce registre donne une information différente sur l'état du lecteur.

Voici la signification de chacun de ces bits :
- bit 0 : Ce bit vaut 1 seulement si une erreur a eu lieu lors de la commande précédente.
- bit 1 : Ce bit n'est pas utilisé sur les lecteurs récents.
- bit 2 : Ce bit vaut 1 si le disque dur a corrigé lui-même des données lors de la commande précédente.
- bit 3 : Ce bit vaut 1 si le contrôleur est en attente de données, soit de données qu'on doit lui envoyer sur son port de données, soit des données que l'on doit lire à partir de son port de données. Voir la section suivante : "Échanges de données avec un disque dur" pour un exemple d'utilisation de cette information.
- bit 4 : Ce bit vaut 0 si le lecteur est en train d'effectuer une recherche, c'est à dire de positionner ses têtes de lecture à l'endroit où on lui a

demandé par l'intermédiaire des registres du contrôleur. Ce bit vaut 1 lorsqu'il a terminé ses recherches et que les têtes de lecture sont à la position demandée.
- bit 5 : Ce bit vaut 1 lorqu'il y a une erreur d'écriture sur le lecteur.
- bit 6 : Ce bit vaut 1 lorque le le lecteur est prêt à recevoir des commandes.
- bit 7 : Ce bit vaut 1 lorque le contrôleur est occupé, généralement en train d'exécuter une commande. Il vaut 0 lorsque le contrôleur n'est plus occupé et est donc disponible pour une autre utilisation.

Le registre d'erreurs

La lecture du registre d'erreur d'un contrôleur permet de connaître l'erreur rencontrée lors de l'exécution de la dernière commande. Ce registre vaut 0 si aucune erreur n'est survenue. Si le bit 0 du registre d'état vaut 1, cela signifie qu'il y a une erreur et il faut alors lire le registre d'erreur pour connaître le type d'erreur survenue. Chaque bit de ce registre désigne un type d'erreur différent.

Voici la signification de chacun de ces bits :
- bit 0 : Ce bit vaut 1 si le lecteur a trouvé le secteur demandé mais qu'il ne peut pas y accéder. Cela signifie généralement que ce secteur est endommagé.
- bit 1 : Ce bit vaut 1 s'il y a eu une erreur de recalibrage.
- bit 2 : Ce bit vaut 1 si la commande précédente a été interrompue, pour quelque raison que ce soit.
- bit 3 : Ce bit ne concerne pas les disques dur. Il est utilisé seulement pour les lecteurs amovibles. Il vaut 1 si le contrôleur a reçu une demande de modification pour le lecteur amovible concerné
- bit 4 : Ce bit vaut 1 si le lecteur n'a pas trouvé l'adresse ou le secteur demandé.
- bit 5 : Ce bit ne concerne pas les disques dur. Il est utilisé seulement pour les lecteurs amovibles. Il vaut 1 si le support a été changé.
- bit 6 : Ce bit vaut 1 si le lecteur a rencontré des données corrompues qu'il ne peut pas corriger.
- bit 7 : Ce bit vaut 1 si le lecteur a détecté un secteur corrompu.

12.7.6 Échanges de données avec un disque dur

Certaines commandes, typiquement les commandes d'identification, de lecture et d'écriture que nous verrons ensuite, ont besoin d'envoyer (ou de

recevoir) des données du disque dur. Quand le disque dur est prêt à envoyer (ou recevoir) des données, le bit d'attente de données (bit 3) de son registre d'état vaut 1. On peut alors envoyer (ou recevoir) des données vers (ou de) le port de données du contrôleur du disque concerné.

Les disques durs sont des périphériques en mode bloc. Ces échanges ne peuvent donc se faire que par tranche de 256 mots. Lorsque l'envoi ou la réception de la quantité de données définie par la commande en cours (qui est toujours un multiple de 256 mots) est terminée, le disque dur remet le bit d'attente de données de son registre d'état à 0. Si on n'envoyait (ou recevait) pas autant de données que prévu, le bit d'attente de données du registre d'état du disque dur ne serait pas remis à 0 et il resterait alors dans un état d'attente. C'est pourquoi lorsque nous souhaitons écrire moins de 256 mots, on doit compléter l'écriture par l'envoi de données nulles (on écrit des 0), et lorsque nous souhaitons lire moins de 256 mots, on continue la lecture jusqu'à 256 mots sans stocker les données qui ne nous intéressent pas.

12.7.7 Commandes

Mis à part l'initialisation, toutes les autres commandes commencent par l'envoi d'un code identifiant la commande sur le port de commandes.

Initialisation

L'initialisation d'un contrôleur de disques durs n'est pas vraiment nécessaire. En effet, il est tout à fait possible de lire et d'écrire sur ses disques durs, ainsi que d'effectuer d'autres opérations avec ceux-ci, sans avoir initialisé le contrôleur au préalable. La seule opération qui ne fonctionne pas sans l'initialisation est la détection du type de périphérique branché sur le contrôleur, à savoir ATA ou ATAPI.

Pour cette raison et parce que cela semble plus propre de toutes façon, nous initialisons toujours les contrôleurs de disques durs.

Pour initialiser un contrôleur de disque dur, il faut utiliser le registre de contrôle. Les différents bits de ce registre permettent de :

- Activer ou désactiver les interruptions matérielles du contrôleur (bit 1)
- Initialiser les périphériphériques branchés sur le contrôleur (bit 2)
- Définir si les lecteurs branchés sur le contrôleur ont plus ou moins de 8 têtes de lecture (bit 3)

La séquence d'initialisation est la suivante :

1. Désactiver les interrutions et positionner le bit d'initialisation pendant au moins 4.8 microsecondes (bit 1 et bit 2)
2. Enlever le bit d'initialisation (reste donc seulement le bit 1)
3. Dire au contôleur que notre lecteur a plus de 8 têtes de lecture (c'est le cas de la plupart des lecteurs récents) et réactiver les interruptions matérielles du contrôleur (bit 3 uniquement)

Obtention d'informations sur un disque

Pour obtenir des informations sur un disque, il faut suivre les étapes suivantes :

1. Envoyer le code de la commande de demande d'informations (0xec pour les disques dur) sur le port de commandes.
2. Vérifier que le bit d'attente de données vaut bien 1.
3. Lire 256 mots depuis le port de données.

À partir de ces données que nous avons obtenu, nous sommes capables de retrouver toutes les informations souhaitées. Ces informations incluent par exemple le nombre de cylindres, de têtes et de secteurs du lecteur, la marque du constructeur et encore bien d'autres informations.

Lecture

Pour lire des informations sur un disque, il faut suivre les étapes suivantes :

1. Envoyer sur le port de lecteur et de mode adressage le lecteur que l'on souhaite utiliser (maître ou esclave) ainsi que le mode d'adressage (CHS ou LBA)
2. Envoyer sur les ports d'adressage toutes les informations permettant au disque de déterminer où il devra lire sur le disque
3. Envoyer sur le port de nombre de secteurs, le nombre de secteur que l'on souhaite lire
4. Envoyer le code de la commande de lecture (0x20) sur le port de commandes.
5. Vérifier que le bit d'attente de données vaut bien 1.
6. Lire autant de fois 256 mots depuis le port de données que le nombre de secteurs que l'on souhaite lire.

Écriture

Pour écrire des informations sur un disque, il faut suivre les étapes suivantes :

1. Envoyer sur le port de lecteur et de mode adressage le lecteur que l'on souhaite utiliser (maître ou esclave) ainsi que le mode d'adressage (CHS ou LBA)
2. Envoyer sur les ports d'adressage toutes les informations permettant au disque de déterminer où il devra écrire sur le disque
3. Envoyer sur le port de nombre de secteurs, le nombre de secteur que l'on souhaite écrire
4. Envoyer le code de la commande d'écriture (0x30) sur le port de commandes.
5. Vérifier que le bit d'attente de données vaut bien 1.
6. Écrire autant de fois 256 mots depuis le port de données que le nombre de secteurs que l'on souhaite écrire.

12.7.8 Interface exportée

De même que pour les autres services, le driver IDE, après avoir effectué l'initialisation des contrôleurs IDE, se met en attente de messages. Des messages pourront ainsi lui être envoyés par les services de même niveau ou les programmes utilisateurs.

Le driver IDE fournit actuellement deux services :
- L'écriture sur le disque dur, grâce à la fonction ide_write de la libc.
- La lecture depuis le disque dur, grâce à la fonction ide_read de la libc. Dans ce cas, le service IDE renvoie le résultat de la lecture en envoyant un message de réponse au programme qui lui a demandé d'effectuer cette opération. Celui-ci, après avoir appelé la fonction ide_read devra donc se mettre en attente de message à son tour pour recevoir la réponse du service IDE.

Chapitre 13

Les services

13.1 Définition

Un service est une tâche qui comme son nom l'indique fournit un service. En d'autres termes, ce type de tâche est appellé par les tâches du même niveau ou du niveau inférieur pour effectuer une opération spécifique nommée service.

13.2 Fonctionnement

Dans KoinKoin OS, tous les services fonctionnent de façon similaire, et il est par conséquent très simple d'en créer un. En effet les étapes sont les suivantes :

1. chaque fonction **main** d'un service commence par initialiser les structures propres au service, comme décrit par la suite. KoinKoin met désormais à disposition des tâches utilisateur un gestionnaire d'ensembles ou **set**, particulièrement adapté aux structures de gestion des services. Les ensembles devraient être fortement appréciés par les développeurs de services.
2. Puis le service se met en attente de requêtes en provenance de tâches du même niveau ou inférieur, par un appel à la fonction **wait_msg** de la libc.
3. Une fois le message reçu par le service, il est parsé par une routine du service, chaque service pouvant utiliser la syntaxe de son choix pour les données du message. Généralement cela se traduit par un **switch** sur le premier octet du message, contenant le code identifiant la requête.

4. La requête étant identifiée, le service effectue le traitement associé, ou commande, en lisant les arguments dans le message si nécessaire. Liberté est laissée au développeur du service de déléguer le traitement à un autre thread, obtenu par un appel à la fonction de la librairie C utilisateur (niveau 3) :

   ```
   int launch_local_thread(void *function, t_thrid *thrid);
   ```

   Cette fonction est mise à disposition par la librairie C utilisateur spécifique de KoinKoin, fournit par le header **koinstd.h**. Il s'agit d'une abstaction des appels systèmes, simplifiant la création d'un thread local. Elle se charge en effet de la création du thread, de son attachement à la tâche locale, de la réservation de la pile (taille par défaut : 1 page, soit 4Ko) et de l'initialisation de son contexte d'exécution, avec comme point d'entrée la routine **function** passée en paramètre.

   Ainsi il suffira au développeur d'un service de créer une fonction par fonctionnalités du service, qui sera appelée par un thread dédié à réception de la requête correspondante. Le message reçu sera passé en paramètre du thread, le passage d'argument au thread sera en effet bientôt ajouté à **launch_local_thread()**.

Petit rappel : n'oubliez pas, dans KoinKoin OS les drivers sont avant tout des services, donc ce modèle de développement s'applique également à eux.

13.3 Les services présents dans KoinKoin

13.3.1 Le service Mod

Le service Mod est le gestionnaire de modules, un module étant une entité exécutable chargée en mémoire principale.

Le gestionnaire de modules fût introduit pour rendre le noyau encore plus simple et le système plus modulaire. En effet c'est le gestionnaire de modules qui s'occupent de charger un exécutable dans un espace d'adressage. De ce fait, toutes les créations de tâches font forcément appel au gestionnaire de modules.

Le service Mod permet une optimisation intéressante, puisqu'il permet d'enlever régulièrement de la mémoire principale les modules qui n'ont pas été utilisés depuis un certain temps et ayant une certaine durée de vie.

Un autre avantage de ce service est que le noyau ne manipulera désormais jamais de chaînes de caractères ; ce qui est extrêmement intéressant. Toute

la gestion des chemins étant faite dans les systèmes de fichiers et la relation entre les deux étant fournie par le service Mod.

Une autre conséquence de ce service est le fait que étant donné que seul ce service connaîtra la position des modules en mémoire, il est le seul à pouvoir lancer les services fondamentaux du système, jouant un rôle semblable au processus **Init** sous UNIX.

La seule contrainte étant que le noyau doive lancer manuellement la tâche correspondant au service Mod et ensuite passer à ce service la structure **multiboot_info_t** pour que ce service puisse en extraire la position des modules mais surtout le fichier de configuration nécessaire au lancement des services fondamentaux.

Démarrage

Comme dit précédemment, le service **Mod** est lancé manuellement par le noyau, qui lui transmet la structure Grub. Le service Mod démarre alors et commence par initialiser ses structures de gestion des modules (appel à la fonction **set_init**), avant de démarrer les services fondamentaux.

Pour cela, actuellement, **Mod** parcoure la structure Grub, récupère les informations de chaque module, crée les structures nécessaires à leur exécution, et démarre ces services. Mais il est prévu, à terme (en fait une fois que le système de fichiers sera présent), d'utiliser le fichier **/service/modules**.

Ainsi c'est ce fichier de configuration qui indiquera au gestionnaire de modules le nom du module, sa classe, sa priorité par défaut, son comportement, son nombre de pages de pile mais également les arguments à lui passer au lancement. De fait, **Mod** utilisera la structure Grub dans le seul but de récupérer l'adresse du fichier **/service/modules**.

Syntaxe des messages et commandes

Les fonctions permettant d'effectuer une requête auprès du service **Mod** sont présentes dans la libc, dans le fichier mod.c.

Lors du développement de ces fonctions, la syntaxe suivante a été mise en place :

1. Le premier octet des données contient le code de la commande.
2. Les octets suivants contiennent les paramètres de la commande.

Les commandes existantes au sein du service **Mod** sont les suivantes :

- MOD_CMD_LSMOD : cette commande ne prend aucun paramètre et liste les modules chargés en mémoire principale ; cette liste est affichée sur le TTY courant.
  Fonction libc :

```
int lsmod(int arg);
```

– MOD_CMD_GET_MODID : cette commande prend en paramètre un nom de module, et renvoit son identifiant.
Fonction libc :
```
t_modid get_modid(char *module_name);
```

– MOD_CMD_GET_TSKID : cette commande prend en paramètre le nom d'un service et renvoit son identifiant de tâche.
Fonction libc :
```
t_tskid get_tskid(char *module_name);
```

– MOD_CMD_EXEC : cette commande prend en paramètre un nom de module et l'exécute.
Fonction libc :
```
int exec(char *module_name, char **argv, t_thrid *thrid);
```

Chapitre 14

Les programmes utilisateurs

Dans cette partie nous présenterons les programmes utilisateurs de KoinKoin OS. Comprendre par là les programmes appartenant à la couche **User** du système.

14.1 Le Shell

Un Shell a récemment été ajouté à KoinKoin, le koinkoin Shell. Il s'agit en fait d'un minishell[1], extrêmement simple.

Les fonctionnalités présentes sont les suivantes :

- Un scanner élégant et évolutif est présent dans ce shell, basé sur une liste de tokens. L'écriture d'un parseur utilisant un AST (Abstract Syntax Tree) est prévu.
- La builtin **echo**. Cette builtin permet l'affichage d'une ligne de texte. L'option "-n" a été mise en place, et permet de ne pas afficher une nouvelle ligne après l'affichage du texte.
- La builtin **lsmod**. Cette builtin affiche une liste des modules chargés en mémoire, ainsi que leur identifiant. Elle possède les options suivantes :
  - -s : Affiche la taille du module. La taille affichée n'est pas un modèle de précison, puisqu'elle est calculée en se basant sur le nombre de pages physiques occupées par le module (donc la taille est fourni à 4 Ko près).
  - -t : Affiche l'identifiant de tâche attachée au module. Seule la première tâche associée au module est affichée.
- La builtin **clear**, qui efface totalement l'écran.
- La builtin **exit**, qui termine l'exécution du shell.

[1] Projet 10.5sh d'Ing1

- La builtin **date**, qui affiche la date et l'heure système. Cette builtin a juste pour but de tester l'horloge système, et deviendra à terme un binaire indépendant.
- Un prompt en couleur.
- La gestion du caractère ", c'est-à-dire l'effacement de caractères.
- La gestion de l'exécution, grâce à la fonction libc **exec(char *mod_name)**.

Bien sûr il est prévu d'étendre ces fonctionnalités par la suite, voire de porter un jour le 42sh, une fois que la libc aura été étendue. En attendant ce shell continue d'être amélioré afin de faciliter le test des fonctionnalités ajoutées. De plus il commence à utiliser des fonctions spécifiques à KoinKoin, telle que **task_wait()**, et reflétera donc parfaitement le fonctionnement de KoinKoin OS. En bref un code en C/KoinKoin.

14.2 Des programmes de test

Les répertoires *src/user/hello* et *src/user/test* contiennent le code des programmes de test :

- **hello** est utilisé pour le développement du CRT (C-Run Time) de Koin-Koin.
- **test** est utilisé pour tous les autres tests, tels que ceux concernant les drivers, services, multi-threading, ...

14.3 Banner

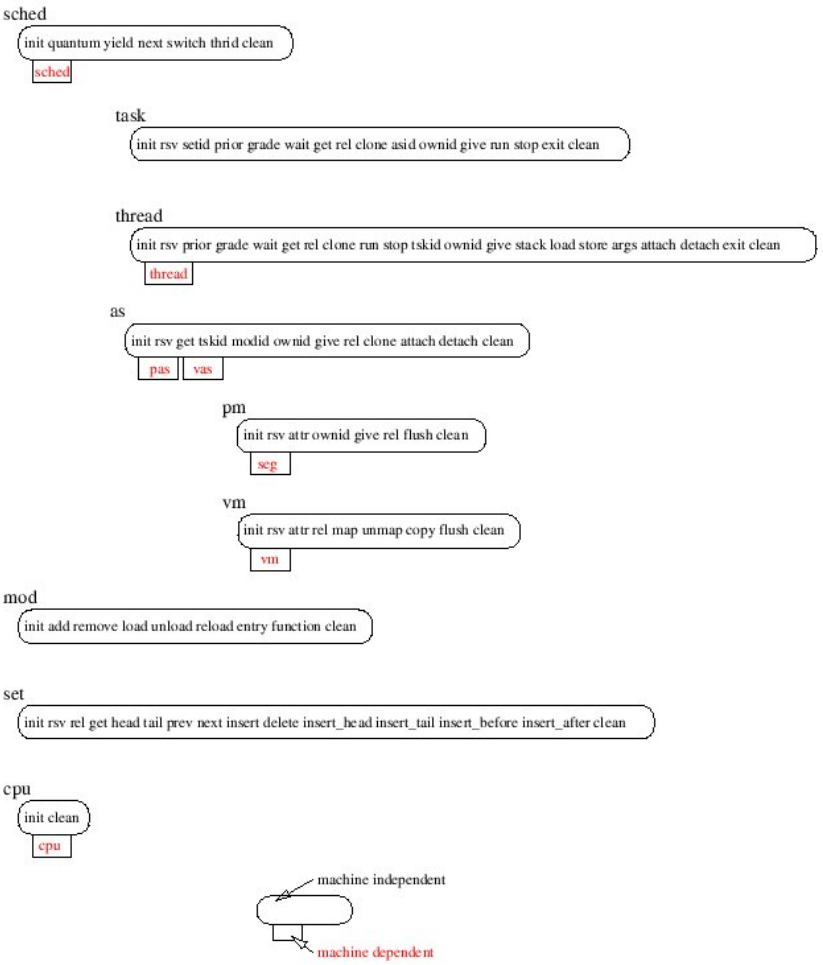
Le code source de Banner est présent dans le répertoire *src/user/banner*. Actuellement, ce programme n'est pas utilisable, il nous manque en effet la fonction **realloc**. Nous préférons d'abord externaliser le **malloc** actuel avant de coder **realloc**.

A noter que Banner est en fait le premier port d'un programme Linux sur KoinKoin OS, le code source étant celui utilisé par Gentoo Linux.

Chapitre 15

Visualisation du système

Voici une vue système au niveau des interfaces :



Chapitre 16

Évolutions futures

Grâce à l'implémentation incrémentale de briques solides, le développement de KoinKoin peut se poursuivre sans que nous devions trop souvent revenir aux briques élémentaires. Sa conception évolutive et sa structure modulaire permettent d'apporter de nouvelles fonctionnalités qui s'intégreront facilement à l'ensemble du système en ayant pas ou peu de changements à faire sur le noyau ou les autres services.

Voici quelques idées des évolutions futures possibles, à court ou moyen terme :

- Implémentation d'un système de fichier, puis plus tard d'une interface unifiée pour pouvoir implémenter d'autres systèmes de fichiers.
- Évolutions du shell, et implémentation ou portage d'autres utilitaires de base d'un système d'exploitation.
- Implémentation de pilotes pour les cartes réseaux, puis implémentation d'une pile TCP/IP.
- Interface unifiée pour l'implémentation des pilotes de périphériques. Ceci comprend l'écriture du service **Dev** et la gestion de la sécurité sur les I/O ports.
- L'écriture du service **Finder**, qui permettra d'identifier un service au sein de KoinKoin OS, et donc de permettre la mise en place d'identifiant de tâche dynamique pour les services.
- Ré-écriture des gestionnaires de mémoire virtuelle et d'espace d'adressage avec utilisation des ensembles.
- Repenser l'implémentation des tty de façon plus cohérente et fonctionnelle.
- A long terme, ajout d'une API compatible POSIX pour permettre le portage de programmes complexes, voire de pilotes.
- Ecriture d'un outil de génération de code pour les services, compte tenu

de la structure unifiée des services de KoinKoin OS.

Deuxième partie

Bibliographie

16.1 Architectures système et matérielle

Voici les différents documents sur lesquels nous nous sommes appuyés lors de la conception et de l'implémentation de KoinKoin :

- "Systèmes d'exploitation 2ième édition", Andrew Tanenbaum
- Documentation et sujets du projet Kaneton
- Intel Architecture Software Developer's Manual (volumes 1 à 3) :
  1. Basic Architecture
  2. Instruction Set Reference
  3. System Programming Guide
- Documentation technique du format Elf (Executable and Linkable Format)
- Spécification technique BIOS (Jean-Pascal Billaud)
- Documentation officielle du micro-noyau Mach
- Documentation officielle du micro-noyau L4
- ATA/ATAPI Specifications
- CHS Translation
- Partition Specifications
- Ralph Interrupt List
- La documentation du projet KOS (Kid Operating System)

16.2 Les noyaux

Définitions des différents types de noyaux, appuyés d'exemple :
http ://fr.wikipedia.org/wiki/Noyau_(informatique)

16.3 Disques durs

Fonctionnement matériel des disques durs
http ://www.commentcamarche.net/pc/disque.php3

Fonctionnement logiciel des disques durs, liste des ports E/S
http ://fr.wikipedia.org/wiki/Integrated_drive_electronics

Troisième partie

Annexes

16.4 Comment utiliser KoinKoin

Les lignes qui commencent par un $ sont des commandes à taper dans votre shell.

16.4.1 Prérequis

- Vous devez avoir Bximage et Grub installés, sinon installez les avec un simple apt-get :)
- Vous devez avoir Sudo installé et configuré, sinon tapez les commandes des étapes 1, 2 et 3 en root
- Vous devez avoir une machine virtuelle comme Qemu ou Bochs installée pour faire tourner KoinKoin
- Vous aurez besoin du programme qemu-img qui est inclut dans le paquet de Qemu si vous voulez utiliser le disque dur

16.4.2 Installation

1. Créer l'image de disquette pour KoinKoin. Le fichier image de disquette contient tous les fichiers nécessaires au chargement de notre noyau et des services fondamentaux.

   Pour le créer, nous utilisons l'utilitaire bximage, fourni avec bochs. Ensuite, nous formatons cette image au format de fichiers ext2, ce qui nous permet ensuite de la monter dans notre système de fichier de travail pour pouvoir y ajouter l'exécutable de notre noyau ainsi que des services fondamentaux.

   $ make image
   (Choisissez fd, 1.44, floppy.img, puis tapez "y")

2. Installer Grub sur l'image de disquette. Pour installer GRUB, il faut tout d'abord copier les fichiers "stage1" et "stage2", ansi que le fichier de configuration "menu.lst" dans le répertoire "boot/grub" de l'image de disquette.

   Le fichier "menu.lst" contient la configuration de base pour GRUB ainsi que la liste des modules à charger. Ces modules incluent notre chargeur de démarrage, notre noyau, ainsi que ses modules qui doivent être chargés au démarrage.

   Ensuite, il faut installer GRUB sur le périphérique utilisé.

$ sudo grub floppy.img
> device (fd0) /home/$USER/koinkoin/floppy.img (remplacer $USER par votre nom d'utilisateur, changez le chemin si KoinKoin n'est pas dans votre répertoire personnel)
> root (fd0)
> setup (fd0)
> quit

Si Grub vous dit quelque chose qui ressemble à ça :

"grub> setup (fd0)
Checking if "/boot/grub/stage1" exists... no
Checking if "/grub/stage1" exists... no

Error 15 : File not found"

Alors recommencez cette étape jusqu'à que ça marche (c'est un bug de Grub)

3. Compiler KoinKoin

   $ make

4. Créer une image de disque dur (optionnel, seulement si vous voulez utiliser un disque dur dans KoinKoin)

   $ qemu-img create disk.img 200M

16.4.3 Démarrer KoinKoin !

Avec Qemu

– Avec un disque dur

  $ qemu -fda $HOME/koinkoin/floppy.img -hda $HOME/koinkoin/disk.img -boot a

– Sans disque dur

```
$ qemu -fda $HOME/koinkoin/floppy.img -boot a
```

Avec Bochs

Vous devez mettre les ligne suivantes dans votre /.bochsrc et commenter les anciennes lignes correspondantes (Seule la première ligne est nécessaire si vous n'utilisez pas de disque dur) :

```
— début des lignes à ajouter à votre /.bochsrc —
floppya : 1_44=/home/$USER/koinkoin/floppy.img, status=inserted
ata0 : enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master : type=disk, path="/home/$USER/koinkoin/disk.img", mode=flat,
cylinders=1024, heads=16, spt=63
— fin des lignes à ajouter à votre /.bochsrc —
```

Remplacez $USER par votre nom d'utilisateur dans ces lignes.

Puis :

```
$ bochs
```

16.4.4 Mises à jour

Après avoir effectué une mise à jour, tapez simplement :

```
$ make
```

Si le fichier "menu.lst" a changé (c'est à dire que certains modules ont été ajoutés ou enlevés), vous devez taper également :

```
$ make menu
```

16.5 Les machines virtuelles

16.5.1 Bochs

Bochs est un émulateur de PC à architecture Intel 32 bits (x86). Il permet également l'émulation de quelques périphériques communs.

Il dispose de fonctionnalités de débogage avancées, c'est pourquoi il est tout à fait adpat au développement de nouveaux noyaux comme c'est notre cas.

La seule limitation de Bochs que nous n'avons pas pu contourner est sa vitesse d'exécution. C'est pourquoi au délà d'un certain stade d'évolution de KoinKoin, sa lenteur est devenu suffisament pénible pour que le besoin d'un outil de remplacement se fasse sentir.

16.5.2 Qemu

Qemu est un émulateur de processeur qui permet d'émuler plusieurs architectures matérielles dont Intel (x86), sparc, arm et powerpc. Il offre également l'émulation d'un certain nombre de périphériques.

De plus, sa vitesse d'exécution est sans commune mesure avec celle de Bochs. C'est la raison pour laquelle pour utilisons principalement Qemu pour développer KoinKoin. Cependant, son manque de fonctionnalités de débogage intégrées nous conduit à utiliser Bochs de temps à autre pour corriger certains problèmes difficiles à diagnostiquer avec Qemu.

16.6 Démarrage réel

16.6.1 Démarrage à partir d'une disquette

Pour démarrer KoinKoin à partir d'une disquette, la procédure la plus simple est de monter notre image de disquette dans un répertoire, puis de copier tous les fichiers de cette image sur une disquette. Ensuite, il faut installer GRUB sur cette disquette de la même façon que décrit précédemment. Il est alors possible de démarrer KoinKoin en choisissant le démarrage à partir d'une disquette dans les options de démarrage du BIOS de l'ordinateur sur lequel on souhaite le lancer, en ayant bien évidemment la disquette dans le lecteur de disquette au moment du démarrage.

16.6.2 Démarrage à partir d'une clé USB

Les clés USB fonctionnent essentiellement de la même façon que les disquettes. Ainsi, pour démarrer à partir d'une clé USB, il suffit, de façon similaire au démarrage à partir d'une disquette, de copier les fichiers de l'image de disquette sur la clé USB et d'installer GRUB sur celle-ci. On peut alors

démarrer KoinKoin à partir de cette clé USB en choisissant ce mode de démarrage dans les options du BIOS de l'orfinateur sur lequel on souhaite lancer KoinKoin, avec, de même, la clé USB branchée sur l'ordinateur au moment du démarrage.

Pour les ordinateurs dont le BIOS ne supporte pas le démarrage à partir d'une clé USB, il est également possible, si GRUB est déjà installé sur cet ordinateur, de démarrer sur la clé USB grâce à celui-ci. Les lignes à indiquer à GRUB pour effectuer ce démarrage sont généralement :
root (hd1,0) boot

La première ligne, celle contenant "root", peut varier en fonction de l'ordinateur utilisé.

16.6.3 Démarrage à partir d'un CDROM

Il est également possible de démarrer KoinKoin à partir d'un CDROM. Cependant, étant donné la taille réduite de notre noyau et de ses modules et le manque de souplesse des CDROM, nous n'avons jamais testé cette méthode. Mais elle pourrait s'avérer utile dans le futur quand la taille totale de notre noyau et de ses modules sera plus importante, et si on souhaite le déployer plus facilemenent, par exemple en distribuant des CDROM permettant de démarrer KoinKoin à nos amis.