

RustOS

Système d'exploitation en Rust

Orphée Antoniadis

Projet de Bachelor - Prof. Florent Glück

Hepia ITI 3ème année

Semestre de Printemps 2017-2018

Résumé

Le but de ce projet est d'étudier le langage Rust, en particulier son utilisation pour l'implémentation d'un système d'exploitation de type *bare metal*. Le langage Rust se révèle particulièrement intéressant en tant que digne successeur de C : beaucoup plus robuste que ce dernier et potentiellement tout aussi rapide. La première partie du projet sera de comprendre les paradigmes de programmation utilisés par Rust ainsi que ses caractéristiques principales. Dans un deuxième temps, il s'agira d'implémenter un système d'exploitation très simple, similaire à celui réalisé au cours logiciel « Programmation système avancée » mais écrit en Rust plutôt qu'en C.

Table des matières

1	Introduction	9
1.1	Contexte	9
1.2	Objectif	9
2	Analyse	10
3	Conception	11
3.1	Environnement de développement	11
3.2	Technologies	11
3.2.1	Nasm	11
3.2.2	Rustup	11
3.2.3	Cargo et Xargo	11
3.2.4	QEMU	12
3.3	Architecture	12
4	Rust	13
5	Exécution du <i>kernel</i>	14
5.1	Compilation	14
5.2	<i>Linking</i>	14
5.3	<i>Boot</i>	15
5.3.1	Principe général	15
5.3.2	GRUB	16
5.3.3	Image ISO	16
6	Gestion mémoire	17
6.1	Introduction	17
6.2	GDT	18
6.3	Segmentation	20
6.4	Pagination	21
6.4.1	Principe général	21
6.4.2	Activer la Pagination	23
7	Peripherals	24
7.1	Interruptions	24
7.1.1	Introduction	24
7.1.2	IDT	24
7.2	Ports	24
7.3	VGA	24
7.4	<i>Timer</i>	24
7.5	Clavier	24
8	Système de fichiers	25
8.1	Introduction	25
8.2	Structure	25
9	Résultats	26

10 Discussions	27
10.1 Problèmes rencontrés	27
10.2 Améliorations possibles	27
11 Conclusion	28
12 Références	29

Table des figures

1	Strucutre du fichier ELF	15
2	<i>Boot</i> d'une machine à base de BIOS	15
3	Exemple d'adressage mémoire	17
4	Protection mémoire avec un MMU	17
5	Exemple d'une GDT	18
6	Structure d'une entrée dans la GDT	18
7	<i>Access byte et Flags</i>	19
8	Descripteur de GDT	19
9	Translation d'adresse	20
10	Structure d'un sélecteur de segment	21
11	Structure d'une <i>Page Entry</i>	22
12	Exemple de pagination à 3 niveaux	22
13	Couleurs du mode texte VGA	24

Remerciements

Convention typographique

Lors de la rédaction de ce document, les conventions typographique ci-dessous ont été adoptées.

- Tous les mots empruntés à la langue anglaise ont été écrits en *italique*
- Toute référence à un nom de fichier (ou dossier), un chemin d'accès, une utilisation de paramètre, variable, ou commande utilisable par l'utilisateur, est écrite avec la police d'écriture **Courier New**.
- Tout extrait de fichier ou de code est écrit selon le format suivant :

```
1   fn main() {  
2       println!("Hello, world!");  
3   }
```

Acronymes

BIOS *Basic Input Output System.* 14, 15

CPU *Central Processing Unit.* 18, 19, 23

ELF *Executable and Linkable Format.* 10, 11, 14

GCC *GNU Compiler Collection.* 11, 13

GDT *Global Descriptor Table.* 17–20

GRUB *GRand Unified Bootloader.* 11, 15

IA-32 *Intel Architecture, 32-bit.* 16, 17

IDT *Interrupt Descriptor Table.* 23

ISO *International Organization for Standardization.* 11, 15

LDT *Local Descriptor Table.* 19, 20

MBR *Master Boot Record.* 14, 15

MMU *Memory Management Unit.* 16, 19

OS *Operating System.* 10, 13–15, 20, 22, 23

PC *Personal Computer.* 14

PIO *Port Input/Output.* 23

RAM *Random Access Memory.* 16, 22

VGA *Video Graphics Array.* 23

VRAM *Video Random Access Memory.* 14

1 Introduction

1.1 Contexte

1.2 Objectif

2 Analyse

3 Conception

3.1 Environnement de développement

La machine utilisée pour le développement du projet est un MacBook Pro avec un processeur Intel à 3 GHz. Il a quand même fallu utiliser une machine virtuelle (VMware) utilisant Linux (Ubuntu 16.04.4 LTS) pour la compilation. Ce choix a été fait car il existe beaucoup plus de documentation sur l'implémentation de systèmes d'exploitation sur Linux que sur Mac. Bien que Mac OS soit un système UNIX, les exécutable générés sur cet environnement n'ont pas le même format que ceux générés sur Linux qui sont au format ELF. Ceci rend le développement d'OS légèrement différent sur Mac OS.

3.2 Technologies

3.2.1 Nasm

Bien que le système d'exploitation développé devait être sur Rust, certaines parties ont dû être faites en assembleur car étant trop bas niveau pour le Rust. Ces éléments seront décrits plus loin dans ce document. Nasm a été utilisé pour compiler le code assembleur x86 en ELF 32-bits. Nasm produit des fichiers objets pouvant être liés à d'autres fichiers objets afin de créer un exécutable.

3.2.2 Rustup

Rust sera décrit plus en détails dans un prochain chapitre. Ce qu'il faut savoir est que Rust est distribué sous trois versions différentes. La version *stable*, la version *beta* et la version *nightly*. La version *nightly* possède plus de fonctionnalités mais sa stabilité n'est pas garantie. Cette version a été utilisée pendant le développement du projet et l'utilitaire Rustup a été utilisé pour son installation. Cet utilitaire permet de simplifier l'installation de Rust quand on souhaite une version différente de la dernière version stable de Rust.

3.2.3 Cargo et Xargo

Lors du développement d'un système d'exploitation type *bare metal*, on souhaite s'affranchir de toute dépendance à une librairie externe. Tout doit être refait depuis le début. Le code est donc compilé sans la bibliothèque standard (std). Rust a tout de même besoin d'une base pour être compilé. Cette base est fournie par la librairie **core**. Cette librairie est minimale et permet de ne définir que les primitives de Rust. Pour gérer les dépendances d'un projet Rust, il est conseillé d'utiliser le gestionnaire de paquets cargo. Le problème est que cargo ne permet pas de lier la librairie **core** à un projet. Heureusement, un autre utilitaire basé sur cargo existe et permet d'installer par défaut la librairie **core** pour des projets sans bibliothèque standard. Cet utilitaire se nomme xargo et est utilisé pour compiler le code Rust en fichiers objets

3.2.4 QEMU

Le compilateur GCC a été utilisé pour *linker* les fichiers objet générés par nasm et xargo. GCC génère un fichier au format ELF. Pour utiliser ce fichier comme un système d'exploitation *bootable*, il faut en faire une image ISO *bootable*. Pour se faire, l'utilitaire **genisoimage** est utilisé, couplé au *bootloader* GRUB. L'image ISO est finalement exécutée par la machine virtuelle QEMU. QEMU est une machine virtuelle pouvant émuler une architecture. Pour ce projet, l'architecture i386 a été choisie afin d'émuler un processeur Intel 32-bits.

3.3 Architecture

4 Rust

5 Exécution du *kernel*

5.1 Compilation

Quand on veut compiler un simple code C en utilisant GCC par exemple, le compilateur passe par plusieurs étapes. Le préprocesseur génère d'abord un fichier C en fonction des directives de préprocesseur. Ce fichier C est ensuite compilé en code assembleur qui est lui même compilé en code objet. Le *linker* permet ensuite de lier les différents fichiers objets et générer un exécutable. Nous avons déjà eu un aperçu des différentes étapes de la compilation d'un OS de type *bare metal* dans la partie 3.2. A la différence de la compilation d'un code C, nous avons d'un côté du code assembleur et de l'autre du code Rust. Nasm et cargo permettent tous deux de générer des fichiers objets. Il n'y a donc que la dernière étape à effectuer ce que GCC permet de faire avec la commande suivante.

```
gcc $(OBJS) -T $(LINKER) -static -m32 -ffreestanding -nostdlib -o $@ $(RUST)
```

Ici, `$(OBJS)` représente les fichiers objets générés par `nasm`, `$(LINKER)` est un fichier permettant de faire l'édition des liens et `$(RUST)` représente les fichiers objets générés par Rust.[1]

5.2 Linking

Nous avons vu dans la partie précédente que GCC a besoin d'un fichier pour faire l'édition des liens. Si ce fichier n'est pas donné, il en utilise un par défaut. Le *linker* permet de structurer le code par sections. Prenons pour exemple le *script* utilisé pour ce projet.

```
1 ENTRY(entrypoint)
2 SECTIONS {
3     . = 1M;
4     .boot ALIGN(4) :
5     {
6         *(.multiboot)
7     }
8     .stack ALIGN(4) :
9     {
10        *(.stack)
11    }
12    .text ALIGN(4K) :
13    {
14        *(.text*)
15    }
16    .rodata ALIGN(4K) :
17    {
18        *(.rodata*)
19    }
20    .data ALIGN(4K) :
21    {
22        *(.data*)
23    }
24    .bss ALIGN(4K) :
25    {
26        *(COMMON)
27        *(.bss*)
28    }
29 }
```

L'appel à `ENTRY` permet de spécifier l'entrée du *kernel*. Pour un simple programme en C l'entrée serait le *main*. Ici, ce sera l'entrée de notre *kernel* donc la première fonction exécutée au *boot*. `SECTION` va dire au linker où placer les parties du code. Par exemple, la section `.text` contiendra le code et la section `.data` contiendra les variables initialisées [1, 2, 3, 4]. Voici donc la structure du fichier ELF qui serait généré à l'aide de ce *script*.

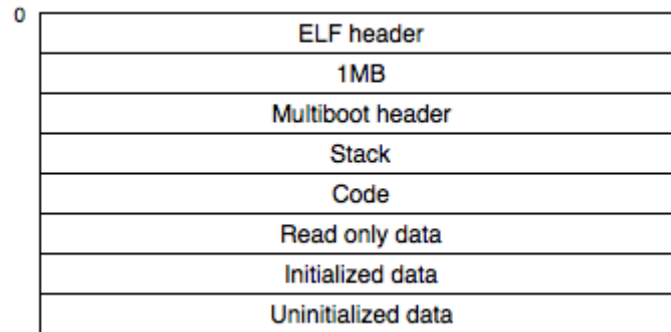


FIGURE 1 – Strucutre du fichier ELF

A noter que les sections commencent avec un *offset* de 1MB. Nous avons eu besoin de faire ça car les premiers 1MB dans un OS sont réservés [1, 5]. La mémoire vidéo (VRAM) se situe par exemple dans cette zone.

5.3 Boot

5.3.1 Principe général

Quand un ordinateur est allumé, un signal est envoyé à la carte mère qui démarre l'alimentation. Le processeur démarre alors en mode 16-bits. Le signal "Power Ok" est envoyé au BIOS qui est le *firmware* du PC (localisé en mémoire flash de la carte mère). Le BIOS initialise alors la séquence POST (*Power On Self Test*) qui vérifie que chaque périphérique est alimenté et que la mémoire est ok puis initialise chaque périphérique et enfin redonne la main au BIOS qui continue le *boot*. Le BIOS charge ensuite les 512 premiers bytes (MBR) du premier disque qui doit charger le *kernel* en mémoire et l'exécuter. Pour résumer, le *boot* d'une machine à base de BIOS se déroule de la manière ci-dessous.[1]

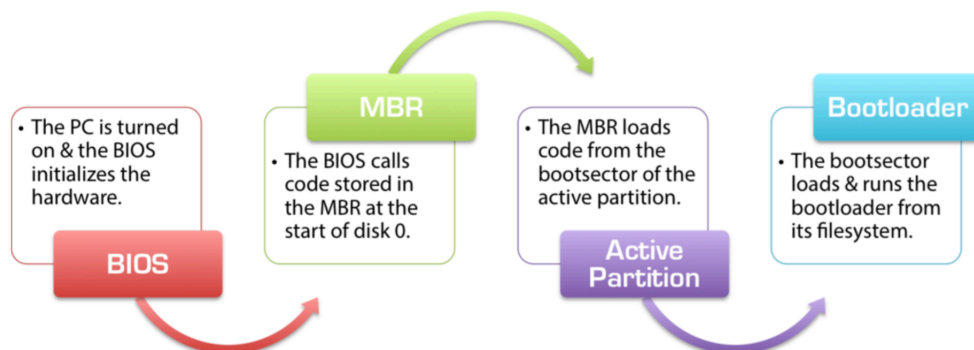


FIGURE 2 – Boot d'une machine à base de BIOS

5.3.2 GRUB

Le MBR contient ce qui est appelé le *bootloader*. Le *bootloader* est le morceau de code qui va charger le *kernel* en mémoire et l'exécuter. C'est ici qu'entre en scène GRUB. GRUB est un *bootloader* puissant et versatile permettant de charger n'importe quel type de système d'exploitation. Son initialisation se fait par étapes.

- *Stage 1* : Chargé en mémoire par le BIOS depuis le MBR, il contient le code pour charger le *Stage 1.5*
- *Stage 1.5* : Chargé en mémoire par le *Stage 1*, il contient les drivers nécessaires à l'accès au système de fichiers par le *Stage 2*
- *Stage 2* : Chargé en mémoire par le *Stage 1.5*, il affiche le menu de GRUB. Il permet de sélectionner et charger un OS

GRUB permet de charger n'importe quel type de système d'exploitation grâce au standard *Multiboot*. Ce standard permet à tout *bootloader* de charger tout OS compatible [1, 6].

5.3.3 Image ISO

Nous avons déjà pu voir que le *boot* du *kernel* se faisait à partir d'une image ISO dans la partie 3.2.4. Pour qu'une image ISO soit *bootable*, il est nécessaire que GRUB soit installé dans les huit premiers KB du disque. Prenons l'arborescence suivante :

```
isofiles
  boot
    grub
```

Les fichiers `kernel.elf` (kernel sur lequel nous voulons *booter*), `menu.lst` (fichier de configuration de GRUB) et `stage2_eltorito` doivent être copiés de manière à obtenir l'arborescence suivante :

```
isofiles
  boot
    grub
      menu.lst
      stage2_eltorito
    kernel.elf
```

Pour finir, il faut exécuter la commande :

```
genisoimage -R -b boot/grub/stage2_eltorito -input-charset utf8 -no-emul-boot \
-boot-info-table -o os.iso isofiles
```

Cette commande générera une image ISO *bootable* nommée `os.iso`[1].

6 Gestion mémoire

6.1 Introduction

Le système d'exploitation développé est exécuté sur une architecture IA-32 aussi appelée i386. Ceci qui veut dire que la mémoire est adressée sur 32 bits. $2^{32} = 4Go$, donc la mémoire physique (RAM) a une taille totale de 4Go dans notre système d'exploitation. Lorsqu'une tâche est exécutée, elle est chargée en mémoire et est définie par la paire base et limite. La base est son adresse physique dans la RAM et la limite est sa taille. La figure 3 donne un exemple d'adressage de plusieurs processus.[1]

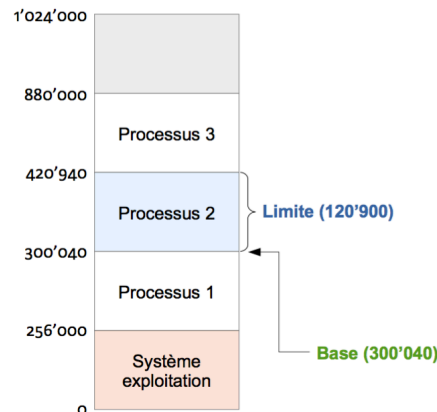


FIGURE 3 – Exemple d'adressage mémoire

Une tâche possède son propre espace d'adressage dit virtuel. Pour le processus 1 de la figure 3, l'adresse 0 est en fait à l'adresse physique 300040. Il y a donc besoin de traduire l'adresse virtuelle en adresse physique. C'est là qu'entre en jeu le MMU (Memory Management Unit). Le MMU est un dispositif matériel permettant de faire cette translation d'adresses. A chaque référencement mémoire, il va convertir l'adresse virtuelle en adresse physique et regarder si elle ne dépasse pas la limite du processus. Le MMU permet donc aussi de protéger la mémoire car il va empêcher toute référence à une zone extérieure au processus (voir figure 4).[1]

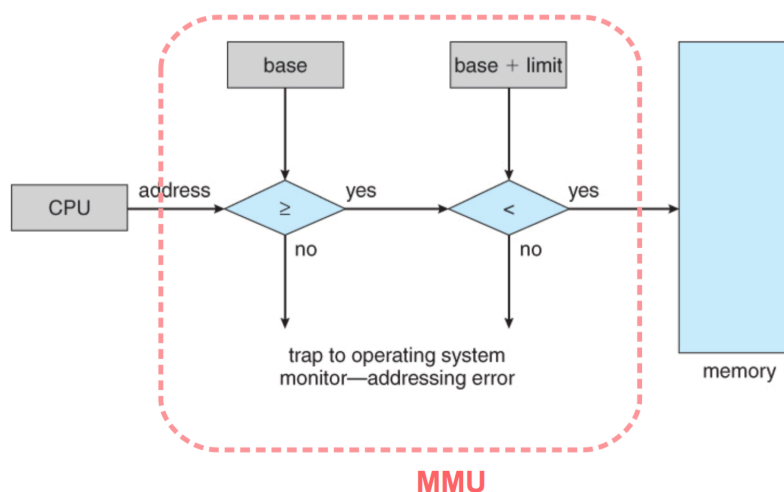


FIGURE 4 – Protection mémoire avec un MMU

6.2 GDT

Dans une architecture IA-32, la translation d'adresses se fait à l'aide de descripteurs définissant des segments de mémoire. Ces descripteurs sont contenus dans une table de descripteurs. Cette table est la GDT (Global Descriptor Table) [7]. Chaque descripteur est défini par sa base (son adresse physique), sa limite (sa taille) et son niveau de privilèges (allant de 0 à 3, le niveau 0 ayant le plus de privilèges et le niveau 3 le moins). Ci dessous, la figure 5 montre un exemple d'une GDT.[1]

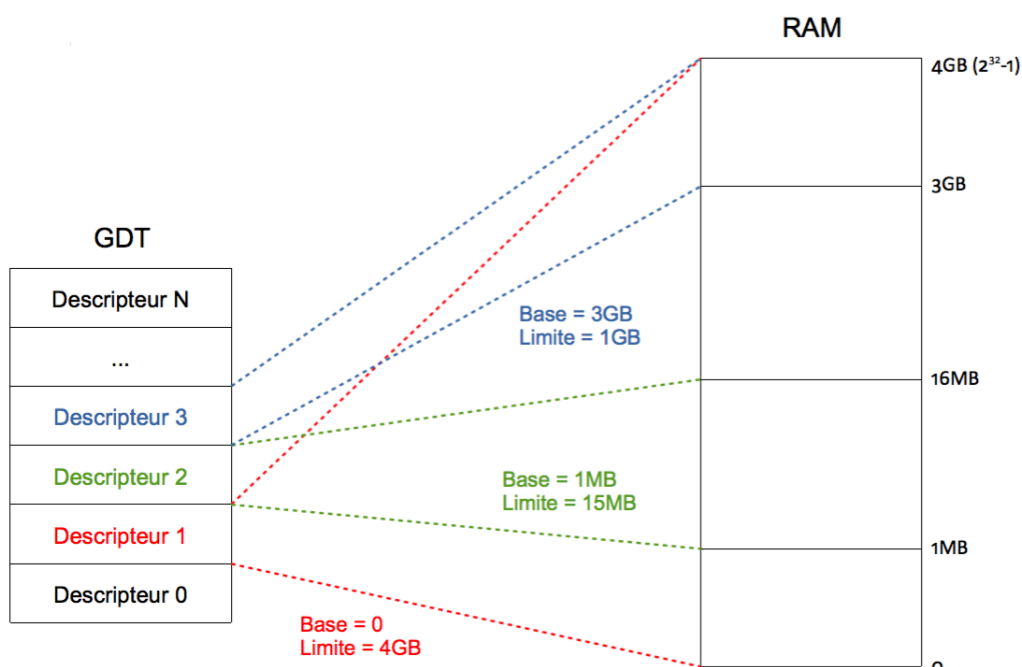


FIGURE 5 – Exemple d'une GDT

La GDT est contenue en mémoire. Chaque entrée (ou descripteur) de la table de descripteurs sont sur 64 bits et sont définis par la structure de la figure 6.[7]

31				16		15				0					
Base 0:15						Limit 0:15									
63		56		55 52		51 48		47		40		39		32	
Base 24:31				Flags		Limit 16:19		Access Byte				Base 16:23			

FIGURE 6 – Structure d'une entrée dans la GDT

- La base est sur 32 bits et est divisée en 3 parties dans une entrée. Les bits 16 à 31, 32 à 39 et 56 à 63 contiennent la base
- La limite est sur 20 bits et est divisée en 2 parties dans une entrée. Les bits 0 à 15 et 48 à 51 contiennent la limite
- L'*Access byte* contient des bits de contrôle pour l'accès aux données du segment (privilèges, écriture ou lecture, etc...). Il est décrit plus en détails dans la figure 7

- Les *Flags* sont aussi des bits de contrôle et sont décrits plus en détails dans la figure 7[7]

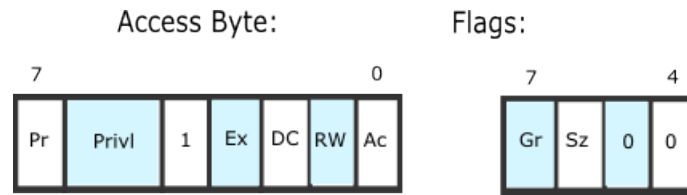


FIGURE 7 – Access byte et Flags

- Pr : *Present* bit, doit être à 1 si le segment est valide
- Privl : Niveau de privilèges sur deux bits
- Ex : *Executable bit*, est à 1 si le segment peut être exécuté (par exemple dans un segment de code, ce bit est à 1 alors que dans un segment de données, ce bit est à 0)
- DC : *Direction bit*
- RW : Bit de lecture/ écriture
- Ac : *Accessed bit*, ce bit est mis à 1 lorsque le CPU accède à ce segment
- Gr : Bit de granularité, à 0 la limite est en octets, à 1 la limite est en blocs de 4Ko
- Sz : *Size bit*, à 0 le segment est sur 16 bits, à 1 le segment est sur 32 bits

Par exemple, pour obtenir un segment sur toute la mémoire disponible, il faut mettre le bit de granularité à 1 (pour avoir une limite en blocs de 4Ko) et mettre la limite à 0xFFFFF. Une fois la GDT construite, il faut utiliser l'instruction `lgdt` pour la charger. L'adresse du descripteur de la GDT doit être donnée en argument à cette instruction. Le descripteur de GDT est défini par la structure 48-bits décrite dans la figure 8.[7]

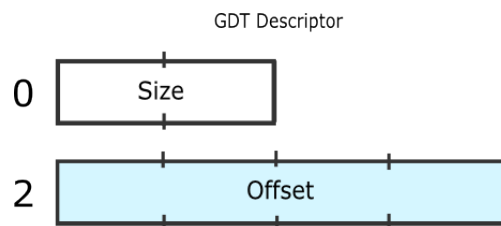


FIGURE 8 – Descripteur de GDT

- *Size* est la limite sur 16 bits (c'est à dire la taille de la GDT - 1)
- *Offset* est l'adresse physique de la GDT sur 32 bits

6.3 Segmentation

La segmentation est une technique permettant de découper la mémoire en segments de mémoire logique. Une adresse logique est convertie par le MMU en adresse linéaire en utilisant une GDT ou une LDT. Si la pagination (dont on parlera plus tard) est activée, l'adresse linéaire est convertie en adresse physique. Toute cette mécanique est décrite dans la figure 9. La segmentation permet de faire la conversion d'adresse logique en adresse linéaire et est obligatoire en mode protégé (32-bits).[8]

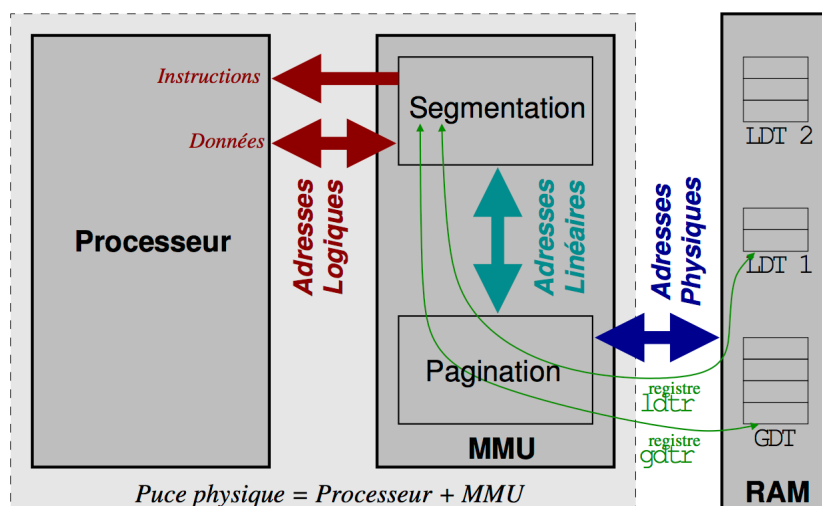


FIGURE 9 – Translation d'adresse

La gestion de la segmentation par le CPU se fait à l'aide de registres spéciaux nommés registres de segment. Ces registres sont au nombre de 6 et ont chacun une taille de 16 bits.[1, 9]

Registre	Segment
CS	<i>Code Segment</i>
DS	<i>Data Segment</i>
SS	<i>Stack Segment</i>
ES	<i>Extra Segment</i>
FS GS	<i>General Purpose Segments</i>

En mode protégé (32-bits), ces registres doivent pointer sur des descripteurs de segment de la GDT. Au minimum les trois premiers registres décrits doivent être utilisés en mode protégé (CS, DS, et SS). Les opérations adressant le code (décodage des instructions en mémoire, sauts, etc...) référencent le descripteur de segment sur lequel pointe le registre CS. Les opérations adressant les données (adressage de variables ou d'adresses mémoires) référencent le descripteur de segment sur lequel pointe le registre DS. Les opérations adressant la pile (**push** et **pop**) référencent le descripteur de segment sur lequel pointe le registre SS.

Nous avons vu qu'un descripteur de segment fait 64 bits et un registre de segment fait 16 bits. Ceci est possible car le registre de segment ne va pas contenir l'intégralité d'une entrée dans la GDT mais un sélecteur de cette entrée. Un sélecteur a une taille de 16 bits

(comme les registres de segment) et contient l'index du descripteur dans la GDT, un bit indiquant si l'entrée est dans la GDT ou dans la LDT ainsi que son niveau de privilège (figure 10).[1]

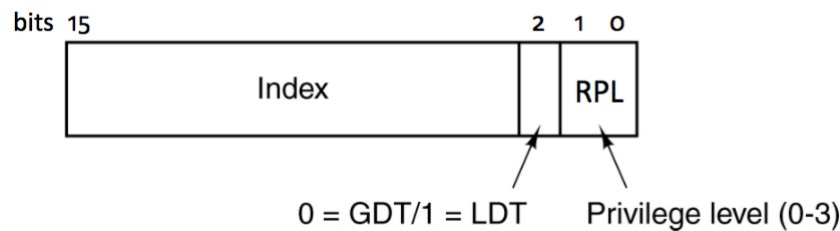


FIGURE 10 – Structure d'un sélecteur de segment

Pour récupérer l'index dans la GDT d'un segment à partir de son descripteur, il faut donc faire un décallage de 3 bits. Prenons un segment si situant à l'index 2 de la GDT. Si on veut initialiser le segment de code (registre CS) avec ce segment, il faut mettre la valeur 16 dans le registre CS ($2 \ll 3 = 16$).

Dans un premier temps, l'OS développé a eu un adressage segmenté de type *FLAT*, c'est-à-dire que toute la mémoire était accédée de manière linéaire. Ceci se fait en initialisant trois descripteurs dans la GDT. Un descripteur nul à l'index 0 (obligatoire dans tous les modèles de segmentation), un segment de code couvrant toute la mémoire et un segment de données couvrant aussi toute la mémoire. Les segments de code et de données adressent donc les mêmes zones de mémoire. On verra par la suite que d'autres entrées ont été ajoutées à la GDT pour la gestion des tâches.

6.4 Pagination

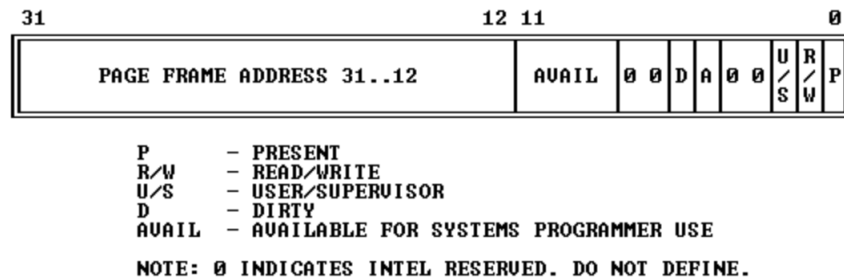
6.4.1 Principe général

La pagination est une autres techinque de gestion de mémoire qui diffère de la segmentation. Alors que la segmentation permet d'allouer des morceaux de mémoire de taille variable, la pagination divise la mémoire en blocs de taille fixe appelés pages (de 4Ko, 2Mo ou 4Mo). De plus, la segmentation est obligatoire dans une architecture i386 alors que la pagination ne l'est pas[10]. Quand une tâche fait référence à une adresse logique en mémoire, cette adresse est convertie en adresse linéaire grâce au mécanisme de segmentation et c'est le mécanisme de pagination qui permet de translater cette adresse linéaire en adresse physique (comme vu précédemment dans la partie sur la segmentation). Quand la pagination est activée, l'adresse linéaire est divisée en deux parties lorsque des pages de 4Mo sont utilisées et en trois parties lorsque des pages de 4Ko sont utilisées. Le *kernel* développé utilise des pages de 4Ko, une adresse linéaire est donc sous la forme suivante :

- 10 bits pour le *directory index*
- 10 bits pour le *page index*
- 12 bits pour l'*offset*

On dit que cette pagination est une pagination à trois niveaux. En général, une pagination à trois niveaux est utilisée mais il peut exister des systèmes utilisant plus ou moins de niveaux. Le système d'exploitation doit créer un répertoire de pages (*Page Directory*) et

au moins une table de pages (*Page Table*) pour chaque tâche. Les répertoires et les tables de pages ont la taille d'une page et sont composées d'entrées sur 32 bits (4 octets). Une entrée dans un répertoire permet d'adresser une table de pages et une entrée dans une table permet d'adresser une page. Dans notre cas, un répertoire permet donc d'adresser 1024 tables et une table 1024 pages ce qui permet bien d'adresser au total 4Go ($1024 \times 1024 \times 4096$). Une entrée est sur 32 bits mais seulement les 20 bits de poids fort sont utilisés pour l'adressage car les adresses sont alignées avec 4096 ce qui laisse les 12 bits de poids faible pour la configuration.[11]

FIGURE 11 – Structure d'une *Page Entry*

Quand une adresse linéaire est lue, le *directory index* permet de lire la bonne entrée dans le *Page Directory*. Il faut ensuite utiliser le *page index* pour récupérer la bonne entrée dans la table des pages. De la même manière que l'entrée dans le répertoire de pages pointait sur une table des pages, l'entrée dans une table des pages pointe sur une *Page Frame*. Cette page contient finalement la donnée pointée par l'adresse linéaire, il faut utiliser l'*offset* pour trouver cette donnée dans la page. La figure 12 résume bien ce mécanisme.[12] A noter que le *Page Directory* est pointé par le registre CR3. A chaque fois qu'un changement de tâche a lieu, le registre CR3 doit être mis à jour avec le *Page Directory* de la nouvelle tâche.[13]

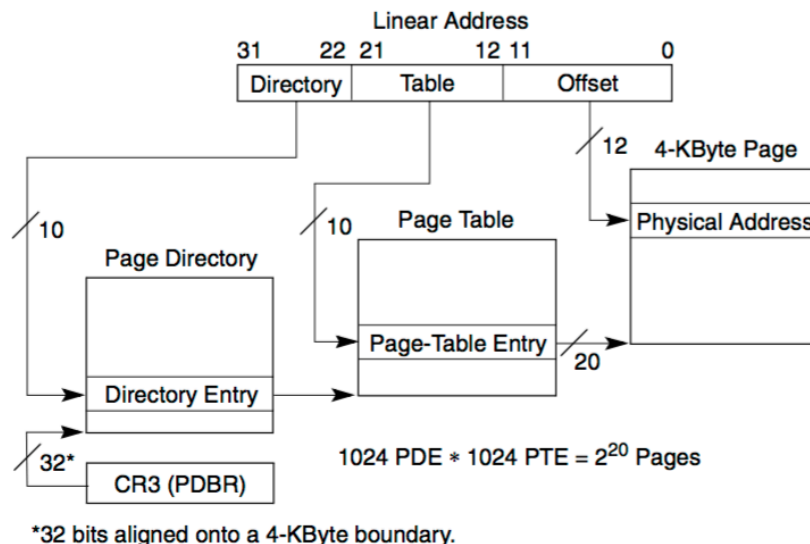


FIGURE 12 – Exemple de pagination à 3 niveaux

6.4.2 Activer la Pagination

Pour initialiser la pagination sur architecture x86, il faut d'abord construire un répertoire de pages valide contenant les entrées vers les pages du *kernel*. Il est obligatoire de commencer par cela car si la pagination est activée et que le *kernel* n'est pas *mappé* dans le répertoire chargé, une exception sera levée (*Page Fault*). Par soucis de simplicité pour la suite du développement de l'OS, le *kernel* va être déplacé au dernier Go de la RAM. Grâce à la pagination, ceci peut se faire assez simplement, il suffit de compléter le répertoire de pages ainsi que ses tables de pages correctement. Pour rappel, le *kernel* commence à l'adresse 0x100000 (1Mo) mais il faut aussi rendre accessible le premier Mo de RAM. Il faut donc déplacer les adresses physiques allant de 0x0 à la fin du *kernel* (qui n'est pas fixe). Dans un premier temps, le *linker* doit être modifié de cette manière :

```
1  SECTIONS {
2      /* Low memory Kernel */
3      . = 0x00100000;
4      .boot ALIGN(4) :      { *(.multiboot) }
5      .low_text ALIGN (4K) : { *(.low_text) }
6      .low_data ALIGN (4K) : { *(.low_data) }
7      .low_bss ALIGN (4K) :  { *(.low_bss) }
8
9      /* Higher-half Kernel */
10     . += 0xC0000000;
11     .stack ALIGN(4) : AT(ADDR(.stack) - 0xC0000000)    { *(.stack) }
12     .text ALIGN(4K) : AT(ADDR(.text) - 0xC0000000)     { *(.text*) }
13     .rodata ALIGN(4K) : AT(ADDR(.rodata) - 0xC0000000) { *(.rodata*) }
14     .data ALIGN(4K) : AT(ADDR(.data) - 0xC0000000)     { *(.data*) }
15     .bss ALIGN(4K) : AT(ADDR(.bss) - 0xC0000000)       { *(COMMON) *(.bss*) }
16 }
```

Ici, le *kernel* est divisé en deux parties. La première est celle qui va être appelée au démarrage du système et qui va initialiser la pagination. Une fois la pagination active, le *kernel* va continuer son exécution dans la deuxième partie qui est située dans le dernier Go de RAM. Nous sommes obligés de démarrer le *kernel* au début de la mémoire physique car toutes les adresses sont virtuelles. En réalité, le *kernel* dispose de beaucoup moins (variable selon la configuration de l'émulateur, ici QEMU). Il n'existe donc pas d'adresse physique située à 3Go dans la mémoire physique du *kernel* et il est donc impossible de démarrer le système à cette adresse.

7 Périphériques

7.1 Interruptions

7.1.1 Introduction

7.1.2 IDT

7.2 Ports

Les ports d'entrées/sorties sur architecture x86 se situent dans un espace d'adresses séparé de la mémoire physique. Il n'est donc pas possible d'écrire dans un PIO de la même manière que l'on écrirait dans la mémoire (avec une instruction `MOV`). Ainsi, le CPU utilise des instructions spéciales pour accéder aux périphériques PIO. Ces instructions sont les instructions `IN` et `OUT`. `IN` permet de lire tandis que `OUT` permet d'écrire. A noter que l'adresse du port doit toujours être spécifiée dans le registre `dx` et la lecture et l'écriture se font toujours avec les registres `ax/al`. [1]

7.3 VGA

Dans l'OS développé, le mode texte VGA a été utilisé pour l'affichage. Toute carte graphique offre ce mode texte de 80 colonnes par 25 lignes. Les 16 couleurs disponibles sont les suivantes : [14]




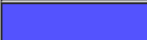












Number	Colour	Name	Number + bright bit	bright Colour	Name
0		Black	0+8=8		Dark Gray
1		Blue	1+8=9		Light Blue
2		Green	2+8=A		Light Green
3		Cyan	3+8=B		Light Cyan
4		Red	4+8=C		Light Red
5		Magenta	5+8=D		Light Magenta
6		Brown	6+8=E		Yellow
7		Light Gray	7+8=F		White

FIGURE 13 – Couleurs du mode texte VGA

7.4 *Timer*

7.5 Clavier

8 Système de fichiers

8.1 Introduction

8.2 Structure

9 Résultats

10 Discussions

10.1 Problèmes rencontrés

10.2 Améliorations possibles

11 Conclusion

12 Références

- [1] Florent Glück. Programmation système avancée, 2017.
- [2] Linker scripts (osdev). https://wiki.osdev.org/Linker_Scripts.
- [3] Linker scripts (scoberlin). http://www.scoberlin.de/content/media/http/informatik/gcc_docs/ld_3.html.
- [4] Linker scripts (math.utah.edu). https://www.math.utah.edu/docs/info/ld_3.html.
- [5] Memory map (x86). [https://wiki.osdev.org/Memory_Map_\(x86\)](https://wiki.osdev.org/Memory_Map_(x86)).
- [6] Multiboot specifications. <https://www.gnu.org/software/grub/manual/multiboot/multiboot.html>.
- [7] Gdt. <https://wiki.osdev.org/GDT>.
- [8] David Decotigny et Thomas Petazzoni. Segmentation et interruptions. <http://sos.enix.org/wiki-fr/upload/SOSDownload/sos-texte-art2.pdf>.
- [9] Segmentation. <https://wiki.osdev.org/Segmentation>.
- [10] David Decotigny et Thomas Petazzoni. Mise en place de la pagination. <http://sos.enix.org/wiki-fr/upload/SOSDownload/sos-texte-art4.pdf>.
- [11] Page translation. https://pdos.csail.mit.edu/6.828/2011/readings/i386/s05_02.htm.
- [12] Intel. Ia-32 intel architecture - software developer's manual - volume 3 : System programming guide. <http://flint.cs.yale.edu/cs422/doc/24547212.pdf>, 2003.
- [13] Jean Gareau. Advanced embedded x86 programming : Paging, june 1998.
- [14] Text ui. https://wiki.osdev.org/Text_UI.
- [15] Rust book first edition. <https://doc.rust-lang.org/book/first-edition>.
- [16] Rust book second edition. <https://doc.rust-lang.org/book/second-edition>.
- [17] Cargo book. <https://doc.rust-lang.org/cargo>.
- [18] Target option. https://doc.rust-lang.org/1.1.0/rustc_back/target/struct.Target.html.
- [19] Target i386 example. <https://github.com/rust-lang/rust/issues/33497>.
- [20] `__floatundisf` issue. <https://users.rust-lang.org/t/kernel-modules-made-from-rust/9191>.
- [21] Writing an os in rust. <https://os.phil-opp.com>.
- [22] Writing an os in rust (second edition). <https://os.phil-opp.com/second-edition>.
- [23] Setting up paging. https://wiki.osdev.org/Setting_Up_Paging.