RustOS
Système d'exploitation en Rust

Orphée Antoniadis

Projet de Bachelor - Prof. Florent Glück

Hepia ITI 3ème année

Semestre de Printemps 2017-2018

h e p i a
Haute école du paysage, d'ingénierie
et d'architecture de Genève

Hes·SO GENÈVE
Haute Ecole Spécialisée
de Suisse occidentale

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 **8**
1.1 Contexte 8
1.2 Objectif 8

2 Analyse **9**

3 Conception **10**
3.1 Environnement de développement 10
3.2 Technologies 10
3.3 Architecture 11

4 Rust **12**

5 Exécution du *kernel* **13**
5.1 Compilation 13
5.2 *Linking* 13
5.3 *Boot* 14

6 Gestion mémoire **16**
6.1 Introduction 16
6.2 GDT 17
6.3 Segmentation 19
6.4 Pagination 20

7 Periphériques **24**
7.1 Ports 24
7.2 Interruptions et Exceptions 24
7.3 VGA 27
7.4 *Timer* 30
7.5 Clavier 30

8 Système de fichiers **31**
8.1 Introduction 31
8.2 Structure 31

9 Tâches utilisateur **32**

10 Résultats **33**

11 Discussions **34**
11.1 Problèmes rencontrés 34
11.2 Améliorations possibles 34

12 Conclusion **35**

13 Références **36**

Table des figures

1 Strucutre du fichier ELF 14
2 *Boot* d'une machine à base de BIOS 14
3 Exemple d'adressage mémoire 16
4 Protection mémoire avec un MMU 16
5 Exemple d'une GDT 17
6 Structure d'une entrée dans la GDT 17
7 *Access byte et Flags* 18
8 Descripteur de GDT 18
9 Translation d'adresse 19
10 Structure d'un sélecteur de segment 20
11 Structure d'une *Page Entry* 21
12 Exemple de pagination à 3 niveaux 21
13 Répertoire de pages adressant le *kernel* au début de la RAM 22
14 Répertoire de pages adressant le *kernel* à la fin de la RAM 23
15 Table des interruptions et exceptions sur IA-32 25
16 Table de correspondance des IRQs 26
17 Différents types de descripteur d'interruption 28
18 Relation entre le registre IDTR et l'IDT 29
19 Structure d'un caractère en mode texte VGA 29
20 Couleurs disponibles en mode texte VGA 29

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, 24, 27
CRTC *Cathode Ray Tube Controller*. 29

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

GCC *GNU Compiler Collection*. 11, 13
GDT *Global Descriptor Table*. 17–20, 27, 28
GRUB *GRand Unified Bootloader*. 11, 15

IA-32 *Intel Architecture, 32-bit*. 16, 17, 24, 25
IDT *Interrupt Descriptor Table*. 25–28
IRQ *Interrupt Request*. 26
ISO *International Organization for Standardization*. 11, 15
ISR *Interrupt Service Routine*. 24, 26, 27

LDT *Local Descriptor Table*. 19, 20

MBR *Master Boot Record*. 14, 15
MMIO *Memory Mapped Input/Output*. 24
MMU *Memory Management Unit*. 16, 19

NMI *Non Maskable Interrupt*. 26

OS *Operating System*. 10, 13–15, 20, 22, 28, 29

PC *Personal Computer*. 14, 29
PIC *Programmable Interrupt Controller*. 26
PIO *Port Input/Output*. 24

RAM *Random Access Memory*. 16, 22, 23, 28

VGA *Video Graphics Array*. 29
VRAM *Video Random Access Memory*. 14, 29

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 fallut 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 exécutables 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 du 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 dépendences 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.

```
ENTRY(entrypoint)
SECTIONS {
  . = 1M;
  .boot ALIGN(4):
  {
    *(.multiboot)
  }
  .stack ALIGN(4):
  {
    *(.stack)
  }
  .text ALIGN(4K) :
  {
    *(.text*)
  }
  .rodata ALIGN(4K) :
  {
    *(.rodata*)
  }
  .data ALIGN(4K) :
  {
    *(.data*)
  }
  .bss ALIGN(4K) :
  {
    *(COMMON)
    *(.bss*)
  }
}
```

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*.

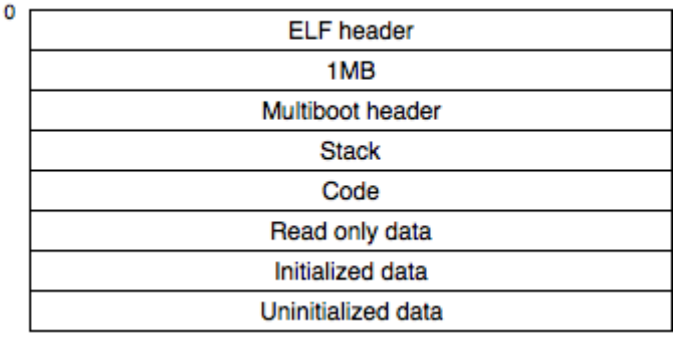

FIGURE 1 – Strucutre du fichier ELF

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 reservés [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]

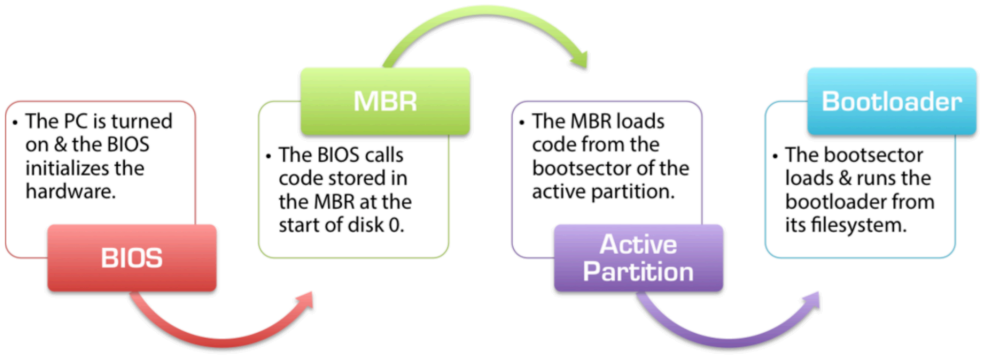

FIGURE 2 – *Boot* d'une machine à base de BIOS

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 grace 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 partie 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 génerera 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 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 tache 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]

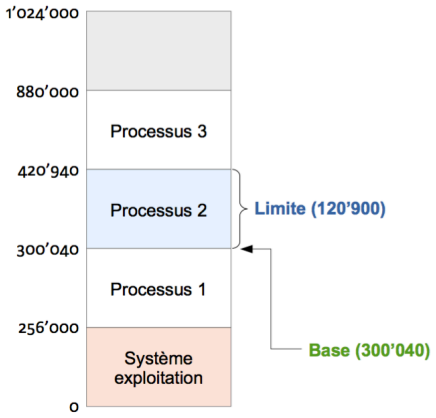

FIGURE 3 – Exemple d'adressage mémoire

Une tache 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 translater l'adresse virtuelle en adresse physique. C'est là qu'entre en jeu le MMU (Memory Mangement 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]

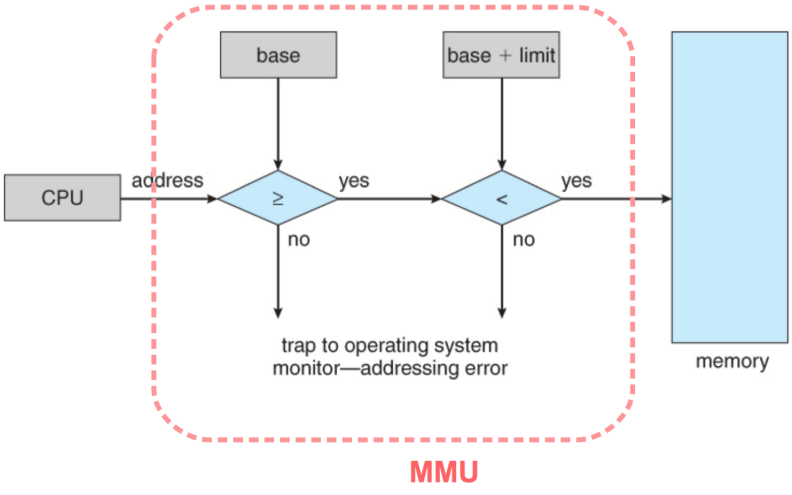

FIGURE 4 – Protection mémoire avec un MMU

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]

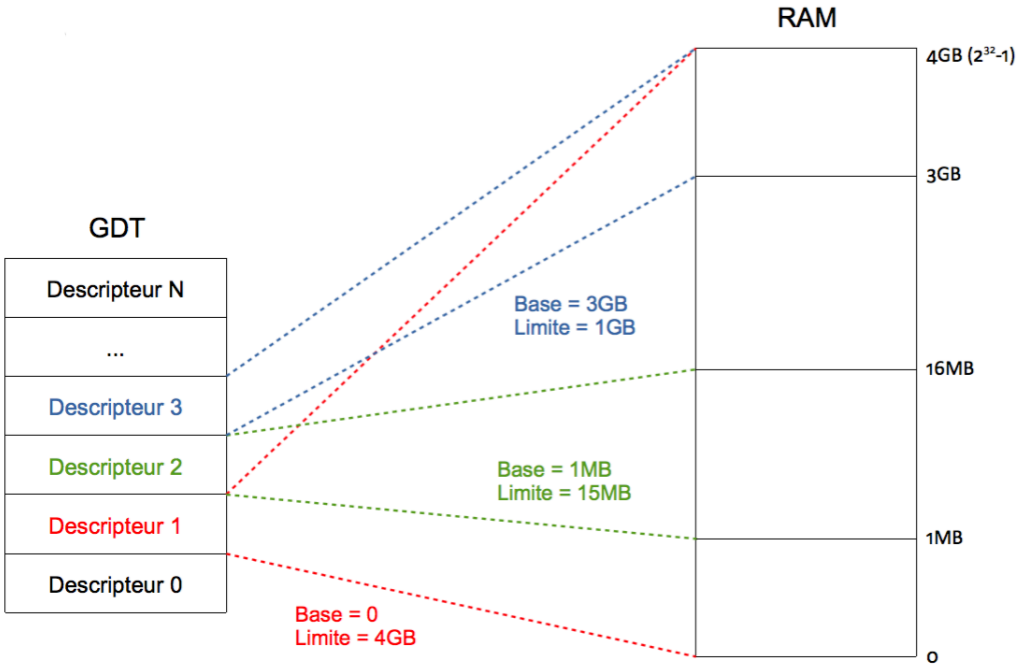

FIGURE 5 – Exemple d'une GDT

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 détail dans la figure 7

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

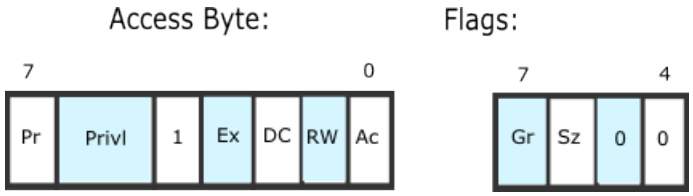

FIGURE 7 – *Access byte et Flags*

- 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]

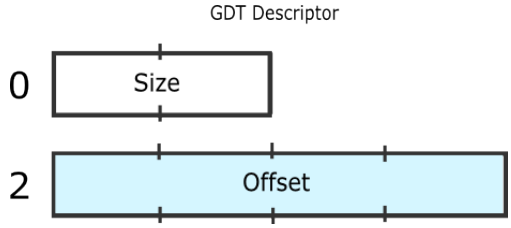

FIGURE 8 – Descripteur de GDT

- *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 convertion d'adresse logique en adresse linéaire et est obligatoire en mode protégé (32-bits).[8]

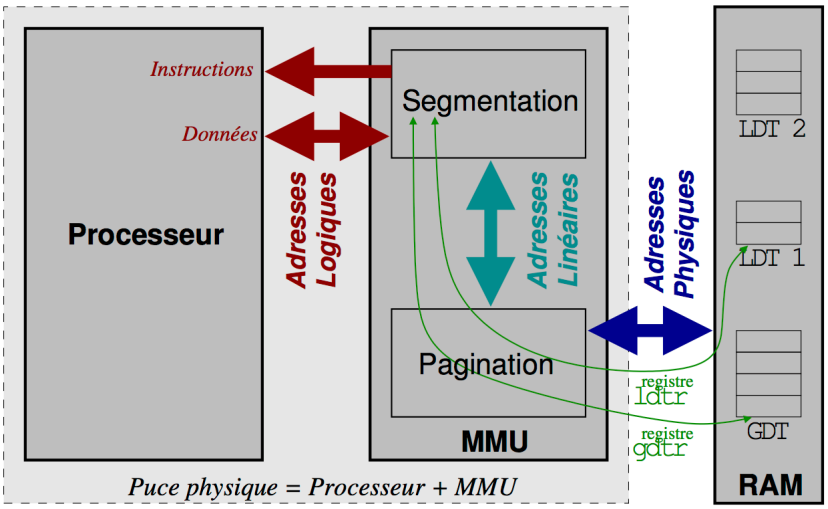

FIGURE 9 – Translation d'adresse

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	*Code Segment*
DS	*Data Segment*
SS	*Stack Segment*
ES	*Extra Segment*
FS GS	*General Purpose Segments*

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 utilisé en mode protégé (CS, DS, et SS). Les opérations adressant le code (décodage des instructons 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'integralité 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]

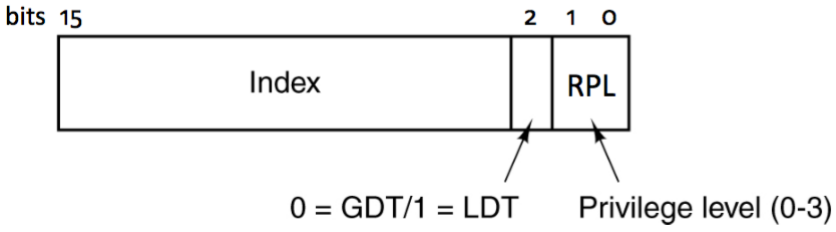

FIGURE 10 – Structure d'un sélecteur de segment

Pour récupérer l'index dans la GDT d'un segment à partir de son descripteur, il faut donc faire un décallage de 3 bits. Prennons 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 ($2 << 3 = 16$).

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 touts 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 aujoutées 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 grace au mécanisme de segmentation et c'est le mécanisme de pagination qui permet de translater cette adresse linéaire en adresse physique (comme vu précedemment 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 des pages (*Page Table*) pour chaque tâche. Les répertoires et les tables des pages ont la taille d'une page et sont composés 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]

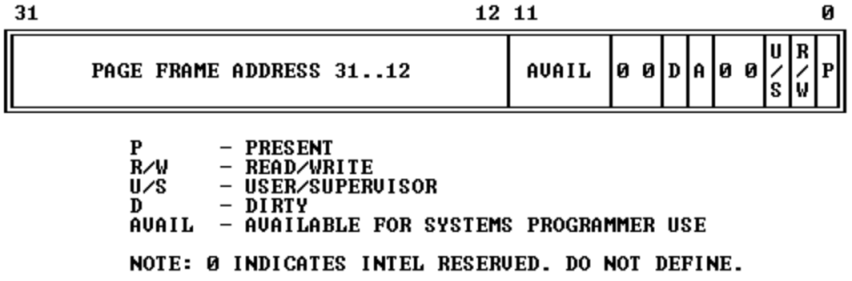

FIGURE 11 – Structure d'une *Page Entry*

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]

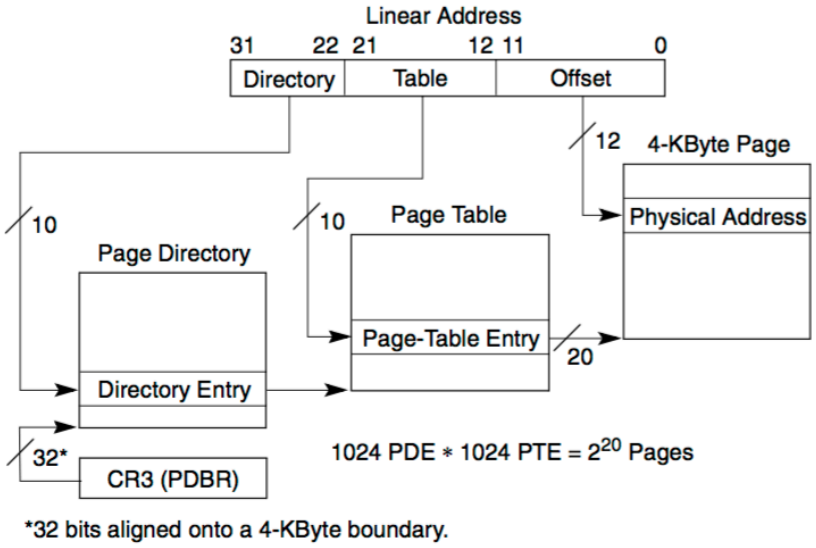

FIGURE 12 – Exemple de pagination à 3 niveaux

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. Grace 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 :

```
SECTIONS {
    /* Low memory Kernel */
    . = 0x00100000;
    .boot ALIGN(4) :        { *(.multiboot) }
    .low_text ALIGN (4K) :  { *(.low_text) }
    .low_data ALIGN (4K) :  { *(.low_data) }
    .low_bss ALIGN (4K) :   { *(.low_bss) }
    /* Higher-half Kernel */
    . += 0xC0000000;
    .stack ALIGN(4) : AT(ADDR(.stack) - 0xC0000000)     { *(.stack) }
    .text ALIGN(4K) : AT(ADDR(.text) - 0xC0000000)      { *(.text*) }
    .rodata ALIGN(4K) : AT(ADDR(.rodata) - 0xC0000000)  { *(.rodata*) }
    .data ALIGN(4K) : AT(ADDR(.data) - 0xC0000000)      { *(.data*) }
    .bss ALIGN(4K) : AT(ADDR(.bss) - 0xC0000000)        { *(COMMON) *(.bss*) }
}
```

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. Regardons plus en détail de quelle manière la première partie du *kernel* initialise la pagination. Comme dit précedemment, un répertoire de pages initial doit être construit. Etant donné que nous allons exécuter du code dans le premier Go et aussi dans le dernier, le *kernel* doit être *mappé* dans ces deux zones mémoire en même temps. La première partie va être adressée linéairement, ce qui veut dire que l'adresse physique 0x0 correspondra à l'adresse virtuelle 0x0 et ainsi de suite jusqu'à la fin du *kernel*. Cet adressage donne le répertoire de pages shématisé dans la figure 13.

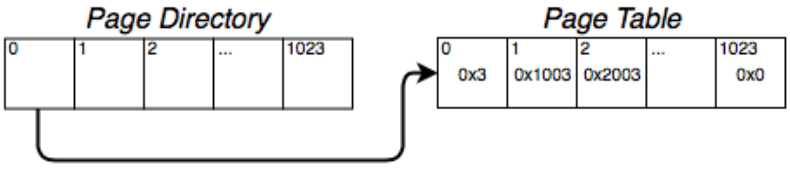

FIGURE 13 – Répertoire de pages adressant le *kernel* au début de la RAM

On peut voir ici que la première entrée du répertoire de pages pointe sur une table de pages adressant le début de la RAM. Chaque entrée est incrémentée de 4096 (0x1000 en hexadécimal) car une page fait 4096 octets. De plus, les deux premiers bits de poids faible de chaque page sont à 1 pour indiquer que la page est active et que l'on peut écrire et lire dedans (voir figure 11). L'entrée dans le répertoire de page correspondant au dernier Go (soit 0xC0000000 en hexadécimal) doit pointer sur une table des pages identique. Pour trouver une entrée dans le répertoire de pages depuis une adresse il faut faire un décallage à droite de 22 bits sur cette adresse (ce qui est équivalent à diviser par 4096, soit la taille d'une page, puis de nouveau diviser par 1024, soit le nombre de pages adressées par une table). Ici, 0xC0000000 >> 22 = 0x300 (768 en décimal). Il faut donc faire pointer l'entrée 768 du répertoire de pages à une table des pages identique à celle pointée par l'entrée 0 ce qui donne finalement le répertoire suivant.

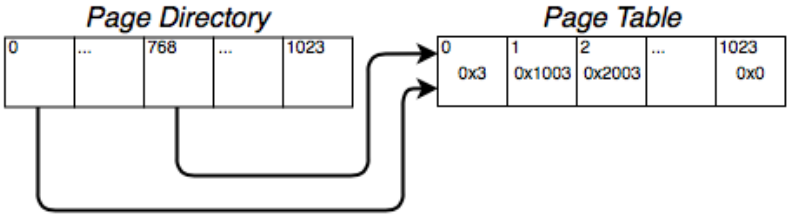

FIGURE 14 – Répertoire de pages adressant le *kernel* à la fin de la RAM

Une fois le répertoire de pages initialisé de cette manière, il ne reste plus qu'à faire pointer le registre CR3 dessus et activer la pagination en mettant le bit 31 du registre CR0 à 1. Le code peut ensuite sauter à la partie haute de la RAM où nous avons déplacé le *kernel*. A partir de là, tout le code qui sera exécuté sera dans le dernier Go de RAM, il n'y a donc plus besoin de faire pointer la première entrée du répertoire de pages sur le table des pages du *kernel* ce qui peut être fait en écrivant 0 dans cette entrée. Le code rust peut finalement être appelé avec la pagination active.

7 Periphériques

7.1 Ports

Un processeur IA-32 a la possibilité de transférer des données en utilisant les ports d'entrée/sortie. Ces ports sont utilisés par le processeurs pour communiquer avec des périphériques. Il peuvent être utilisés pour envoyer et recevoir des données (par exemple un *timer* va utiliser les ports d'entrée/sortie pour envoyer son état). Les ports peuvent aussi être utilisés pour contrôler un péripéhrique à partir de registres de contrôle (par exemple avec un controlleur de disque).[14] Etant donné que nous ne sommes pas sur du vrai *hardware*, QEMU va se charger d'émuler les différents périphériques utilisés par un processeur Intel 32-bits.

Les ports d'entrées/sorties sur architecture x86 se situent dans un espace d'adresses séparé de la mémoire physique. Cet espace permet d'adresser 64000 (soit 2^{16}) ports de 8 bits. Les ports sont donc adressés sur 16 bits mais il n'est pas possible d'écrire dans un PIO de la même manière que l'on écrirait dans la mémoire (avec une instruction `MOV`) car nous sommes dans deux espaces d'adresses différents. Ainsi, le CPU utilise des instructions speciales pour accéder aux 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]

Exemple de lecture et d'écriture dans un port d'entrée/sortie :

Ecrire 4 dans le port 0x2A :

```
mov dx, 0x2A
mov al, 4
out dx, al
```

Lire un octet depuis le port 0x2A :

```
mov dx, 0x2A
in  byte al, dx
```

Il existe une autre méthode pour écrire dans les ports utilisant le même bus d'adresse pour la mémoire physique et pour les périphériques. Cette méthode consiste à *mapper* les ports d'entrées/sorties dans la mémoire physique (MMIO). En écrivant dans la zone reservée aux ports, on écrirait alors directement dans les ports et pas dans la mémoire physique. Le *kernel* développé utilise la première méthode (PIO).

7.2 Interruptions et Exceptions

7.2.1 Principe général

Les interruptions et les exceptions sont des évenements qui indiquent que l'attention du processeur est demandée quelque part soit dans le code, soit par un périphérique. Il existe deux types d'interruptions, les interruptions logicielles et les interruptions matérielles. Les exceptions sont générées par le processeur mais diffèrent des interruptions logicielles. Une interruption peut arriver à n'importe quel moment en réponse au signal d'un périphérique ou bien si le processeur le demande avec l'instruction `INT` (interruption logicielle). Une exception est levée lorsque le processeur détecte une erreur à l'exécution d'une instruction (par exemple une division par 0). Quand une interruption ou une exception a lieu, une routine logicielle est appelée (ISR). Les processeurs IA-32 supportent jusqu'à 256 interruptions dont les 32 premières sont reservées aux exceptions processeur (voir figure 15).[1, 12]

Vector No.	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	RESERVED	Fault/ Trap	No	For Intel use only.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD2 instruction or reserved opcode.[1]
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (Zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[2]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	—	(Intel reserved. Do not use.)		No	
16	#MF	x87 FPU Floating-Point Error (Math Fault)	Fault	No	x87 FPU floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes (Zero)	Any data reference in memory.[3]
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.[4]
19	#XF	SIMD Floating-Point Exception	Fault	No	SSE and SSE2 floating-point instructions[5]
20-31	—	Intel reserved. Do not use.			
32-255	—	User Defined (Non-reserved) Interrupts	Interrupt		External interrupt or INT *n* instruction.

NOTES:
1. The UD2 instruction was introduced in the Pentium Pro processor.
2. IA-32 processors after the Intel386 processor do not generate this exception.
3. This exception was introduced in the Intel486 processor.
4. This exception was introduced in the Pentium processor and enhanced in the P6 family processors.
5. This exception was introduced in the Pentium III processor.

FIGURE 15 – Table des interruptions et exceptions sur IA-32

Comme vu précedemment, une interruption logicielle peut être exécutée par le processeur avec l'instruction `INT`. L'instruction `INT` suivie du numéro d'interruption sur 8 bits déclenchera l'interruption en question. Par exemple, l'instruction `INT 0x30` déclenchera l'interruption 48. Au moment de l'appel à l'instruction `INT`, le pointeur d'instruction va sauter à l'adresse du code contennant la routine d'interruption correspondant au numéro d'interruption logicielle specifiée. C'est la table des descripteurs d'interruption (IDT) qui permet de définir l'adresse du code à exécuter pour chaque numéro d'interruption (que ce soit une interruption logicielle, matérielle ou une exception). A noter aussi que les interruptions logicielles sont synchrone étant donné qu'elles sont exécutées par le processeur, contrairement aux interruptions matérielles qui sont asynchrones (exécutées par les périphériques, elle peuvent arriver à n'importe quel moment).

Nous avons vu que les interruptions matérielles étaient générées par le *hardware*. Il existe deux types d'interruptions matérielles, les NMI (*Non Maskable Interrupt*) et les IRQ (Interrupt Request). Une NMI indique qu'un problème est survenu au niveau matériel (mémoire défectueuse, erreur de bus, ...). Comme son nom l'indique, une NMI ne peut pas être ignorée (ou masquée), l'interruption doit donc dans tous les cas être servie. Le but ici est d'arrêter le processeur afin d'éviter toute perte de données.[1] Une IRQ quant à elle peut être masquée. L'instruction `CLI` permet de masquer les interruptions et l'instruction `STI` permet de les démasquer. En général, un périphérique génère une IRQ lorsque des données sont prêtes à être lues, qu'une commande est terminée ou qu'un évenement particulier a lieu (par exemple la pression d'une touche du clavier ou l'écriture de données sur le disque). Quand une interruption est générée, l'ISR correspondant à l'IRQ doit être appelée. C'est là qu'entre en jeu le controlleur d'interruption (PIC). Le PIC va faire correspondre une IRQ à un numéro d'interruption (voir figure 16). A la manière des interruptions logicielles l'IDT va être utilisée pour appeler la bonne routine d'interruption. Le PIC permet donc de faire le lien entre le matériel et le logiciel.

IRQ	Description	Interruption
0	System timer (PIT)	0x08
1	Keyboard	0x09
2	Redirected to slave PIC	0x0A
3	Serial port (COM2/COM4)	0x0B
4	Serial port (COM1/COM3)	0x0C
5	Sound card	0x0D
6	Floppy disk controller	0x0E
7	Parallel port	0x0F
8	Real time clock	0x70
9	Redirected to IRQ2	0x71
10	Reserved	0x72
11	Reserved	0x73
12	PS/2 mouse	0x74
13	Math coprocessor	0x75
14	Hard disk controller	0x76
15	Reserved	0x77

FIGURE 16 – Table de correspondance des IRQs

En comparant la figure 15 avec la figure 16, on constate que certaines IRQs partagent le même numéro d'interruption que des exceptions. L'interruption du *timer* par exemple a le même numéro d'interruption que l'exception *Double Fault* (0x8). Si on laisse le *mapping* par défaut, une interruption du *timer* va déclencher une *Double Fault* ce qui n'est pas souhaitable. Il a donc été nécessaire de changer cette table de correspondance. Les IRQs 0 à 7 ont été associées aux interruptions 32 à 39 et les IRQs 8 à 15 ont été associées aux interruptions 40 à 47. Ce changement de *mapping* peut se faire assez simplement en assembleur en utilisant les ports des deux PICs utilisés par les IRQs. Un code d'exemple est donné sur le site OSDev.[15]

7.2.2 IDT

La table des descripteurs d'interruption (ou IDT) est similaire à la GDT (la table des descripteurs globaux). Elle est aussi composée de descripteurs de 64-bits permettant chacun de référencer une interruption. Un descripteur est composé d'un offset indiquant l'adresse de l'ISR (la routine d'interruption), un selecteur de segment indiquant le segment où se trouve le code de l'ISR et un niveau de privilège indiquant le niveau de privilège requis pour exécuter l'ISR. Dans le cas d'un adressage de type *FLAT* comme celui utilisé, le selecteur de segment sera forcément le selecteur de segment de code. Il existe aussi plusieurs types de descripteurs d'interruptions[12] décrits dans la figure 17. Dans le cas de notre *kernel* seulement deux types ont été utilisés, le type *Interrupt Gate* et le type *Trap Gate*. La différence entre un *Interrupt Gate* et un *Trap Gate* est uniquement le comportement du CPU lors de l'exécution de l'ISR[1]. Dans le cas du *Interrupt Gate*, le CPU masquera les interruptions lors de l'exécution de l'ISR alors que dans un *Trap Gate* ce ne sera pas le cas.

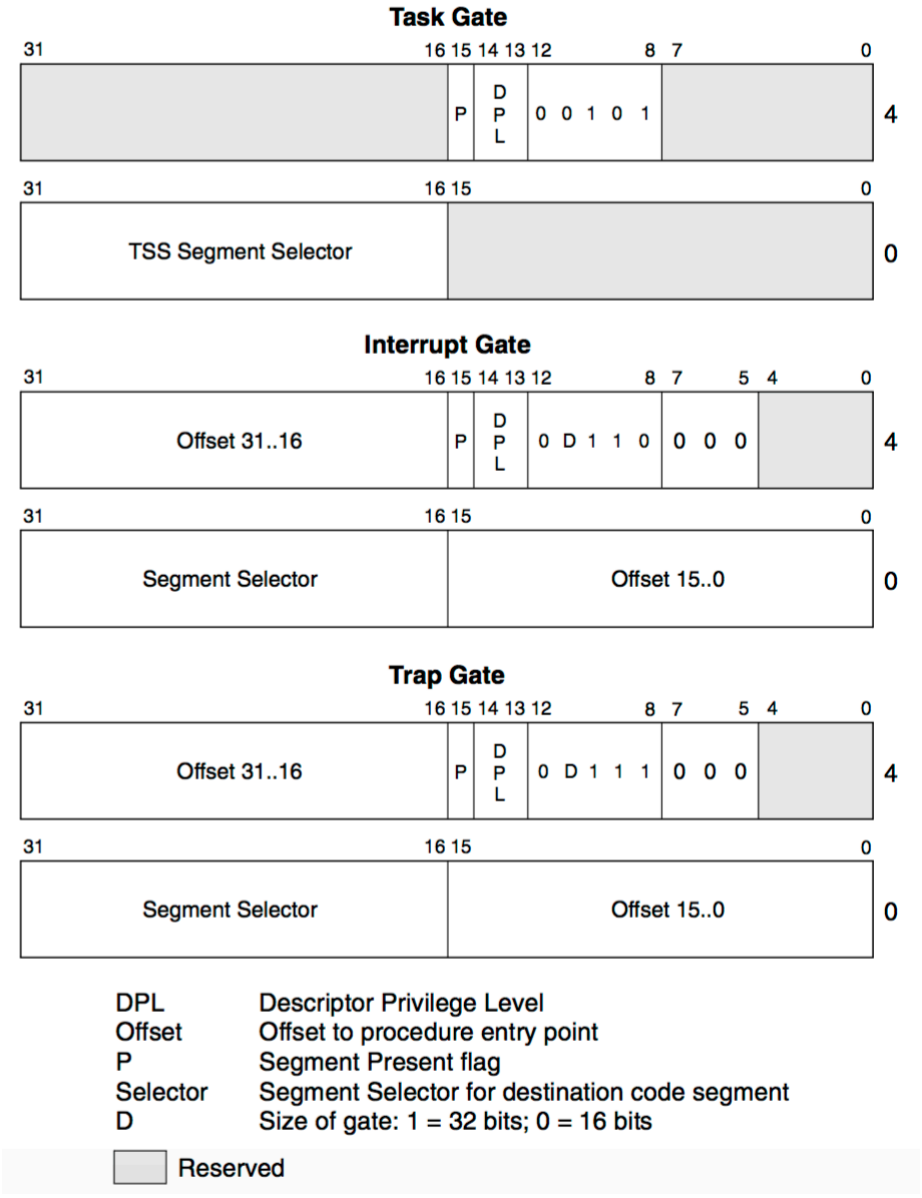

FIGURE 17 – Différents types de descripteur d'interruption

Comme pour la GDT, l'IDT est stockée en RAM et doit donc être initialisée et gérée par l'OS. De la même manière que l'instruction `LGDT` permet de charger la GDT, l'instruction `LIDT` permet de charger l'IDT dans le registre IDTR. Pour se faire il faut donner comme argument à l'instruction `LIDT` l'adresse du descripteur d'IDT sur 48 bits. Ce descripteur est composé de l'adresse de l'IDT sur 32 bits et de sa limite (sa taille en bytes - 1) sur 16 bits. Une fois la table des descripteurs d'interruption chargée avec l'instruction `LIDT`, les interruptions peuvent être activées en utilisant l'instruction `STI`. La figure 18 permet de résumer la relation entre le registre IDTR et l'IDT.

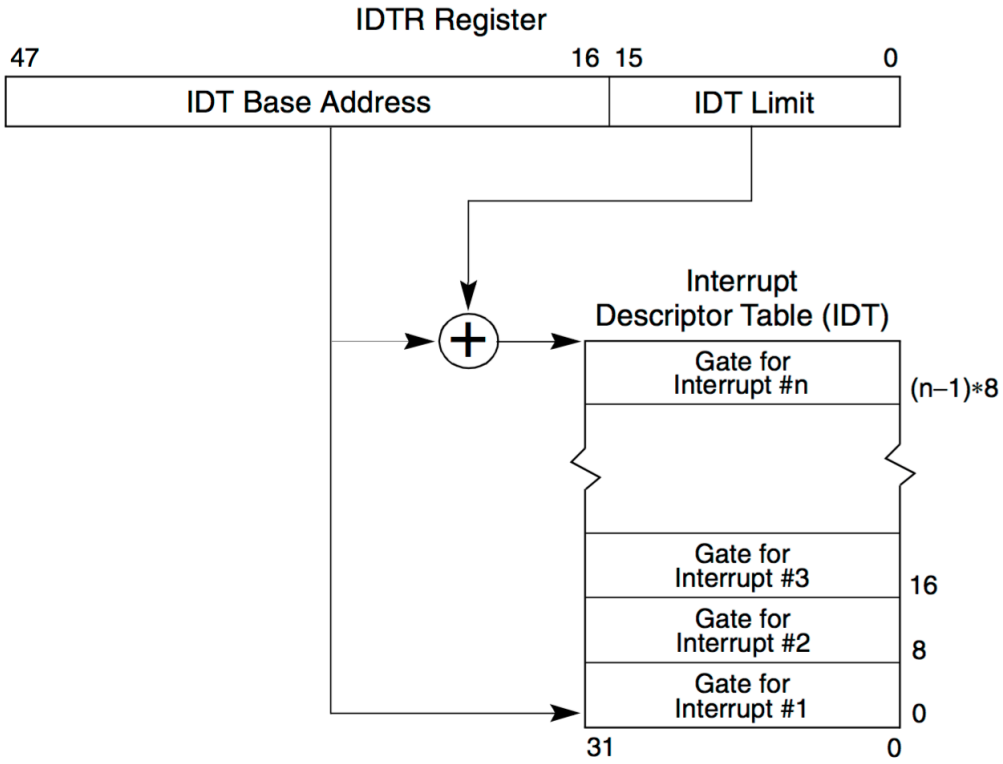

FIGURE 18 – Relation entre le registre IDTR et l'IDT

Dans le *kernel* développé, l'IDT est une structure statique en mémoire. Une fonction assembleur est donc appelée afin de charger cette structure dans le registre IDTR. En plus du chargement de l'IDT, une partie des routines d'interruptions est faite en assembleur. En effet, il a été nécéssaire de passer par du code bas niveau car avant de rentrer dans une routine d'interruption, il faut sauvegarder le contexte. Il est obligatoire de sauvegarder le contexte car, comme déjà dit plus haut, une interruption peut avoir lieu à n'importe quel moment. La partie bas niveau de la routine d'interruption s'occupe donc de sauvegarder le contexte puis d'appeler un gestionnaire d'interruption haut niveau en rust. Ce gestionnaire prend comme argument le numéro d'interruption et appelle la routine d'interruption liée à cette interruption. Par exemple, la routine d'interruption du *timer* va simplement incrémenter un compteur. Lorsque une exception est levée le même mécanisme est employé sauf qu'ici le *kernel* va afficher un message d'erreur en fonction du numéro de l'exception.

7.3 VGA

Un PC possède généralement une carte graphique permettant de gérer l'affichage. Une grande majorité des carte graphiques, même modernes sont compatibles avec le standard d'affichage VGA. Dans notre cas, nous utilisons un émulateur (QEMU) qui va émuler l'affichage VGA. Pour écrire sur l'écran il faut écrire dans la mémoire vidéo (VRAM) qui commence à l'adresse 0xA0000 et finit à l'adresse 0xBFFFF. Différents modes d'écriture existent pour l'affichage mais nous allons nous concentrer sur un seul en particulier.

Le mode texte VGA a été utilisé pour l'affichage dans l'OS développé. En mode texte, l'écran est divisé en caractères plutôt qu'en pixels ce qui permet d'afficher simplement et rapidement quelque chose sur l'écran. La mémoire vidéo reservée au mode texte commence à l'adresse 0xB8000 et a une taille de 80×25 caractères. Un caractère est représenté par 2 octets (16 bits) ce qui fait une taille de 4000 octets ($80 \times 25 \times 2$). L'octet de poids faible d'un caractère représente la valeur ASCII de ce caractère et l'octet de poids fort représente l'attribut qui contient lui même la couleur du caractère et la couleur du fond (voir figure 19) [1]. La couleur en mode texte est donc codée sur 4 bits ce qui fait 16 couleurs différentes. Ces 16 couleurs sont décrites dans la figure 20 [16].

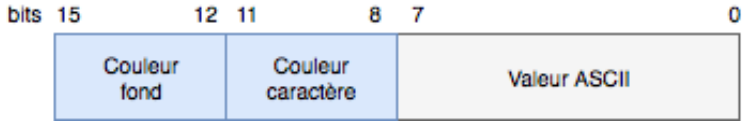

FIGURE 19 – Structure d'un caractère en mode texte VGA

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 20 – Couleurs disponibles en mode texte VGA

Le mode texte VGA permet aussi d'afficher un curseur. Le curseur ne se déplace pas automatiquement quand un caractère est écrit à l'écran, c'est simplement une zone de l'écran mise en évidence par un clignotement et dont la taille, la poisition et la visibilité peuvent être modifiés. [17] L'accès au curseur se fait en utilisant les registres du CRTC (*Cathode Ray Tube Controller*). Les registres du CRTC peuvent être accédés avec la paire registre d'adresse et registre de données. Ces regsitres se trouvent respectivement aux ports 0x3D4 et 0x3D5. L'écriture dans un registre du CRTC se fait donc en deux temps. Tout d'abord, l'adresse du registre doit être specifiée en écrivant dans le port 0x3D4 puis la donnée doit être écrite dans le port 0x3D5. [1]

7.4 *Timer*

7.5 Clavier

8 Système de fichiers

8.1 Introduction

8.2 Structure

9 Tâches utilisateur

10 Résultats

11 Discussions

11.1 Problèmes rencontrés

11.2 Améliorations possibles

12 Conclusion

13 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).

[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] Intel. Ia-32 intel architecture - software developer's manual - volume 1 : System programming guide. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf, 2016.

[15] 8259 pic. https://wiki.osdev.org/8259_PIC.

[16] Text ui. https://wiki.osdev.org/Text_UI.

[17] Crt controller registers. https://web.stanford.edu/class/cs140/projects/pintos/specs/freevga/vga/crtcreg.htm.

[18] Rust book first edition. https://doc.rust-lang.org/book/first-edition.

[19] Rust book second edition. https://doc.rust-lang.org/book/second-edition.

[20] Cargo book. https://doc.rust-lang.org/cargo.

[21] Target option. https://doc.rust-lang.org/1.1.0/rustc_back/target/struct.Target.html.

[22] Target i386 example. https://github.com/rust-lang/rust/issues/33497.

[23] ___floatundisf issue. https://users.rust-lang.org/t/kernel-modules-made-from-rust/9191.

[24] Writing an os in rust. https://os.phil-opp.com.

[25] Writing an os in rust (second edition). https://os.phil-opp.com/second-edition.

[26] Setting up paging. https://wiki.osdev.org/Setting_Up_Paging.