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

Table des figures

Remerciements

Conventions typographiques

Lors de la rédaction de ce document, les conventions typographiques 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

API *Application Programming Interface*. 39, 43

BIOS *Basic Input Output System*. 13, 14

CPU *Central Processing Unit*. 17, 27, 30, 34
CRTC *Cathode Ray Tube Controller*. 32

ELF *Executable and Linkable Format*. 9, 10, 13

FAT *File Allocation Table*. 36–38

GCC *GNU Compiler Collection*. 10, 12
GDT *Global Descriptor Table*. 16–20, 30, 31, 40–42
GRUB *GRand Unified Bootloader*. 10, 14

IA-32 *Intel Architecture, 32-bit*. 15, 27, 28, 34, 36, 40–42
IDT *Interrupt Descriptor Table*. 28–31, 40, 43
IRQ *Interrupt Request*. 29, 34, 35
ISO *International Organization for Standardization*. 10, 14, 36
ISR *Interrupt Service Routine*. 27, 29, 30, 43

LDT *Local Descriptor Table*. 16–18, 20, 40, 41

MBR *Master Boot Record*. 13, 14
MMIO *Memory Mapped Input/Output*. 27
MMU *Memory Management Unit*. 15, 16, 18, 20, 40

NMI *Non Maskable Interrupt*. 29

OS *Operating System*. 9, 12–16, 18, 20, 22, 24, 31–33, 36, 38, 40, 41, 43, 44

PC *Personal Computer*. 13, 32, 34
PIC *Programmable Interrupt Controller*. 29
PIO *Port Input/Output*. 27
PIT *Programmable Interval Timer*. 34
POSIX *Portable Operating System Interface*. 39

RAM *Random Access Memory*. 15, 22, 23, 25, 31, 36, 41, 43

TSS *Task State Segment*. 16, 41, 42

VGA *Video Graphics Array*. 32–34
VRAM *Video Random Access Memory*. 13, 32, 43

1 Introduction

1.1 Contexte

1.2 Objectif

2 Architecture globale

2.1 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.

2.2 Technologies

2.2.1 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.

2.2.2 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.

2.2.3 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

2.2.4 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.

2.3 Architecture

3 Langage Rust

4 Exécution du *kernel*

4.1 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 2.2. 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]

4.2 *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*.

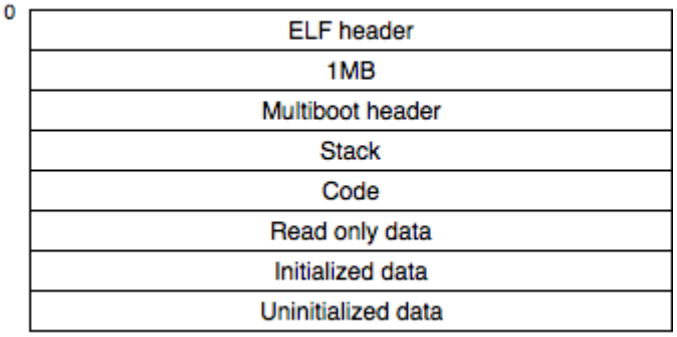

FIGURE 1 – Strucutre du fichier ELF

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.

4.3 *Boot*

4.3.1 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]

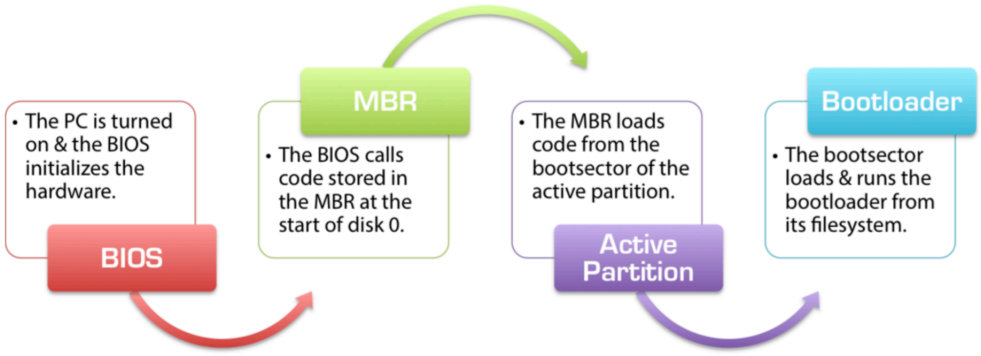

FIGURE 2 – *Boot* d'une machine à base de BIOS

4.3.2 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].

4.3.3 Image ISO

Nous avons déjà pu voir que le *boot* du *kernel* se faisait à partie d'une image ISO dans la partie 2.2.4. 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].

5 Gestion mémoire

5.1 Introduction

Le système d'exploitation développé est exécuté sur une architecture IA-32 (Intel 32-bits) aussi appelée i386. La mémoire est donc adressée sur 32 bits. $2^{32} = 4Go$, on peut en déduire que la taille totale de la mémoire adressable est de 4Go dans notre système d'exploitation. Avoir un espace adressable de 4Go ne veut pas forcément dire que la mémoire physique (RAM) est de 4Go aussi. En réalité, la taille de la RAM dépend du *hardware*. Dans notre cas le matériel est émulé par QEMU. La taille de la mémoire physique de notre OS dépend de la configuration de l'émulateur. Ces 4Go sont donc virtuels. 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].

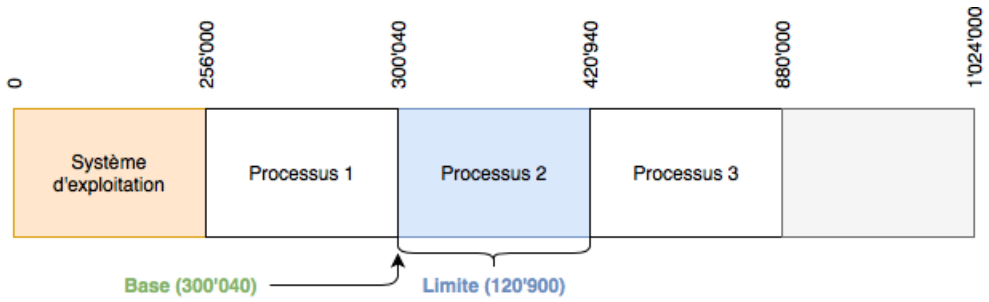

FIGURE 3 – Exemple d'adressage mémoire

Une tache possède son propre espace d'adressage dit virtuel. Pour le processus 2 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].

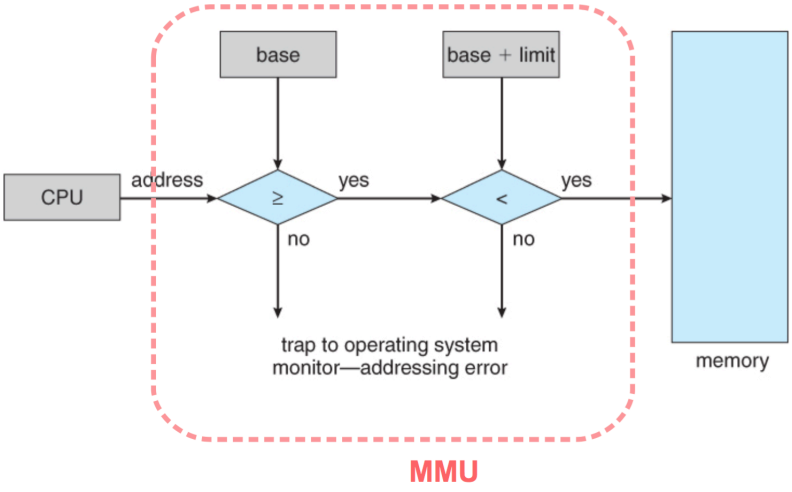

FIGURE 4 – Protection mémoire avec un MMU

Pour convertir une adresse virtuelle en adresse physique, le MMU passe par plusieurs étapes. Quand le *kernel* veut lire une donnée dans la mémoire, l'adresse de cette donnée est appelée adresse logique. Le MMU va commencer par convertir cette adresse en adresse linéaire. Une deuxième conversion est ensuite effectuée afin d'obtenir une adresse physique. Le MMU peut alors renvoyer la bonne donnée au *kernel*. Toutes ces étapes ne sont pas automatiques, le MMU utilise différentes techniques ayant besoin de certaines structures implémentées par le *kernel*. Ces techniques sont la segmentation et la pagination.

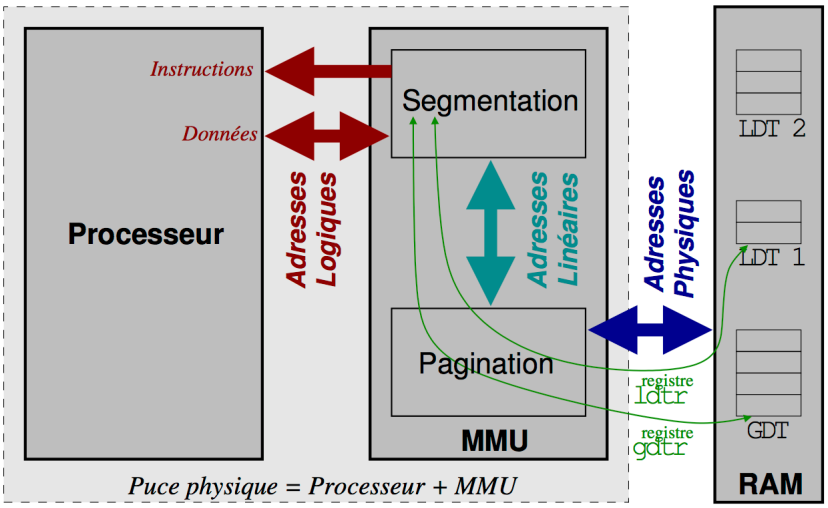

FIGURE 5 – Translation d'adresse

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 table de descripteurs globale (GDT) ou locale (LDT). Si la pagination est activée, l'adresse linéaire est convertie en adresse physique. Toute cette mécanique est décrite dans la figure 5. A noter que la pagination n'est pas obligatoire et un OS pourrait s'en passer contrairement à la segmentation qui est indispensable en mode protégé (32-bits) [7].

5.2 Segmentation

5.2.1 Principe général

Comme expliqué précedemment, la segmentation est un mécanisme divisant l'espace d'adressage du processeur en espaces d'adressage plus petits appelés des segments. Un segment peut être utilisé pour contenir le code, les données ou la pile d'un processus étant exécuté par le processeur. Un segment peut aussi être utilisé pour contenir des structures de données tel qu'une LDT ou un TSS (structure contenant des informations à propos d'une tâche). Les segments d'un système d'exploitation sont contenus dans l'espace d'adressage linéaire. Pour lire l'octet d'un segment se trouvant dans l'espace d'adressage linéaire, le MMU utilise une adresse logique. L'adresse logique est composée d'un sélecteur de segment et d'un *offset*. Le sélecteur permet de trouver le bon segment dans la mémoire linéaire et l'*offset* permet de trouver l'octet dans ce segment. Dans le cas où la segmentation est utilisée seule (sans pagination), la mémoire linéaire est *mappée* directement dans la mémoire physique [8].

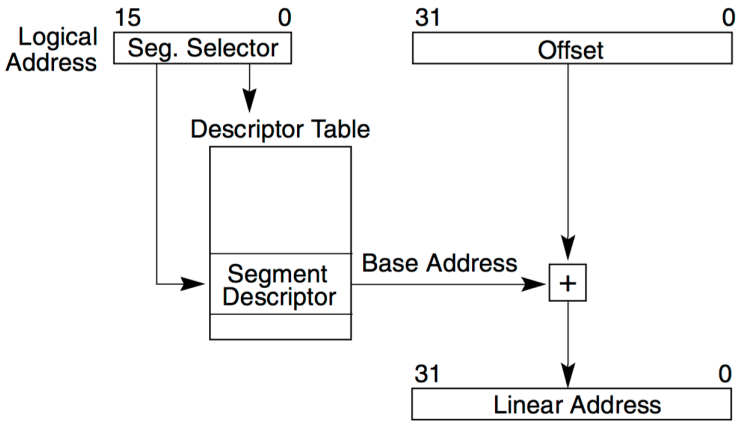

FIGURE 6 – Conversion d'une adresse logique en adresse linéaire

La figure 6 résume bien la conversion d'adresse logique en en adresse linéaire. On peut remarquer de plus que le sélecteur de segment passe par une table de descripteurs (GDT ou LDT) afin de trouver le bon segment dans la mémoire linéaire. En effet, Un sélecteur a une taille de 16 bits et contient l'index d'un descripteur dans une table, un bit indiquant si le descripteur est dans la GDT ou dans une LDT et enfin son niveau de privilège allant de 0 à 3 (figure 7) [1].

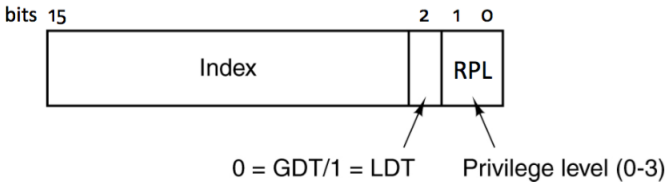

FIGURE 7 – Structure d'un sélecteur de segment

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 six et ont chacun une taille de 16 bits (même taille qu'un sélecteur de segment) [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. Ces registres pointent sur des descripteurs de segments par l'intermédiaire de sélecteurs de segment.

5.2.2 GDT et LDT

Nous avons pu voir que pour translater une adresse logique, des sélecteurs de segment sont utilisés. Ces sélecteurs pointent sur des entrées dans des tables de descripteurs. La GDT et la LDT sont deux types de table de descripteurs différents. La GDT est unique il ne peut y en avoir qu'une seule dans le système. Elle contient toutes les données utilisables en mode superviseur (*ring* 0 ou niveau de privilège 0). Une LDT est contenue dans la GDT. Elle peut être utilisée pour contenir le code et les données d'une tâche utilisateur par exemple. Dans le cas où plusieurs tâches sont exécutées en même temps, une LDT peut être créée par tâche ce qui permet en plus d'isoler le code de chaque processus. Une table de descipteurs est composée, comme son nom l'indique, de descripteurs. Chaque descripteur décrit une zone mémoire qui est défini par sa base (son adresse physique), sa limite (sa taille) et un 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 31 montre un exemple d'une GDT [1]. Etant donné que les descripteurs de la GDT ont la même structure que les descripteurs de la LDT nous allons nous concentrer sur la GDT. De plus, aucune LDT n'est utilisé dans la version actuelle de l'OS.

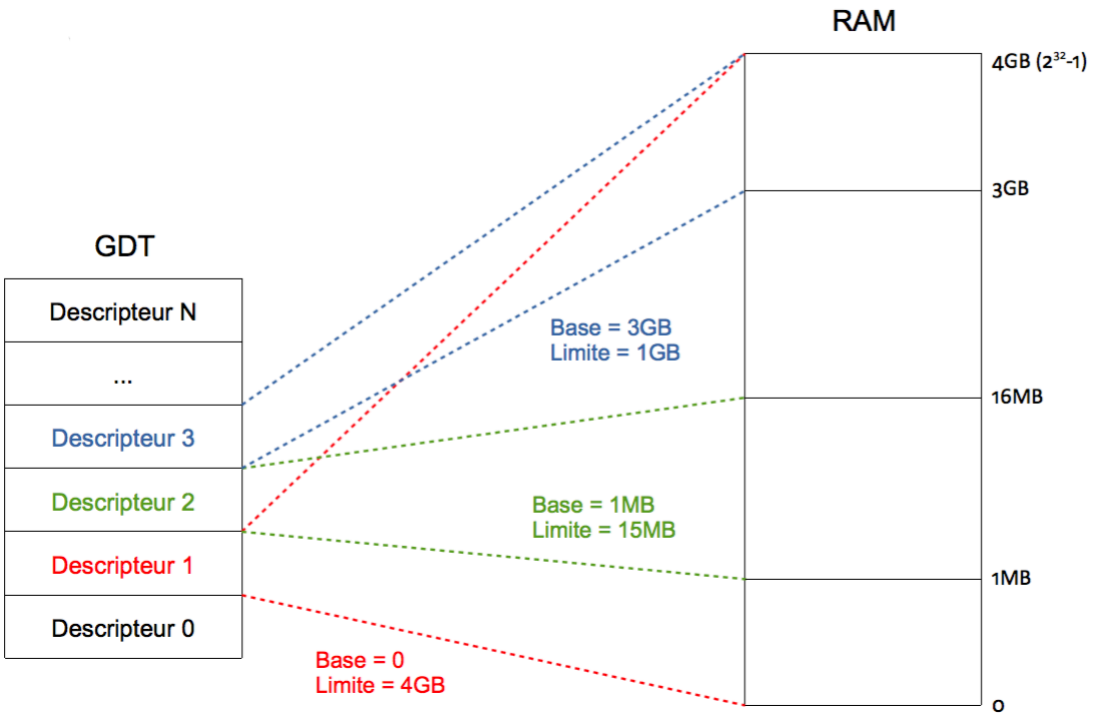

FIGURE 8 – Exemple d'une GDT

La GDT est contenue en mémoire. Cette dernière doit être initialisée par le *kernel* avant d'être chargée et utilisée par le MMU. Pour initialiser la GDT il faut construire ses entrées. Chaque entrée (ou descripteur) de la table de descripteurs est sur 64 bit et décrit une zone mémoire. L'adresse de cette zone mémoire est sur 32 bits et sa taille est sur 20 bits. Les bits restant sont des bits de contrôle pour l'accès aux données par exemple (niveau de privilèfes, droit d'écriture/lecture, ...). Voir la figure 9 pour plus de détails [1, 10].

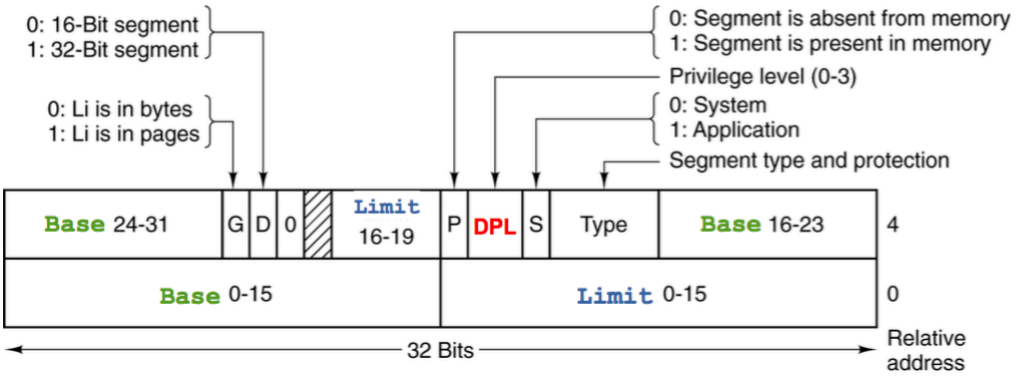

FIGURE 9 – Structure d'une entrée dans la GDT

L'OS développé a un adressage segmenté de type *flat*, c'est-à-dire que toute la mémoire est accédée de manière linéaire. Ce modèle de segmentation est le plus simple car il permet d'ignorer le mécanisme de segmentation car l'intégralité de la zone mémoire devient disponible. Dans un modèle de type *flat*, les segments de code et de données se chevauchent sur l'intégralité de la mémoire disponible. 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 ainsi les mêmes zones mémoire. On verra par la suite que d'autres entrées ont été aujoutées à la GDT pour la gestion des tâches.

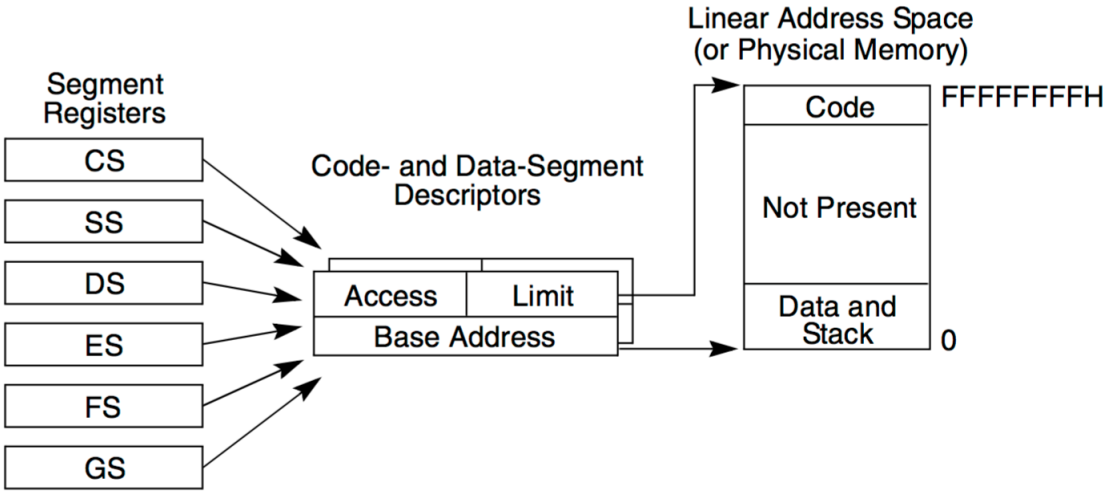

FIGURE 10 – Modèle de segmentation de type *flat*

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. Ensuite, la limite doit être mise à la valeur 0xFFFFF ce qui donne une limite réele de 0x100000 × 0x1000 soit 4Go. Le niveau de privilège doit être laissé à 0 (*ring* 0) si on veut créer un segment pour le *kernel* ou bien être mis à 3 (*ring* 3) si on veut créer un segment pour le mode utilisateur. Ci-dessous un exemple de code permettant de construire un segment de code et un segment de données.

```
fn new(base: u32, limit: u32, type: u8, s: u8, d: u8, g: u8, dpl: u8) -> GdtEntry;

pub fn make_code_segment(base: u32, limit: u32, dpl: u8) -> GdtEntry {
    GdtEntry::new(base, limit, 0xB, 0x1, 0x1, 0x1, dpl)
}

pub fn make_data_segment(base: u32, limit: u32, dpl: u8) -> GdtEntry {
    GdtEntry::new(base, limit, 0x3, 0x1, 0x1, 0x1, dpl)
}
```

Ici, `new` est le prototype d'une méthode pour une structure 64 bits représentant une entrée dans la GDT. le bit `s` est mis à 1 car on construit des segments de code et de données [8]. Le bit `d` est mis à 1 car on veut un segment de 32 bits. Le bit `g` est mis à 1 pour avoir une granularité de 4Ko. L'octet 0xB correspond au type $CODE\_EXEC\_READ$ et 0x3 correspond au type $DATA\_READ\_WRITE$ [1]. Une fois la GDT construite, il faut dans un premier temps utiliser l'instruction `lgdt` pour la charger dans le registre GDTR. Ce registre est utilisé par le processeur pour faire le lien entre le MMU et la GDT créée [8]. L'adresse du descripteur de la GDT doit donc être donnée en argument à l'instruction `lgdt`. Le descripteur de GDT est défini par la structure 48-bits décrite dans la figure 11 [10].

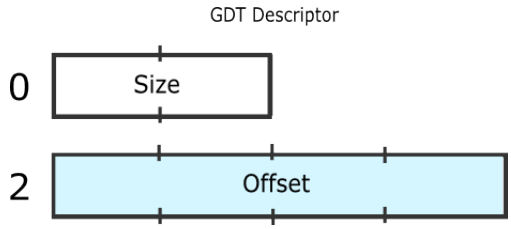

FIGURE 11 – Descripteur de GDT

Dans un descripteur de GDT *Size* est la limite sur 16 bits (c'est à dire la taille de la GDT - 1) et *Offset* est l'adresse physique de la GDT sur 32 bits. Dans le cas de notre OS, la GDT est déclarée statiquement dans le *kernel*. L'adresse de cette variable statique est utilisée pour construire un descripteur qui sera chargé dans le registre GTDR. Après avoir chargé la GDT avec l'instruction `lgdt`, il faut faire pointer les registres de segment sur les segments de la table de descripteurs à l'aide de sélecteurs. Pour rappel, un sélecteur est sur 16 bits et contient l'index d'un segment dans la GDT précedemment chargée. Pour récupérer l'index d'un segment dans la GDT à partir de l'index de son descripteur, il faut faire un décallage à gauche de 3 bits (voir figure 7). Prennons un descripteur se situant à l'index 2 de la GDT. Si on veut initialiser le segment de code (registre CS) avec ce descripteur, il faut mettre la valeur 16 dans le registre CS ($2 << 3 = 16$). Dans le cas où on veut définir ce segment avec un autre niveau de privilèges ou bien pour une LDT, il suffit de mettre les bons bits à 1 après avoir fait le décallage.

5.3 Pagination

5.3.1 Principe général

La pagination est une autres technique 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 [11]. 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 [12].

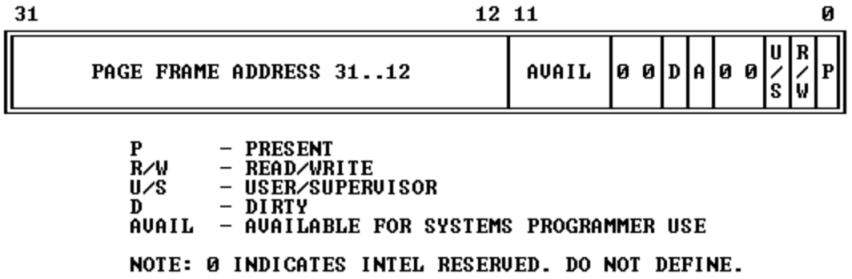

FIGURE 12 – Structure d'une *Page Entry*

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 13 résume bien ce mécanisme [8]. 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].

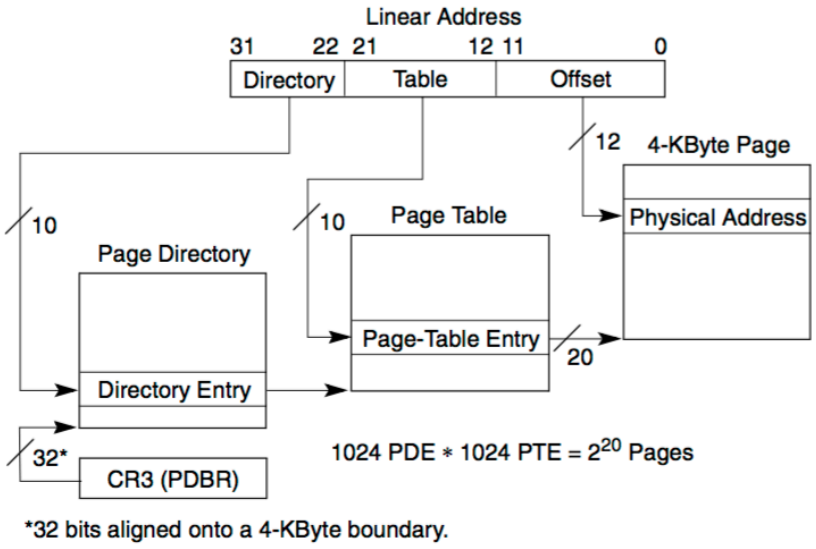

FIGURE 13 – Exemple de pagination à 3 niveaux

5.3.2 Activation de 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 :

```
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      /* Higher-half Kernel */
9      . += 0xC0000000;
10     .stack ALIGN(4) : AT(ADDR(.stack) - 0xC0000000)     { *(.stack) }
11     .text ALIGN(4K) : AT(ADDR(.text) - 0xC0000000)      { *(.text*) }
12     .rodata ALIGN(4K) : AT(ADDR(.rodata) - 0xC0000000)  { *(.rodata*) }
13     .data ALIGN(4K) : AT(ADDR(.data) - 0xC0000000)      { *(.data*) }
14     .bss ALIGN(4K) : AT(ADDR(.bss) - 0xC0000000)        { *(COMMON) *(.bss*) }
15 }
```

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 14.

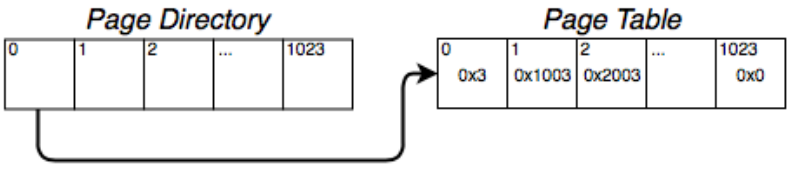

FIGURE 14 – Répertoire de pages adressant le *kernel* au début de la RAM

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 12). 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.

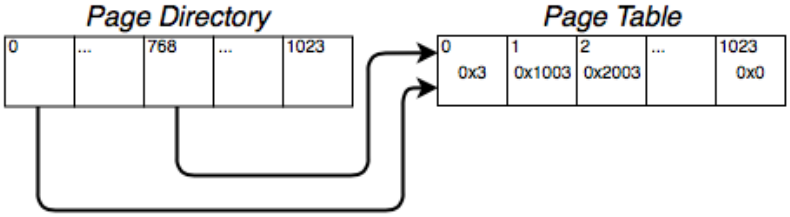

FIGURE 15 – Répertoire de pages adressant le *kernel* à la fin de la RAM

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.

5.4 Allocation dynamique en mode *kernel*

Le dernier élément de gestion mémoire implémenté dans RustOS est l'allocation de mémoire dynamique. L'allocation dynamique consiste à réserver des blocs de mémoire pendant l'exécution du *kernel* ou bien d'un programme utilisateur. Jusqu'à maintenant, toutes les structures utilisées était déclarées statiquement se retrouvant donc dans la zone mémoire du *kernel*, plus précisément dans le segment bss (voir 4.2). Déclarer des variables statiquement est pratique mais fait augmenter la taille du *kernel* et ce n'est pas une solution viable pour allouer de plus grandes régions mémoire comme par exemple du code utilisateur qui peut faire plusieurs kilooctets. Pour implémenter l'allocation de mémoire dynamique, il faut dans un premier temps définir quelle zone mémoire peut être utilisée dans ce but. Actuellement, la totalité du code et des données est contenu dans le *kernel*. Des zone mémoire peuvent donc commencer à être allouées à la fin de ce dernier. Le *linker* a été légèrement modifié afin d'obtenir l'adresse de la fin du *kernel*. Ceci peut se faire en ajoutant une expression au *linker* et en la rendant accessible depuis le code assembleur.

Modification apportée au *linker* :

```
. += 0xC0000000;
...
kernel_end = .;
```

Code assembleur permettant de rendre accessible l'expression `kernel_end` :

```
extern kernel_end
get_kernel_end:
    mov eax, kernel_end
    ret
```

A partir de là, on peut définir la zone d'allocation mémoire aussi appelée tas (ou *heap* en anglais). Dans notre OS, le tas commence à la fin du *kernel* allignée avec la taille d'une page (4096 octets). Ce choix a été fait car le *kernel* n'aura besoin d'allouer que des nouvelles pages. La fin du tas dépend de la fin de la mémoire physique qui dépend de la configuration de QEMU. Quand le *kernel* aura besoin d'allouer une nouvelle page, il ira chercher le prochain bloc libre dans le tas situé donc entre la fin du *kernel* et la fin de la mémoire physique. La recherche de bloc libre se complexifie rapidement si des blocs sont libérés. De nombreuses méthodes sont possibles pour la recherche de bloc libre et la gestion des blocs libérés. Le *kernel* développé utilisé une liste doublement chaînée pour gérer la mémoire dynamique. Chaque bloc mémoire alloué est précédé d'un entête contenant l'adresse du bloc précédent sur 32 bits, l'adresse du bloc suivant sur 32 bits, sa taille en octets sur 32 bits et un booléen indiquant si le bloc est libre ou non sur 8 bits. De plus, l'entête est aligné sur 16 octets.

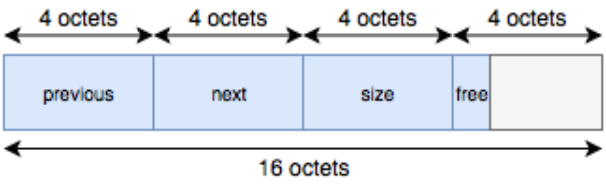

FIGURE 16 – Entête d'un bloc de mémoire dans le tas

Le tas ainsi construit, avec chaque entête lié à ses voisins par des pointeurs, permet de rechercher aisément un bloc libre. Un nouvel entête est créé quand le dernier bloc de la liste chaînée est alloué. Un algorithme a aussi été implémenter pour la gestion de mémoire libérée. Si un bloc est libéré au milieu de blocs alloués ce blocs est simplement marqué comme libre (en utilisant le booléen *free* de l'entête). Si deux blocs libres sont contiguës, ils sont fusionnés pour n'en former qu'un seul. Dans le *kernel*, la fonction pour l'allocation est `kmalloc` et la fonction pour la libération est `kfree`. La fonction `kmalloc` va allouer de nouvelles pages et tables de pages automatiquement s'il y a besoin (comme le montre la figure 17). De la même manière, la fonction `kfree` va libérer les pages et les tables de pages automatiquement. Ces deux fonctions permettent donc beaucoup d'abstraction au niveau de la pagination mais aussi de l'allocation car l'utilisation des entêtes est complètement transparent. La fonction `kmalloc` va simplement renvoyer une adresse sur 32 bits et `kfree` prend comme argument cette adresse pour libérer la mémoire.

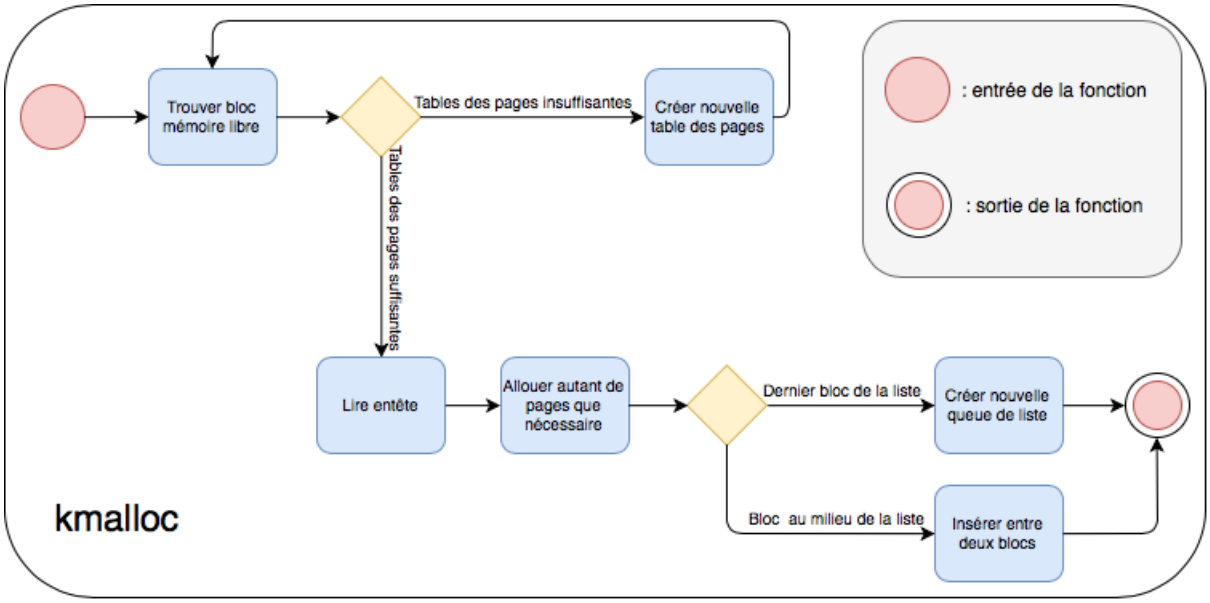

FIGURE 17 – Algorithme utilisé pour l'allocation dynamique dans le *kernel*

Voyons maintenant un exemple d'allocation et de libération mémoire utilisant les deux algorithmes décrits ci-dessus. Dans cet exemple, nous allons allouer plusieurs blocs mémoire puis les libérer afin de voir plus en détail le comportement de ces fonctions. Nous allons supposé que la fin du *kernel* se situe à l'adresse 0x3FC000 et que la fin de la RAM se situe à l'adresse 0x1000000. A l'initialisation du *kernel*, un premier entête est créé au début de la zone d'allocation dynamique. Ce premier entête est donc situé à l'adresse 0x3FC000 et a pour taille 0x1000000 − 0x3FC000 = 0xC04000. C'est cette information qui permet de trouver la premier entrée de la liste et par la suite de la parcourir.

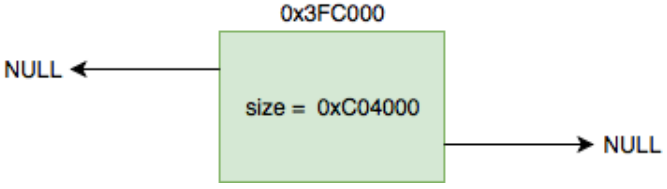

FIGURE 18 – Etat initial de la chaîne d'entêtes

Si le *kernel* a besoin d'allouer une nouvelle page maintenant, la fonction 17 va être appelée qui va parcourir les blocs libre. Dans ce cas précis, l'intégralité du tas est disponible donc la fonction va simplement créer un nouveau bloc. Dans la figure 19, les blocs libre sont en vert et les blocs alloués sont en rouge. A noter que que les blocs aloués sont alignés avec la taille d'une page (4096Ko ou 0x1000 en hexadécimal). Etant donné qu'il faut compté l'entête de 16 octets, il faudra toujours allouer une page de plus.

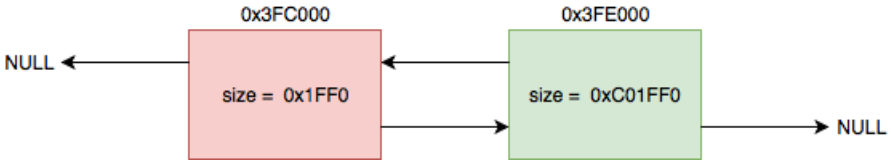

FIGURE 19 – Allocation d'une page

Si le *kernel* a de nouveau besoin d'une page, on va se retrouver dans la situation où il n'y a plus assez de place dans la table des pages. Pour rappel, une table des pages a une taille de 4Mo (0x400000 en hexadécimal). Comme expliqué avec la figure 17, la fonction d'allocation va se charge toute seule de créer une nouvelle table des pages et de l'ajouter au répertoire de pages. La page demandée par le *kernel* sera ensuite allouée. On obtient donc la liste de la figure 20, avec deux blocs alloués au lieu d'un seul. Le bloc à l'adresse 0x3FE000 contient alors la table des pages sur laquelle pointe la deuxième entrée du répertoire des pages. La page demandée par le *kernel* sera finalement à l'adresse 0x400000.

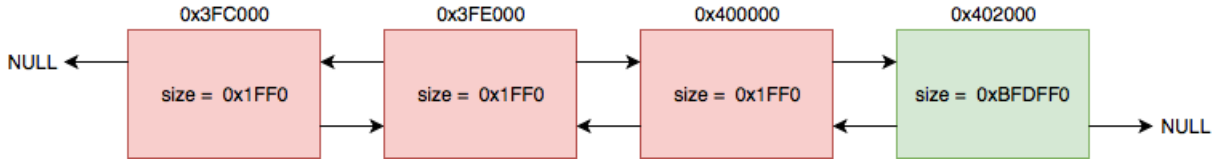

FIGURE 20 – Allocation d'une page et d'une table des pages

Libérer la page à l'adresse 0x400000 aura pour conséquence directe de libérer aussi la table des pages nouvellement créée (la fonction `kfree` libère automatiquement les tables des pages vides). On se retrouverait alors avec le même schéma que dans la figure 19 car les blocs aux adresses 0x3FE000 et 0x400000 seraient fusionnés au reste de la mémoire libre.

6 Périphériques

6.1 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).

6.2 Interruptions et Exceptions

6.2.1 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 (voir figure 21) [1, 8].

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 21 – 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 22). 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 22 – Table de correspondance des IRQs

En comparant la figure 21 avec la figure 22, 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].

6.2.2 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 [8] décrits dans la figure 23. 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.

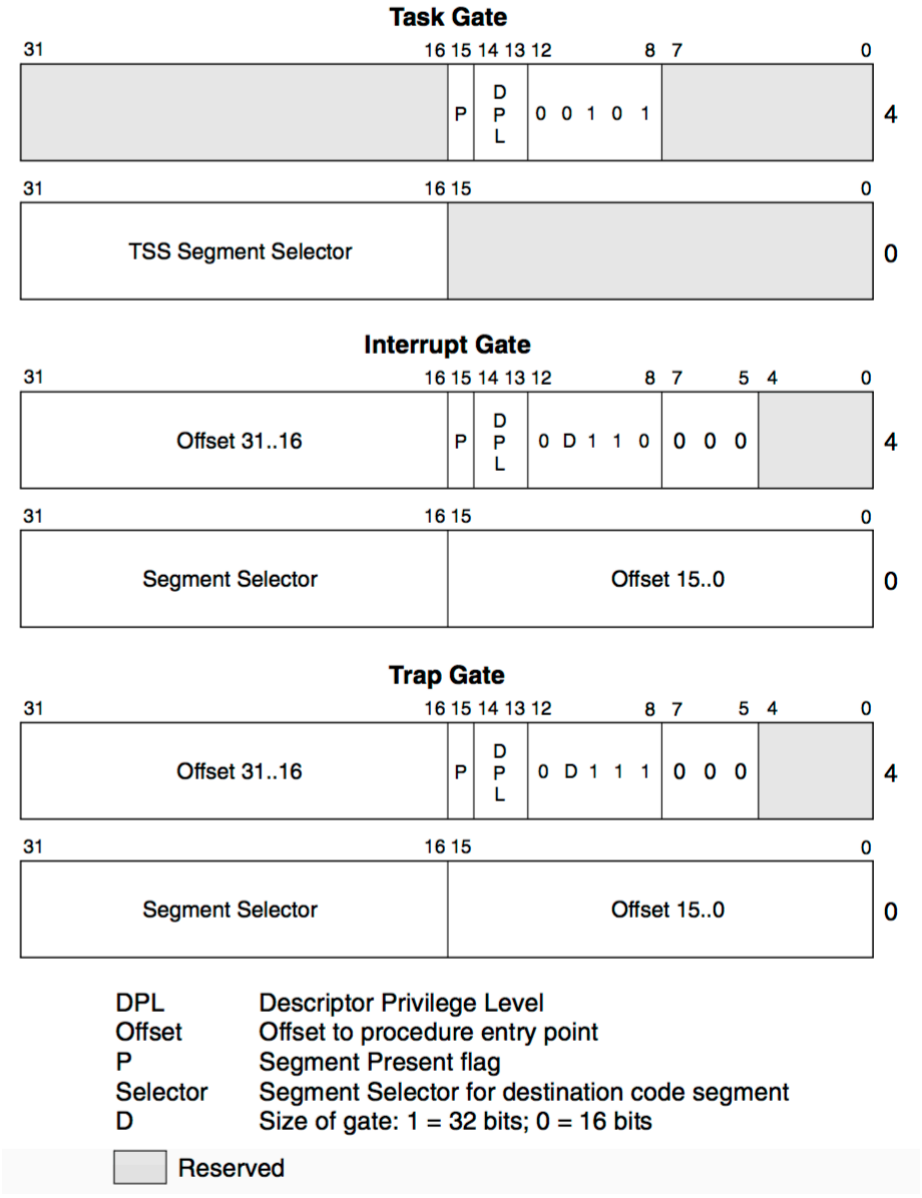

FIGURE 23 – Différents types de descripteur d'interruption

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 24 permet de résumer la relation entre le registre IDTR et l'IDT.

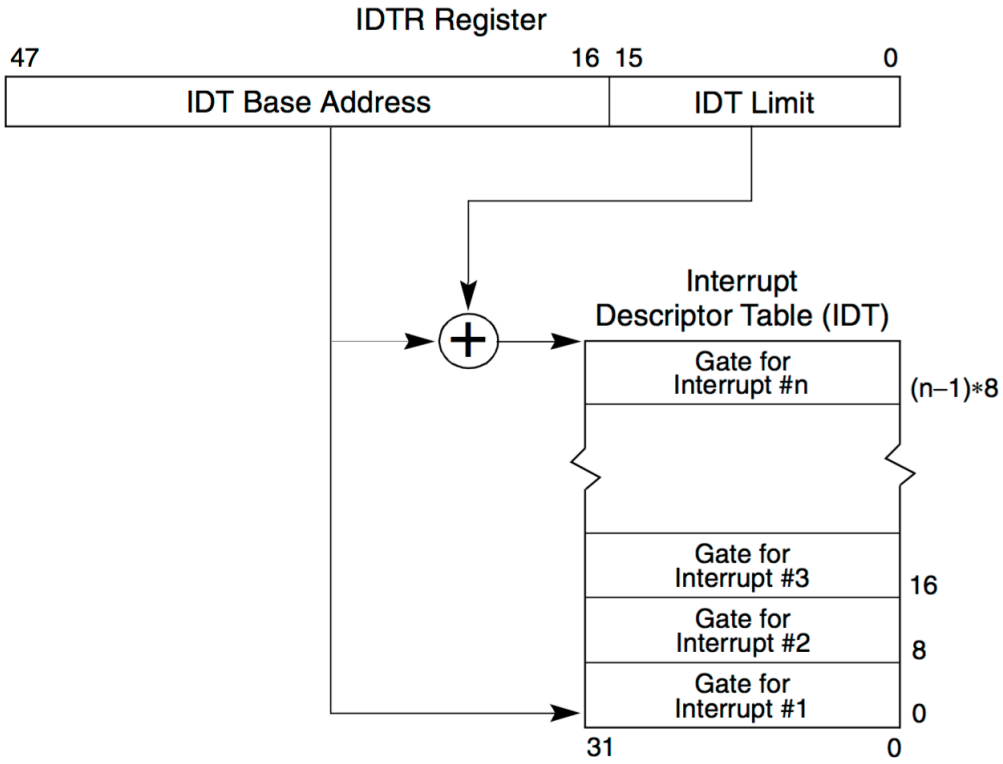

FIGURE 24 – Relation entre le registre IDTR et l'IDT

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.

6.3 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 25) [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 26 [16].

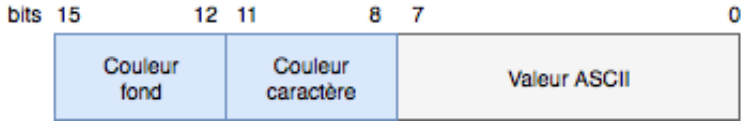

FIGURE 25 – Structure d'un caractère en mode texte VGA

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 26 – 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 position 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].

Pour l'implémentation du support VGA dans le *kernel* plusieurs structure ont été créées. Tout d'abord, la couleur est représentée par un simple `enum`. Une structure pour l'attribut a ensuite été écrite dont le constructeur prend comme argument la couleur de fond et la couleur du caractère. Après cela, une structure représentant un caractère sur 16 bits a été implémentée. Cette structure est composée du caractère sur 8 bits et de l'attribut sur 8 bits aussi. Ces structures permettent de simplifier la construction d'un caractère pour l'écrire dans le *frame buffer* (mémoire vidéo dédiée au mode texte VGA). Une dernière structure a finalement été créée comportant un *raw pointer* vers le *frame buffer* (qui est un tableau de 80×25 caractères) ainsi que les informations sur la couleur et la position du curseur. Si la pagination est active comme c'est le cas dans notre OS, il faut bien penser à mettre l'adresse virtuelle du *frame buffer* (0xB8000 étant l'adresse physique).

```
static mut SCREEN: Screen = Screen {
    buffer: 0xC00B8000 as *mut _,
    attribute: ColorAttribute::new(Color::Black, Color::White),
    cursor_x: 0,
    cursor_y: 0
};
```

La structure `Screen` est déclarée statiquement dans le *kernel* ce qui rend tout appel à une méthode `unsafe` pour rust. En plus des méthodes pour manipuler cette structure, des fonctions ont donc été implémentées afin d'interfacer l'écriture sur l'écran. Ces fonctions permettent en plus d'éviter de mettre le code en `unsafe` à chaque fois que l'on veut afficher quelque chose. En rust, les macros `print` et `println` écrivent sur la sortie standard. Etant donné que nous sommes dans une configuration *bare-metal*, nous n'avons pas de sortie standard et ces deux macros n'existent pas. Si nous étions en C, l'équivalent de ces macros aurait été la fonction `printf` et nous aurions eu à la coder entièrement. Heureusement, rust facilite grandement les choses avec le `trait Write`. Pour implémenter ce `trait` dans une structure, il faut simplement lui indiquer comment écrire une chaîne de caractères ce qui a donc été fait pour la structure `Screen`. L'implémentation de ce `trait` par une structure donne accès à cette dernière à de nombreuses méthodes mais celle qui nous intéresse ici est la méthode `write_fmt`. Cette méthode prend en paramètre une structure `Arguments`. La structure `Arguments` permet la gestion des arguments en ligne de commande ou les macros à arguments variable. La macro `print` possède un nombre d'arguments variable, nous pouvons donc implémenter cette macro ainsi que la macro `println` comme suit :

```
macro_rules! print {
    ($($arg:tt)*) => (vga_write_fmt(format_args!($($arg)*)));
}

macro_rules! println {
    () => (print!("\n"));
    ($fmt:expr) => (print!(concat!($fmt, "\n")));
    ($fmt:expr, $($arg:tt)*) => (print!(concat!($fmt, "\n"), $($arg)*));
}
```

A noter que la fonction `vga_write_fmt` est simplement une fonction interfaçant la méthode `write_fmt` appliquée a la structure statique représentant l'écran et la macro `format_args` convertie les arguments de la macro `print` en structure `Arguments`.

6.4 *Timer*

Dans chaque PC se trouve une puce pour mesure le temps et implémenter des *timers* [1]. Cette puce est le *Programmable Interval Timer* (PIT). Sur architecture IA-32, le PIT est généralement un Intel 8253 et dans notre cas, c'est celui émulé par QEMU. Le PIT génère une interruption matérielle (IRQ0) à une fréquence sélectionnable entre 18.2065 Hz et 1.19318 MHz. L'horloge du PIT oscille à 1.19318 MHz. La fréquence de sortie est modulée à l'aide d'un diviseur configurable par le *kernel*. Ce diviseur est une valeur sur 16 bits. Sa valeur maximale est donc 65536 ce qui explique la valeur minimale de la fréquence du PIT ($\frac{1193180}{65536} \simeq 18.2065$). Le PIT possède aussi plusieurs canaux avec chacun un diviseur propre mais seulement le premier canal (canal 0) est utilisé dans le *kernel* est c'est sur celui-ci que nous allons nous concentrer. Le PIT est programmable à l'aide de différents ports. Le port 0x43 contient le registre de commande du PIT et le port 0x40 permet de configurer le canal 0. Ci-dessous, le code permettant de programmer le *timer* à une fréquence `freq`.

```
let div = 1193180 / freq;
outb(0x43, 0x36);
outb(0x40, (div & 0xFF) as u8);
outb(0x40, (div >> 0x8) as u8);
```

Ecrire 0x36 dans le registre de commande du PIT indique la sélection du diviseur et le mode répétition (réinitialisation du compteur une fois celui-ci arrivé à 0) [1]. L'octet de poids faible du diviseur est ensuite écrit dans le port du canal 0 suivi par l'octet de poids fort. Dans le *kernel* développé, le *timer* est représenté par une structure statique contenant la fréquence du timer sur 32 bits et l'état actuel de son compteur sur 32 bits aussi. Ce compteur est incrémenté à chaque fois qu'une interruption à lieu sur l'IRQ0.

```
static mut TIMER: Timer = Timer { freq: 0, ticks: 0 };
```

Comme pour la structure statique permettant d'interfacer l'affichage texte VGA, toute modification de la variable `TIMER` sera **`unsafe`** au niveau du compilateur rust. Des fonctions ont donc été implémentées pour lire ou écrire le *timer*. Actuellement, le PIT est utilisé seulement pour arrêter le *kernel* pendant un temps donné avec la fonction `sleep`. Cette fonction prend une durée en millisecondes et rentre dans une boucle (attente active).

```
pub fn sleep(ms: u32) {
    let duration = get_ticks() + (ms * get_freq() / 1000);
    loop {
        if get_ticks() >= duration {
            break;
        }
        halt();
    }
}
```

Ici, `get_ticks` et `get_freq` sont des fonctions qui renvoient les attributs `ticks` et `freq` de la structure statique `TIMER`. La fonction `halt` appelle l'instruction assembleur `HLT` qui arrête le processeur jusqu'à ce qu'une interruption matérielle soit déclenchée. Elle est appelée après chaque comparaison pour éviter d'utiliser 100% du CPU lors de l'attente sur un `sleep`. On pourrait imaginer d'autres utilisations du *timer* comme par exemple la génération de nombres pseudo-aléatoires.

6.5 Clavier

Afin d'avoir un système d'exploitation un minimum complet, il est impératif de faire un support pour le clavier. Sur architectur Intel 32-bits, chaque pression ou relâchement d'une touche du clavier déclenche une interruption matérielle (IRQ1) [1]. Le périphérique clavier possède un *buffer* interne qui va stocker les données à chaque interruption. Ces données sont appelées *scan codes*. Chaque *scan code* correspond à une touche du clavier et le bit de poids fort du *scan code* indique si la touche a été pressée (*make code*) ou relâchée (*break code*). La valeur de la touche est stockée dans les sept bits de poids faible mais n'est pas à confondre avec un code ASCII (figure 27). Il faut donc faire correspondre la valeur chaque touche avec une valeur ASCII ce qui a été fait avec une table de correspondance dans le *kernel*.

FIGURE 27 – Structure d'un *scan code*

Le clavier comporte deux registres. Un registre de données et un registre d'état. Ces registres sont accessibles par les ports 0x60 et 0x64. Lors d'une interruption, le *scan code* de la touche est stockée dans le registre de données. Avant de lire une donnée venant du clavier, il est nécessaire de s'assurer que le *buffer* du clavier ne soit pas vide en en venant lire le registre d'état. Normalement si une interruption a eu lieu, cela veut dire que le *buffer* est plein mais il est quand même important de vérifier pour éviter toute source d'erreur. Une donnée est prête à être lue si le bit 0 (bit de poids faible) du registre d'état est à 1. Ci-dessous, un exemple de routine d'interruption stockant la valeur ascii du *scan code* dans un *buffer*.

```
pub fn keyboard_handler() {
    let state = inb(0x64) & 1;
    if state == 1 {
        let key = inb(0x60);
        if key >> 7 == 0 {
            buffer_write(KEY_MAP[key as usize])
        }
    }
}
```

Au niveau de l'implémentation logicielle, un *buffer* circulaire a été utilisé pour la gestion du clavier. Un *buffer* circulaire est un *buffer* de taille fixe dont la fin est connectée au début. Ce type de structure de donnée est bien adapté aux flux constants de données comme la gestion d'un clavier. A chaque fois qu'une touche va être lue par la routine d'interruption du clavier, cette dernière va être stockée dans le *buffer*. La case du *buffer* contenant cette donnée est considérée comme libre si elle a été lue. Le *buffer* circulaire va donc permettre de gérer un nombre d'écritures supérieur au nombre de lectures.

7 Système de fichiers

7.1 Introduction

Dans le chapitre 6 qui traite le sujet des périphériques, nous avons vu très brièvement qu'un processeur IA-32 peut avoir un controlleur de disque dur. Actuellement, tout le *kernel* et ses dépendences sont stockés dans la RAM. Si on veut plus tard exécuter des programmes utilisateur ou lire et écrire des fichiers texte, il est nécessaire d'avoir un disque dur. Un disque dur permet de stocker des fichiers qui pourront être lus par l'OS si besoin. Pour rendre possible la gestion de plusieurs fichiers dans un disque dur, ce dernier doit contenir un système de fichiers. Un système de fichiers va organiser les fichiers ajoutés au disque dur d'une manière bien précise afin de les retrouver rapidement. Pour rappel, notre machine est émulée par QEMU. Comme pour tous les autres périphériques gérés par notre OS, QEMU émule aussi un disque dur. Ce disque dur est vierge, il faut donc lui mettre un système de fichiers. Une option de QEMU permet de mettre un système de fichiers dans le disque dur émulé à partir d'un fichier image. Ci-dessous la modification apportée à l'exécution du *kernel*.

```
$(QEMU) -cdrom $(ISO) -hda $(FS)
```

L'option `-cdrom` spécifie l'image ISO contenant le *kernel* (voir partie 4.3.3) et l'option `-hda` indique que le fichier image contenant le système de fichiers sera chargé dans le premier disque dur. La gestion d'un système de fichiers par le *kernel* s'est donc déroulé en deux étapes. Il a d'abord fallut créer un système de fichiers simple puis implémenter les *drivers* au niveau du *kernel* pour le lire. Le système de fichiers utilisé par notre *kernel* est inspiré de FAT.

Un système de fichiers FAT est divisé en blocs de même taille. Ces blocs, aussi appelés *clusters* sont eux-mêmes divisés en secteurs dont la taille est fixe (512 octets). A noter qu'un *cluster* peut avoir la même taille en octets qu'un secteur. Le système de fichiers gère une table indiquant si un *cluster* est alloué. Cette table, appelée table d'allocation de fichiers (ou *file allocation table* en anglais) est une carte où chaque entrée représente un *cluster*. Une entrée dans la table d'allocations a une taille différente en fonction du type de FAT. La taille maximale du système de fichiers en nombre de *clusters* dépend de la taille de cette entrée en bits. Si par exemple une entrée dans la table d'allocation fait 12 bits, le système de fichiers pourra allouer un maximum de 4096 *clusters* ($2^{12} = 4096$). Il existe plusieurs versions du système de fichiers FAT qui se différencient par la taille de leurs entrées dans la table d'allocation. FAT12 a des entrées de 12 bits, FAT16 des entrées de 16 bits et FAT32 des entrées de 32 bits. La FAT utilisée par notre *kernel* a des entrées de huit bits, elle serait donc une FAT8.

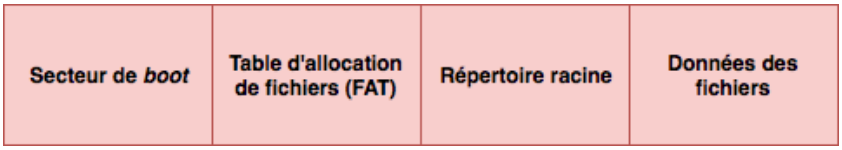

FIGURE 28 – Structure d'un système de fichiers de type FAT

La figure 28 montre de manière simplifiée la structure d'un système de fichiers FAT. Le système de fichiers réalisé s'inspire beaucoup de cette structure.

7.2 Structure

Notre système de fichiers a une structure similaire à FAT mais en plus simplifiée. Dans FAT, les 512 premiers octets sont reservés au secteur de *boot*. Ce secteur contient toutes les informations sur le système de fichiers comme la taille d'un secteur, la taille d'un *cluster*, le type de FAT (FAT12/16/32), etc. Dans notre système de fichiers, l'équivalent est le *superblock*. Sa taille est aussi de 512 octets mais actuellement seulement 23 octets sont utilisés. Ci-dessous, les détails sur le *superblock*.

Position (octets)	Taille (octets)	Nom	Description
11	2	`sector_size`	Taille d'un secteur en octets
13	1	`block_size`	Taille d'un bloc en secteurs
36	4	`fat_size`	Taille de la table d'allocation en blocs
42	2	`version`	Version du système
44	4	`root_entry`	Indice du bloc contenant les métadonnées
82	8	`label`	Nom du système de fichiers
510	2	`signature`	Signature (0x55aa)

La table d'allocation de fichiers vient directement après le *superblock*. Comme expliqué dans la partie précédente, son fonctionnement est similaire en tout point à celui de FAT à la différence que les entrées ont une taille fixe de huit bits. De plus, dans un système de fichiers FAT, il peut y avoir plusieurs tables d'allocation contrairement à notre système qui en aura toujours une seule. Vient ensuite une zone alignée sur la taille d'un bloc (ou *cluster*) contenant toutes les métadonnées des fichiers. Cet espace est aligné car il faut qu'il commence au début d'un bloc afin de le retrouver à partir du *superblock*. La taille de cet espace est d'un bloc et ne peut pour le moment pas être agrandi (contrairement à FAT où un nouveau *cluster* est alloué au besoin). Chaque entrée dans cet espace est une structure de 32 octets contenant notamment l'indice du premier bloc de données du fichier (voir le tableau ci-dessous pour plus de détails sur la structure d'une entrée).

Taille (octets)	Nom	Description
26	`name`	Nom du fichier
2	`start`	Indice du premier bloc de données
4	`size`	Taille du fichier

Les blocs de données sont situés juste après le bloc de métadonnées. A noter de plus que notre système de fichiers n'est pas hiérarchique. Tous les fichiers sont ajoutés à la racine du système (pas de répertoires). A partir de ces structures, ont peut facilement accéder à un fichier dans le disque. Les informations contenus dans le *superblock* permettent de localiser le bloc de métadonnées. Le fichier demandé est trouvé en parcourant ce bloc et son premier bloc de données peut ainsi être lu. Il reste toujours un problème si le fichier a une taille supérieure à la taille d'un bloc. C'est là que la table d'allocation peut être utilisée. En effet, l'entrée de de la table d'allocation correspondant au bloc de données peut soit pointer sur le bloc suivant, soit indiquer que ce bloc est le dernier de la chaîne.

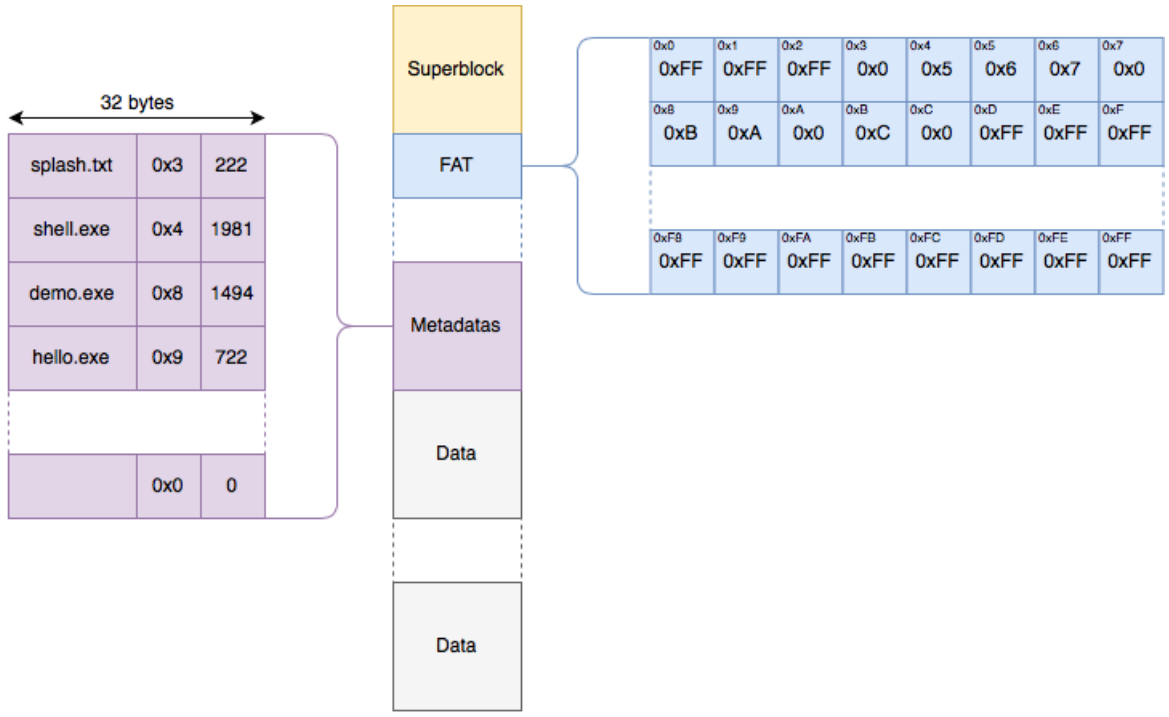

FIGURE 29 – Système de fichiers de l'OS

La figure 29 donne un exemple de comment pourrait fonctionner le système de fichiers développé. Dans cet exemple, la taille d'un bloc est la même que la taille d'un secteur. Le système de fichiers contient quatre fichiers. Le premier, `splash.txt`, rentre dans un seul bloc. La chaîne d'allocation s'arrête dès la première entrée avec un 0x0 écrit dans l'entrée 0x3 de la table d'allocation. Prenons maintenant le deuxième fichier, `shell.exe`. Ce fichier a besoin de quatre blocs pour être stocké. Dans la table d'allocation, on voit que le premier bloc de ce fichier qui d'indice 0x4 pointe sur le bloc d'indice 0x5. En suivant ainsi la chaîne d'allocation, on peut retrouver l'intégralité des blocs de données à partir de leurs indices. Cet exemple montre aussi que la FAT permet de stocker les fichiers de manière discontinue. Le fichier `demo.exe` commence au bloc d'indice 0x8 mais son deuxième bloc n'est pas à l'indice 0x9 mais à l'indice 0xB.

7.3 Implémentation

La construction du système de fichiers se fait à l'aide d'un outil externe. Cet outil a été développé en utilisant la version standard de rust. Il s'exécute par conséquent depuis la machine hôte et le gestionnaire de paquet de rust, cargo, peut être utilisé normalement. L'outil développé possède deux modes de fonctionnement. Soit à l'aide d'un menu soit directement en ligne de commande en spécifiant certaines options. Les deux modes de fonctionnement font exactement la même chose soit manipuler un fichier image donné. Cet outil permet de créer un système de fichiers vide, d'ajouter et de supprimer un fichier, de lister l'intégralité des fichiers contenus dans le système de fichiers et enfin d'afficher les informations du *superblock*. La gestion des arguments en ligne de commande se fait à l'aide du paquet `clap` installé avec cargo. Le menu avec les différentes options décrites se présente comme sur la figure 30. Toutes les options de ce menu peuvent aussi être appelées directement en ligne de commande (sauf l'option *save* qui sauvegarde les modifications apportées au système de fichiers).

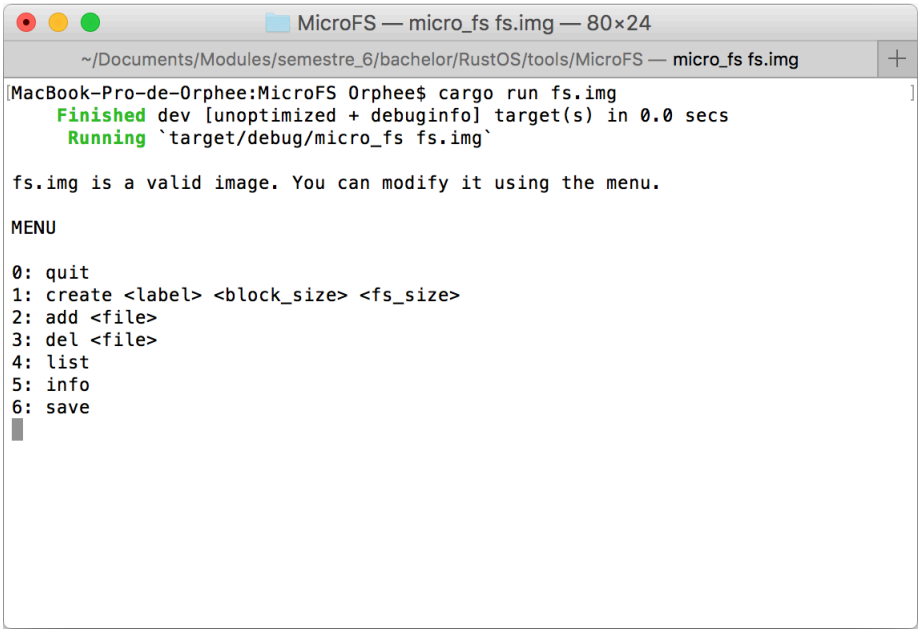

FIGURE 30 – Menu du gestionnaire du système de fichiers

Ce menu est finalement très peu utilisé car nous avons besoin de créer le système de fichiers en ligne de commande, avec le même `makefile` utilisé pour la compilation du *kernel*. Il peut quand même être pratique pour faire des tests rapides ou avoir un retour plus visuel de la modification du système de fichiers. Nous avons vu comment le système de fichiers était construit dans un fichier image et aussi que ce fichier image pouvait être donné à QEMU afin d'être chargé dans le disque dur virtuel. Il reste encore à lire dans le système de fichiers depuis le *kernel*. Pour lire dans le système de fichiers, il faut utiliser les ports du controlleur de disque dur. QEMU peut émuler plusieurs disques. Nous utilisons le premier dont les registres de contrôle sont aux ports 0x1f0 à 0x1f7. La lecture et l'écriture de secteurs (512 octets) du disque dur sont fait par l'intermédiaire de ces registres. Un code d'exemple peut être trouvé sur le site OSDev [18]. Deux fonctions ont été implémentées pour communiquer avec le controlleur de disque dont les prototypes sont ci-dessous.

```
pub fn read_sector(sector: u32, dst: *mut u16);
pub fn write_sector(sector: u32, src: *mut u16);
```

Le système de fichiers actuel ne permet que la lecture, pas l'écriture. Seule la première fonction est donc utilisée par le *kernel*. Ce dernier peut maintenant lire les secteurs du système de fichiers ce qui rend possible la réalisaton d'une API de gestion des fichiers contenus dans le disque dur. Cette API s'inspire de la sémantique POSIX pour l'accès aux fichiers (`open`, `read`, `close`, etc). Le *kernel* maintient une table de descripteurs de fichiers ouverts. Un descripteur de fichier est composé des métadonnées du fichier (structure `Stat`) ainsi que la position dans le flux d'octets du fichier. Quand un fichier est ouvert avec la fonction `file_open`, un descripteur de fichier sur huit bits est renvoyé. Ce descripteur est en fait un indice dans la table des descripteurs. Les fonctions `file_read`, `file_close` et `file_seek` manipulent ce descripteur de fichier. Des fonctions et structures pour itérer les fichiers du système de fichiers ont aussi été implémentées dans cette API.

8 Tâches utilisateur

8.1 Introduction

Pour rappel, notre OS est exécuté sur architecture Intel 32-bits (IA-32) en mode protégé. Dans cette architecture, quatre niveaux de privilèges existent. Nous avons déjà fait référence aux niveaux de privilèges (ou *ring*) dans ce document quand les différentes tables de descripteurs ont été décrites (chapitres 5.2.2 sur la GDT et la LDT et 6.2.2 sur l'IDT). Les niveaux de privilèges vont de 0 à 3. Le *ring* 0 (*kernel*) a le plus de privilèges et peut accéder à tout le jeu d'instructions du processeur alors que le *ring* 3 (*user*) en a le moins et a accès à un jeu d'instructions restreint [1].

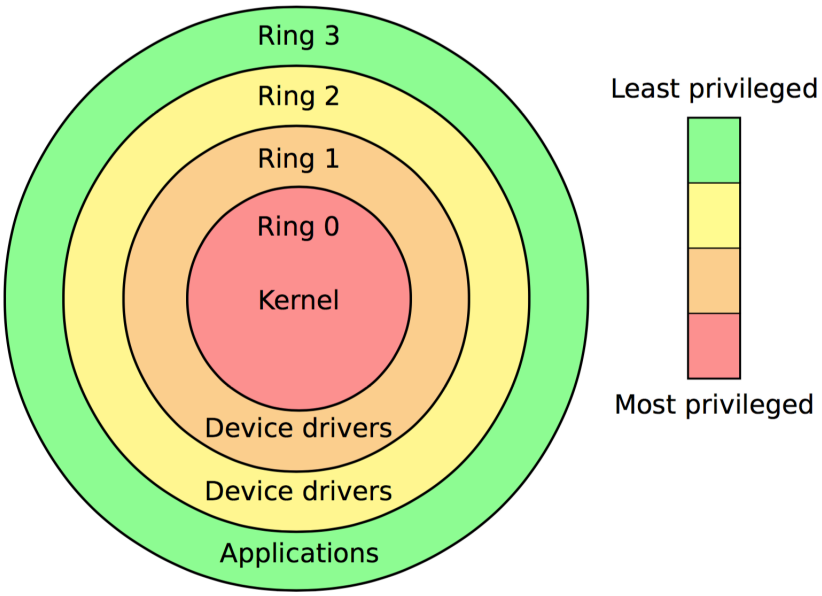

FIGURE 31 – Niveaux de privilèges sur architecture IA-32

Le *ring* 0 est aussi appelé mode privilégié. Ce mode peut accéder aux régions privilégiées de la mémoire (définies lors de l'initialisation de la GDT ou de la pagination). Seulement ce mode peut contrôler le MMU, accéder aux périphériques, définir les vecteurs d'interruptions ou encore arrêter le processeur. L'OS démarre en mode privilégié car le *kernek* doit pouvoir accéder au matériel sans aucune restriction. Si ce n'était pas le cas toutes les configurations de la mémoire et des périphériques décrites dans ce document auraient été impossibles. En revanche, Les applications utilisateur s'exécutent en mode utilisateur (*ring* 3). Si une application utilisateur démarrait avec le niveau de privilèges maximal, elle pourrait écrire dans les régions mémoires du *kernel*, modifier la configuration de la mémoire (GDT ou répertoire de pages) ou encore changer l'IDT. Il est donc impératif d'exécuter les applications en *ring* 3. Grâce à ce mécanisme de protection, le *kernel* peut être complètement isolé des applications. Le rôle du *kernel* est dans un premier temps d'attribuer et de gérer l'espace mémoire de chaque application. Il doit ensuite programmer correctement le processeur pour isoler les tâches exécutées et assurer le bon fonctionnement du système.

8.2 Exécution d'une tâche

8.2.1 Structure d'une tâche

L'architecture IA-32 implémente la gestion des tâches au niveau matériel. Une tâche doit être constituée d'un espace d'exécution et d'une structure TSS. L'espace d'exécution est constitué d'un segment de code, d'un segment de données et d'un segment de pile. La manière dont ces segments sont définis dépend de la méthode de gestion mémoire utilisée. Si la pagination n'est pas utilisée, ces segment seront définis par une LDT. Avant d'implémenter la pagination, cette méthode était utilisée par notre *kernel* pour gérer l'espace d'adressage d'une tâche. Avec la pagination, deux segments en *ring* 3 sur tout l'espace d'adressage suffisent (un segment de code et un segment de données). Ces segments sont identiques à ceux déjà construits pour le *kernel* et décrits dans la partie 5.2.2 à la différence qu'ils n'ont pas le même niveau de privilèges. Ainsi, une tâche a théoriquement un espace d'exécution de 4Go (taille de la RAM). En réalité, l'espace d'exécution de la tâche est défini par un répertoire de pages (différent de celui du *kernel*) où seront allouées autant de pages qu'il faut pour contenir l'application utilisateur. C'est la structure TSS qui spécifie les segments définissant l'espace d'exécution. Si la pagination n'est pas utilisée, c'est dans cette structure que le sélecteur vers la LDT doit être spécifié. Dans le cas contraire, le TSS contient aussi un champs `cr3` pour définir le répertoire de pages de la tâche (voir figure 32) [8].

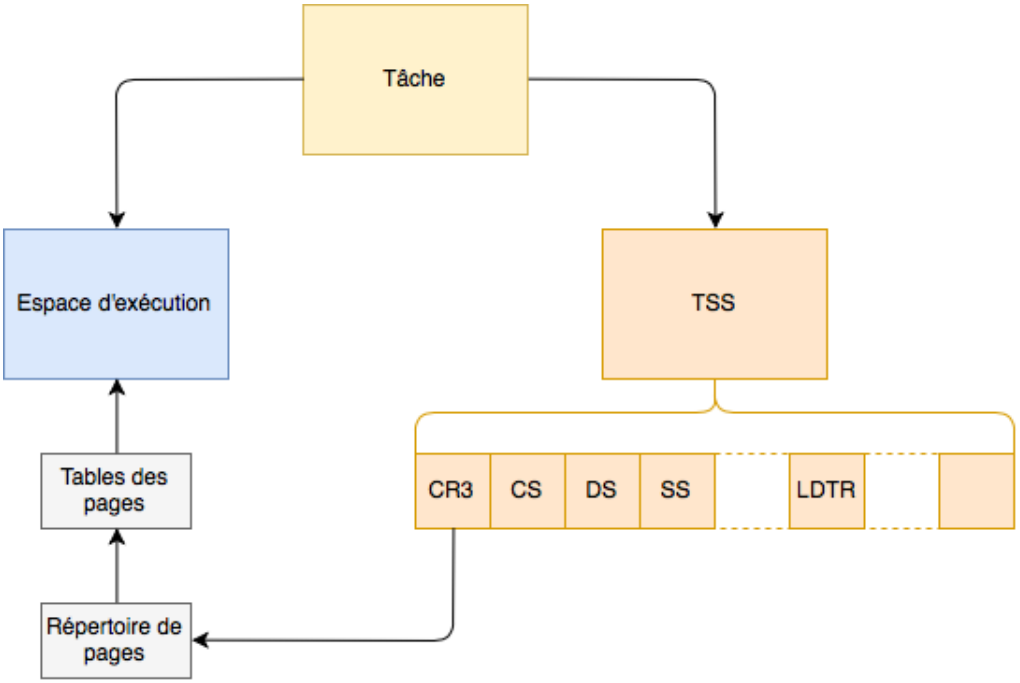

FIGURE 32 – Structure d'une tâche avec la pagination

La structure TSS permet aussi de sauvegarder le contexte de la tâche dans le cas où une tâche en appelle une autre. Chaque TSS a un descripteur dans la GDT. Il y a donc autant de descripteurs de TSS dans la GDT que de tâches. Une tâche est alors identifiée par le sélecteur de segment référençant son TSS dans la GDT [1]. Etant donné que la pagination est utilisée dans la version finale de notre OS, nous allons nous concentrer sur la gestion des tâches utilisant un répertoire de pages.

8.2.2 Commutation de tâche

Nous avons vu dans la partie précédente qu'une tâche est divisée en deux parties, son espace d'exécution et sa structure TSS. Le processeur fourni un mécanisme permettant de sauvegarder le contexte de la tâche courante et de commuter vers une nouvelle tâche. La sauvegarde du contexte se fait en utilisant le TSS lié à la tâche à exécuter et plus particulièrement le sélecteur de ce TSS dans la GDT. Ce dernier doit être donné commer argument à l'instruction `ltr`. Cette instruction permet de charger le sélecteur de TSS dans un registre spécial, le *task register*. Ce registre indique au processeur quelle est la tâche courante. Lorsqu'une commutation de tâche a lieu, l'état courant du processeur est automatiquement sauvegardé dans le TSS pointé par le *task register*. Il est donc nécessaire de créer un TSS initial avant d'exécuter la première tâche car l'état du processeur doit être sauvegardé [1]. Ci-dessous, la structure TSS en rust.

```
pub struct Tss {
    previous_task_link : u16, reserved0    : u16,
    esp0               : u32,
    ss0                : u16, reserved1    : u16,
    esp1               : u32,
    ss1                : u16, reserved2    : u16,
    esp2               : u32,
    ss2                : u16, reserved3    : u16,
    cr3                : u32,
    eip                : u32, eflags       : u32, eax: u32, ecx: u32, edx: u32,
    ebx                : u32, esp          : u32, ebp: u32, esi: u32, edi: u32,
    es                 : u16, reserved4    : u16,
    cs                 : u16, reserved5    : u16,
    ss                 : u16, reserved6    : u16,
    ds                 : u16, reserved7    : u16,
    fs                 : u16, reserved8    : u16,
    gs                 : u16, reserved9    : u16,
    ldt_selector       : u16, reserved10   : u16,
    reserved11         : u16,
    iomap_base_addr    : u16
}
```

Dans le TSS initial, seuls les champs `esp0`, `ss0` et `cr3` sont utilisés. Les champs `esp0` et `ss0` sont utilisés pour stocker la pile pour le niveau de privilèges 0 et `cr3` doit pointer sur le répertoire de pages du *kernel*. Ce TSS est ensuite chargé dans le *task register* avec l'instruction `ltr`. Le *kernel* peut maintenant exécuter une tâche utilisateur. L'architecture IA-32 offre de nombreux mécanismes pour commuter vers une nouvelle tâche. La méthode utilisée est un appel explicite à la tâche avec l'instruction `call far`. Cette instruction prend comme argument le sélecteur du TSS de la tâche à exécuter (comme l'instruction `ltr`). L'initialisation du TSS d'une tâche est légèrement plus complexe que celle du TSS initial. Les sélecteurs de segment doivent pointer sur les bons segments dans la GDT. De plus, le pointeur d'exécution doit pointer là où commence le code du programme utilisateur et les champs `ss`, `esp` et `ebp` sont utilisés pour stocker la pile pour le niveau de privilèges 3. L'adresse du début du programme utilisateur dépend de la manière dont le répertoire de pages de la tâche a été initialisé. Dans notre cas, le programme utilisateur commence toujours à l'adresse 0x0. Un programme utilisateur contenu par exemple dans le disque dur peut ainsi être exécuté en *ring* 3. Ce programme ne pourrait par contre pas faire grand chose car il n'aurait le droit d'accéder ni aux périphériques, ni à la mémoire. Les appels systèmes permettent de résoudre ce problème.

8.3 Appels systèmes

Les appels systèmes sont des fonctions exposés aux applications utilisateur par le *kernel*. Lors d'un appel système, un changement de privilèges a lieu car du code du *kernel* est appelé. Il est important d'avoir des appels systèmes dans un *kernel* appelant des applications utilisateurs car un programme utilisateur ne pourrait que modifier des variables et appeler des fonctions s'il ne pouvait pas appeler du code au niveau du *kernel*. Prenons par exemple un simple affichage texte. Le programme utilisateur étant isolé du reste de la mémoire, il ne peut pas accéder à la VRAM et par conséquent ne peut rien afficher à l'écran. Les appels systèmes se présentent donc comme une API de l'OS. Le mécanisme permettant à du code exécuté en mode utilisateur d'appeler du code en mode *kernel* est réalisé grâce à une interruption logicielle choisie et configurée par le *kernel*. Cette interruption est configurée afin d'être exécutable en *ring* 3. Dans le cas de notre *kernel*, l'interruption utilisée est la première libre après les interruptions matérielles. C'est l'interruption 48. Sa configuration se fait avec le code rust ci-dessous.

```
IDT[48] = IdtEntry::new(GDT_KERNEL_CODE_SELECTOR as u16,
                        _syscall_handler as *const () as u32,
                        TYPE_TRAP_GATE, DPL_USER);
```

La structure des entrées de l'IDT est décrite dans la figure 23. Dans ce code, le constructeur d'une entrée prend comme argument le sélecteur de segment pour accéder à l'ISR (ici le sélecteur de segment de code du *kernel*), le pointeur vers l'ISR en question, le type d'entrée (ici c'est une *trap gate*) et enfin le niveau de privilèges pour appeler cette interruption. Ainsi, cette interruption peut être appelée avec l'instruction `INT 48`. A noter qu'une seule interruption logicielle gère tous les appels systèmes. Ceci est possible car un numéro d'appel système est passé en argument à la routine d'interruption qui s'occupe d'appeler la bonne fonction. En général, d'autres paramètres doivent être passés à l'appel système dépendamment de l'action faite par ce dernier. Par exemple un chaîne de caractères à afficher à l'écran. Tous ces paramètres sont envoyés à la routine d'interruption dans les registres du processeur (`eax`, `ebx`, `ecx`, `edx` et `esi`).

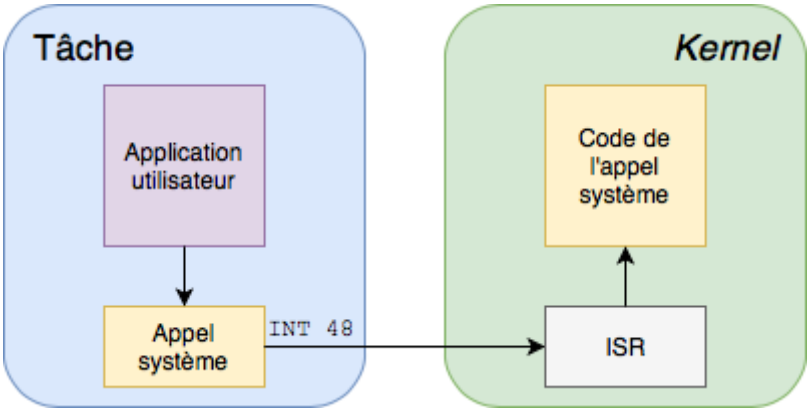

FIGURE 33 – Fonctionnement des appels systèmes

Avec la pagination active, il est important de copier les tables des pages et les pages du *kernel* dans le répertoire de pages de la tâche. C'est pour cette raison que nous avons déplacé le *kernel* dans le dernier gigaoctet de RAM (chapitre 5.3.2). De cette façon, les trois premiers gigaoctets de RAM peuvent être utilisés par une application utilisateur et le reste par le *kernel*.

8.4 Allocation dynamique en mode utilisateur

Le concept d'allocation dynamique a été expliqué dans le chapitre 5.4. Les fonction décrites dans ce chapitre permettent seulement d'allour de la mémoire en mode *kernel*. Les tâches utilisateur pourraient très bien se passer d'allocation dynamique et utiliser des structures statiques à la place. C'est d'ailleurs de cette manière qu'avaient été implémenté les tâches dans une version précédente de l'OS. Le problème qu'il y a avec l'utilisation de structures statique est que la même quantité de mémoire va être utilisée pour une application faisant 200 octets et une application faisant 200000 octets. De plus, toute la mémoire dédiée aux tâches doit alors être réservée dès la compilation du *kernel* ce qui augmente très rapidement sa taille. Un autre avantage à allouer la mémoire dynamiquement en *ring* 3 est que des fonctions d'allocation dynamique peuvent alors être implémentées au niveau utilisateur (comme `malloc` et `free`).

Dans le chapitre 5.4, nous avons vu les structures et mécanismes mis en place pour allouer de la mémoire dynamiquement en mode *kernel*. Des fonctions supplémentaires ont été implémentées pour gérer la mémoire dynamique en mode *user*. Ces fonctions sont `umalloc` et `ufree`. La fonction `umalloc` alloue d'abord de la mémoire en *ring* 0 puis *map* ces blocs mémoire dans le répertoire de pages de la tâche utilisateur en commençant à l'adresse 0x0. Le code du programme utilisateur est donc situé physiquement dans le *heap* du *kernel* mais est placé virtuellement au début de l'espace d'adressage de la tâche. Ce mécanisme est bien expliqué par la figure 34. Dans cet exemple, on peut voir que le code du programme utilisateur se situe dans le tas du *kernel* mais qu'il est *mappé* à l'adresse 0x0 dans l'espace d'adressage de la tâche. A noter que le *kernel* et son tas sont encore présents dans la mémoire en mode *user* pour rendre possible les appels systèmes (comme expliqué dans la partie précédente).

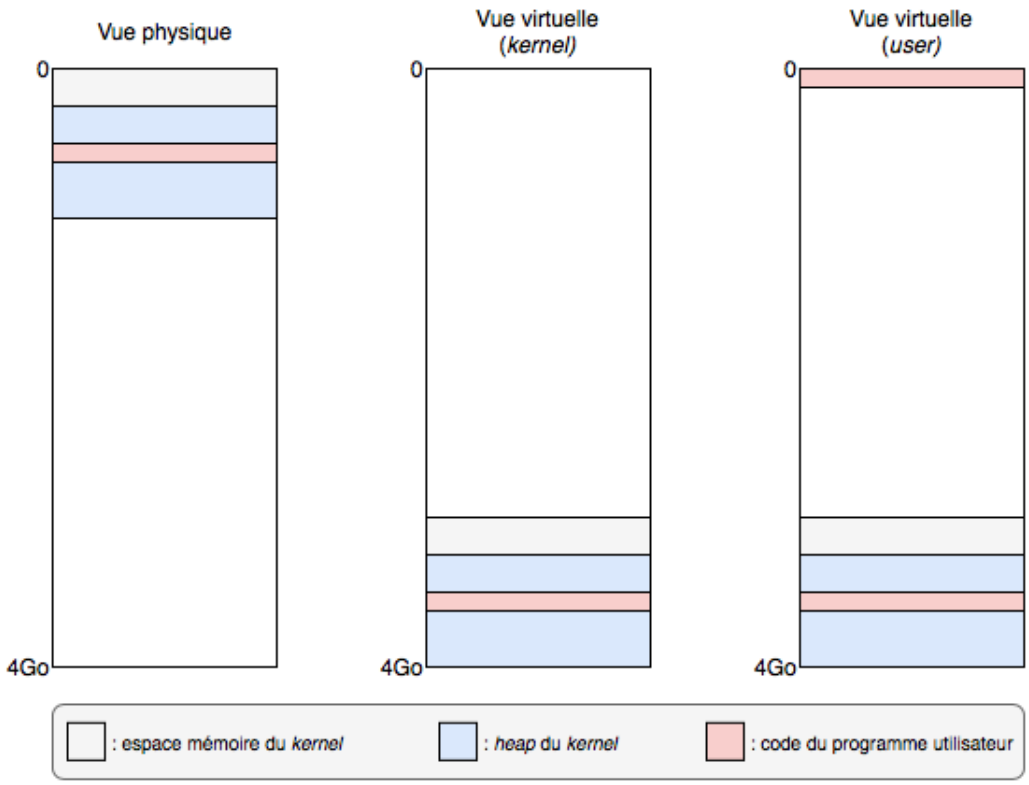

FIGURE 34 – Vues de la mémoire après allocation d'un code utilisateur

8.5 Librairie système et applications

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).

[6] Multiboot specifications. https://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[7] David Decotigny et Thomas Petazzoni. Segmentation et interruptions. http://sos.enix.org/wiki-fr/upload/SOSDownload/sos-texte-art2.pdf.

[8] Intel. Ia-32 intel architecture - software developer's manual - volume 3 : System programming guide. http://flint.cs.yale.edu/cs422/doc/24547212.pdf, 2003.

[9] Segmentation. https://wiki.osdev.org/Segmentation.

[10] Gdt. https://wiki.osdev.org/GDT.

[11] David Decotigny et Thomas Petazzoni. Mise en place de la pagination. http://sos.enix.org/wiki-fr/upload/SOSDownload/sos-texte-art4.pdf.

[12] Page translation. https://pdos.csail.mit.edu/6.828/2011/readings/i386/s05_02.htm.

[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] Ata pio mode. https://wiki.osdev.org/ATA_PIO_Mode.

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

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

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

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

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

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

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

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

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