

SYSTEME D'EXPLOITATION BARE METAL AVEC RUST

Thèse de Bachelor présentée par

Monsieur Orphée ANTONIADIS

pour l'obtention du titre de Bachelor of Science HES-SO en

Ingénierie des technologies de l'information avec orientation en

Informatique matérielle

Septembre 2018

Professeur HES responsable

GLUCK Florent

<p align="center">INGÉNIERIE DES TECHNOLOGIES DE L'INFORMATION ORIENTATION – INFORMATIQUE MATÉRIELLE SYSTÈME D'EXPLOITATION BARE METAL AVEC RUST</p>

Descriptif :

Traditionnellement, le langage C est le langage choisi pour l'écriture de systèmes d'exploitation. Le langage C a été créé dans ce but en 1972 et reste, malgré son âge, le langage le plus populaire pour les tâches systèmes et de bas niveau comme l'implémentation de systèmes d'exploitation ou l'embarqué.

Bien que C offre des performances (presque) inégalées, le langage souffre de nombreux problèmes, notamment en termes de fiabilité : gestion mémoire délicate source de nombreux bugs utilisateurs, bibliothèques standards incomplètes, absence de vrais modules, compilateurs trop permissifs, etc.

Ce projet vise à explorer le langage Rust, une alternative moderne au langage C dans le cadre de l'implémentation d'un système d'exploitation *bare metal* simple.

Premièrement, il s'agira d'étudier le langage Rust d'un point de vue système. Le langage Rust se révèle particulièrement intéressant en tant que successeur de C. Il a été conçu pour être robuste et extrêmement sécurisé tout en offrant des performances comparables à C. La première partie de ce 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, en Rust, un système d'exploitation simple pour l'architecture Intel IA-32.

Travail demandé :

- Etude et prise en main du langage Rust, notamment familiarisation avec le compilateur et l'éditeur de liens dans un contexte système *bare metal*.
- Etude des sujets liés à la conception et l'implémentation d'un système d'exploitation pour IA-32 :
 - Processus de compilation et édition des liens avec Rust.
 - Emulation système avec QEMU.
 - Format ELF et bibliothèques.
 - Assembleur IA-32.
 - Périphériques PC : affichage, clavier, timer, contrôleur d'interruption.
 - Bootloader grub.
 - Architecture et adressage mémoire IA-32 (segmentation, pagination, allocation dynamique).
 - Interruptions logicielles, matérielles et exceptions.
 - Système de fichiers *bare metal* simple.
 - Niveaux de privilège (*rings*), mode noyau et mode utilisateur.
 - Mécanisme de commutation de tâches.
 - Appels systèmes.
 - Applications utilisateur.
- Comparaison avec le langage C.
- Démonstration du système d'exploitation développé.

Candidat :

M. ANTONIADIS Orphée

Filière d'études : ITI

Professeur(s) responsable(s) :

GLUCK Florent

En collaboration avec :

Travail de bachelor soumis à une convention
de stage en entreprise : **non**Travail de bachelor soumis à un contrat de
confidentialité : **non**

Résumé :

Le langage C est généralement le langage utilisé dans le domaine de la programmation système. En effet, le langage étant bas niveau (opérations bit à bit, utilisation de pointeurs), il dispose d'une très grande rapidité d'exécution. De plus, il est compatible avec de nombreuses architectures car il est basé sur un standard ouvert. Ce langage souffre tout de même de nombreux problèmes. La gestion de la mémoire est la cause de beaucoup d'entre eux. Par exemple, le dépassement de tampon (*buffer overflow*) reste une faille de sécurité encore exploitable. Dans ce contexte, des alternatives au langage C (C++, D, Go, Rust) ont rapidement été proposées. Le langage Rust est développé par les ingénieurs de Mozilla. Il est décrit par ces derniers comme étant extrêmement sécurisé. Plus robuste et tout aussi performant, il est vu par beaucoup comme le successeur du C.

Le projet consiste à implémenter un système d'exploitation simple pour l'architecture Intel 32 bits (IA-32) avec le langage de programmation Rust. Ce système d'exploitation a été sobrement appelé RustOS. Tous les tests ont été effectués avec la machine virtuelle QEMU mais RustOS peut être exécuté sur n'importe quel ordinateur compatible.

[illegible]

Les fonctionnalités de RustOS sont les suivantes :

- Gestion mémoire par pagination
- Allocation dynamique (`malloc`, `free`)
- Affichage VGA mode texte
- Système de fichiers inspiré de FAT
- Librairie système
- Applications utilisateurs dont un shell

Candidat :

M. ANTONIADIS Orphée

Filière d'études : ITI

Professeur responsable :

GLUCK Florent

En collaboration avec :

Travail de bachelor soumis à une convention

de stage en entreprise : non

Travail de bachelor soumis à un contrat de confidentialité : non

Table des matières

Descriptif	1
Résumé	3
Table des matières	5
Table des figures	7
Table des tables	8
Table des listings de code	9
Acronymes	10
Avant-propos	11
1 Introduction	12
1.1 Contexte	12
1.2 Objectifs	12
2 Architecture globale	13
2.1 Architecture de l'OS	13
2.2 Environnement de développement	14
3 Langage Rust	15
3.1 Introduction	15
3.2 Installation et compilation	15
3.3 Bases du langage	18
3.4 Spécificités du langage	21
4 Exécution du <i>kernel</i>	23
4.1 Introduction	23
4.2 Compilation du <i>kernel</i>	23
4.3 <i>Boot</i> de l'OS	26
5 Gestion mémoire	28
5.1 Introduction	28
5.2 Segmentation	29
5.3 Pagination	34
5.4 Allocation dynamique en mode <i>kernel</i>	37
6 Périphériques	40
6.1 Ports	40
6.2 Interruptions et Exceptions	40
6.3 VGA	45
6.4 <i>Timer</i>	47
6.5 Clavier	48

7	Système de fichiers	49
7.1	Introduction	49
7.2	Structure	50
7.3	Implémentation	51
8	Tâches utilisateurs	53
8.1	Introduction	53
8.2	Exécution d'une tâche	54
8.3	Appels systèmes	56
8.4	Allocation dynamique en mode utilisateur	57
8.5	Librairie système et applications	58
9	Résultats	61
10	Discussions	62
10.1	Problèmes rencontrés	62
10.2	Améliorations possibles	62
11	Conclusion	63
	Références	64

Table des figures

1	Architecture du système d'exploitation	13
2	Architecture globale du projet	14
3	<i>Boot</i> d'une machine à base de BIOS	23
4	Exemple d'adressage mémoire	28
5	Protection mémoire avec un MMU	28
6	Translation d'adresse	29
7	Conversion d'une adresse logique en adresse linéaire	30
8	Structure d'un sélecteur de segment	30
9	Exemple d'une GDT	31
10	Structure d'une entrée dans la GDT	32
11	Modèle de segmentation de type <i>flat</i>	32
12	Descripteur de GDT	33
13	Structure d'une <i>Page Entry</i>	34
14	Exemple de pagination à 3 niveaux	35
15	Répertoire de pages adressant le <i>kernel</i> au début de la RAM	36
16	Répertoire de pages adressant le <i>kernel</i> à la fin de la RAM	36
17	Entête d'un bloc de mémoire dans le tas	37
18	Algorithme utilisé pour l'allocation dynamique dans le <i>kernel</i>	38
19	Etat initial de la chaîne d'entêtes	38
20	Allocation d'une page	39
21	Allocation d'une page et d'une table des pages	39
22	Table des interruptions et exceptions sur IA-32	41
23	Table de correspondance des IRQs	42
24	Différents types de descripteur d'interruption	43
25	Relation entre le registre IDTR et l'IDT	44
26	Structure d'un caractère en mode texte VGA	45
27	Couleurs disponibles en mode texte VGA	45
28	Structure d'un <i>scan code</i>	48
29	Structure d'un système de fichiers de type FAT	49
30	Système de fichiers de l'OS	51
31	Menu du gestionnaire du système de fichiers	52
32	Niveaux de privilèges sur architecture IA-32	53
33	Structure d'une tâche avec la pagination	54
34	Fonctionnement des appels systèmes	56
35	Vues de la mémoire après allocation d'un code utilisateur	57
36	Shell développé	60

Table des tables

1	Structure du projet	14
2	Description des sections d'un programme	25
3	Etapes d'initialisation de GRUB	26
4	Registres de segment	30
5	Structure du <i>superblock</i>	50
6	Structure d'une entrée dans l'espace de métadonnées	50
7	Fonctions proposées par la librairie système	59

Table des listings de code

1	Premier programme en Rust	15
2	Contenu du fichier <code>Cargo.toml</code>	16
3	Contenu du fichier <code>i386-rust_os.json</code>	17
4	Commentaires en Rust	18
5	Exemple de variable immutable	18
6	Exemple de variable mutable	18
7	Exemple de <i>shadowing</i>	18
8	Déclaration d'une constante et d'une variable statique	18
9	Code source de la macro <code>println!</code>	19
10	Utiliser l'expression <code>if</code> dans une déclaration de variable	19
11	Exemple de structure en Rust	20
12	Appels aux méthodes d'une structure	20
13	Déclaration d'un <code>enum</code>	20
14	Changement d' <i>ownership</i>	21
15	Implémentation de traits pour une structure	21
16	Section à ajouter au fichier TOML	22
17	Module <code>tests</code>	22
18	Section <code>lib</code> du fichier <code>Cargo.toml</code>	24
19	Fichier principal du <i>kernel</i>	24
20	<i>Linker script</i> du <i>kernel</i>	25
21	Constructeurs d'une entrée dans la GDT	33
22	<i>Linker script</i> du <i>higher-half kernel</i>	35
23	Modification apportée au <i>linker</i>	37
24	Code assembleur rendant accessible l'expression <code>kernel_end</code>	37
25	Structure <code>Screen</code>	46
26	Implémentation des macros <code>print</code> et <code>println</code>	46
27	Programmation du <i>timer</i> en assembleur	47
28	Implémentation de la fonction <code>sleep</code>	47
29	Routine d'interruption du clavier	48
30	Prototypes des fonctions d'écriture/lecture dans le disque dur	52
31	Champs de la structure TSS	55
32	Entrée dans l'IDT pour les appels systèmes	56
33	<i>Linker</i> pour un exécutable en format binaire	58
34	Options ajoutées au fichier <code>cargo.toml</code>	58

Acronymes

ABI *Application Binary Interface*. 16

API *Application Programming Interface*. 51, 55

BIOS *Basic Input Output System*. 22, 25

CPU *Central Processing Unit*. 22, 25, 29, 39, 42, 46

CRTC *Cathode Ray Tube Controller*. 44

ELF *Executable and Linkable Format*. 13, 22, 24, 26, 57

FAT *File Allocation Table*. 48–50

GCC *GNU Compiler Collection*. 22, 24, 57

GDT *Global Descriptor Table*. 28–32, 42, 43, 52–54

GRUB *GRand Unified Bootloader*. 13, 22, 25, 26

IA-32 *Intel Architecture 32 bits*. 10–13, 16, 25, 27, 39, 40, 46, 48, 52–54

IDT *Interrupt Descriptor Table*. 40–43, 52, 55

IRQ *Interrupt Request*. 41, 46, 47

ISO *International Organization for Standardization*. 13, 22, 25, 26, 48

ISR *Interrupt Service Routine*. 39, 41, 42, 55

JSON *JavaScript Object Notation*. 16

LDT *Local Descriptor Table*. 28–30, 32, 52, 53

MBR *Master Boot Record*. 22, 25

MMIO *Memory Mapped Input/Output*. 39

MMU *Memory Management Unit*. 27, 28, 30, 32, 52

NASM *Netwide Assembler*. 22, 24

NMI *Non Maskable Interrupt*. 41

OS *Operating System*. 11–13, 16, 24, 25, 27, 28, 30–32, 34, 36, 43–45, 48, 50, 52, 53, 55, 56, 58, 59

PC *Personal Computer*. 22, 44, 46

PIC *Programmable Interrupt Controller*. 41

PIO *Port Input/Output*. 39

PIT *Programmable Interval Timer*. 46

POSIX *Portable Operating System Interface*. 51

RAM *Random Access Memory*. 27, 34, 35, 37, 43, 48, 53, 55

TOML *Tom’s Obvious, Minimal Language*. 15, 21

TSS *Task State Segment*. 28, 53, 54

VGA *Video Graphics Array*. 44–46

VRAM *Video Random Access Memory*. 24, 44, 55

Avant-propos

Présentation

Ce mémoire constitue l'aboutissement de 11 semaines de travail dans le cadre de l'obtention du titre de Bachelor en Ingénierie des Technologies de l'Information, orientation informatique matérielle, à hepia. Le projet consiste à implémenter un système d'exploitation simple pour l'architecture IA-32 avec le langage de programmation Rust. Ce document commence par introduire le sujet en rappelant les objectifs et en décrivant l'architecture du système. Les caractéristiques principales du langage Rust sont ensuite détaillées. L'implémentation du système d'exploitation est expliquée en plusieurs parties. Nous nous intéressons en premier lieu à la compilation et à l'exécution du système. Nous voyons ensuite les différentes techniques de gestion mémoire utilisées (segmentation, pagination, allocation dynamique). Après cela, nous nous intéressons à la manière dont un processeur Intel 32 bits permet de communiquer avec ses périphériques (ports d'entrées/sorties, interruptions). Le système de fichiers développé pour notre système est ensuite présenté. Enfin, nous expliquons le fonctionnement du mode utilisateur et les différents mécanismes permettant à une application utilisateur de communiquer avec le *kernel*. Ce rapport se termine par une présentation des résultats, une partie de discussions au sujet des problèmes rencontrés, des améliorations possibles et des différences entre le Rust et le C avant de conclure sur le bilan du travail accompli.

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 sont 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 }
```

- Toute commande importante dans le cadre du projet est écrite selon le format suivant :

```
$ echo Hello world!
```

Remerciements

J'aimerais remercier l'équipe enseignante pour la qualité de leurs enseignements, leur implication et leur pédagogie que j'ai pu apprécier au long de ces trois années passées à hepia. Merci à M. Glück, qui m'a suivi tout le long de ce travail de Bachelor, pour son aide et ses conseils. Merci à ma famille et à mes proches d'avoir été présents pendant toutes mes études. Pour finir, merci à Camille pour la relecture de ce mémoire mais surtout pour m'avoir soutenu et supporté durant ces 11 semaines.

1 Introduction

1.1 Contexte

Le langage C est généralement le langage utilisé dans le domaine de la programmation système. En effet, le langage étant bas niveau (opérations bit à bit, utilisation de pointeurs), il dispose d'une très grande rapidité d'exécution. De plus, il est compatible avec de nombreuses architectures car il est basé sur un standard ouvert. Ce langage souffre tout de même de nombreux problèmes. La gestion de la mémoire est la cause de beaucoup d'entre eux. Par exemple, le dépassement de tampon (*buffer overflow*) reste une faille de sécurité encore exploitable. Dans ce contexte, des alternatives au langage C (C++, D, Go, Rust) ont rapidement été proposées. Le langage Rust est développé par les ingénieurs de Mozilla. Il est décrit par ces derniers comme étant extrêmement sécurisé. Plus robuste et tout aussi performant, il est vu par beaucoup comme le successeur du C.

1.2 Objectifs

L'objectif principal de ce projet est d'implémenter un système d'exploitation avec le langage Rust. Une première partie a donc été un travail de recherche sur le langage. Le but était de comprendre le fonctionnement de Rust, ses caractéristiques principales ainsi que ses spécificités. De plus, il a fallu étudier son utilisation dans le cadre de la programmation système et plus précisément dans le développement d'un système d'exploitation *bare metal*. La seconde partie a été l'implémentation du système d'exploitation en question pour l'architecture IA-32. Une étude des sujets liés à la conception d'OS pour cette architecture a été faite. Le deuxième objectif de ce projet est de comparer le langage Rust au langage C. On a vu que le Rust est considéré comme étant le successeur du C. Les comparer dans le cadre de la programmation d'OS est un bon moyen de juger les performances de Rust. Pour résumer, ce travail de Bachelor a suivi les étapes suivantes :

- Etude du langage de programmation Rust dans le cadre de la conception d'un système d'exploitation *bare metal*
- Etude des sujets liés à la conception d'un système d'exploitation pour l'architecture IA-32
- Implémentation en Rust d'un système d'exploitation simple
- Comparaison entre le langage Rust et le langage C

2 Architecture globale

2.1 Architecture de l'OS

Notre système d'exploitation a été conçu pour être utilisé sur un ordinateur disposant d'un processeur Intel de la famille x86. L'architecture x86 regroupe plusieurs modes de fonctionnement. Les plus notables sont le mode réel (16 bits), le mode protégé (32 bits) et le *long mode* (64 bits). Le mode réel n'est aujourd'hui plus utilisé mais tous les processeurs de la famille x86, même modernes, démarrent dans ce mode avant de changer de mode. Le système d'exploitation développé utilise le mode protégé. Cette architecture se nomme IA-32 (Intel Architecture 32 bits, parfois appelée i386) [1].

Le mode protégé offre deux mécanismes de gestion mémoire. Ces mécanismes sont la segmentation et la pagination. Ils sont tous deux gérés par l'OS. La gestion de la mémoire est traitée dans le chapitre 5. Un processeur Intel 32 bits peut aussi communiquer avec de nombreux périphériques. Ceci se fait grâce à divers mécanismes comme les ports d'entrées/sorties ou les interruptions matérielles. Un de ces périphériques est le disque dur de l'ordinateur. Le disque dur doit avoir un système de fichiers pour être correctement utilisable par l'OS. Par exemple, le système de fichiers utilisé par Linux est ext2. Les périphériques et le système de fichiers sont décrits dans les chapitres 6 et 7. Enfin, le mode protégé fournit un mécanisme de protection par niveau de privilèges. Ce mécanisme permet d'exécuter du code destiné à l'utilisateur et de garantir la sûreté du système. Le chapitre 8 donne plus d'informations sur le mode utilisateur. Tous ces éléments sont gérés par le noyau (*kernel* en anglais) du système d'exploitation [1].

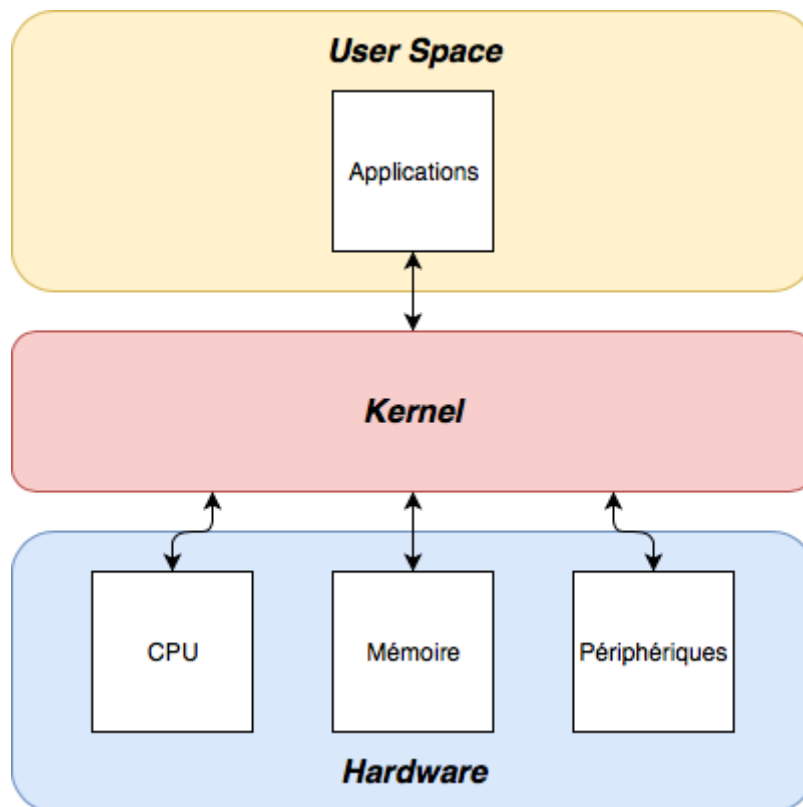


FIGURE 1 – Architecture du système d'exploitation

2.2 Environnement de développement

La machine utilisée pour le développement du projet est un MacBook Pro avec un processeur Intel à 3 GHz. Il a cependant fallu utiliser une machine virtuelle (VMware) utilisant Linux (Ubuntu 16.04.4 LTS) pour la compilation. Ce choix a été fait car il existe beaucoup plus de documentation sur l'implémentation de systèmes d'exploitation sur Linux que sur Mac. Bien que Mac OS soit un système UNIX, les 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. Linux est donc utilisé pour la compilation et l'exécution du système d'exploitation.

Le code doit être compilé pour être exécutable sous une architecture IA-32. La compilation du code Rust est expliquée dans le chapitre 3.2.2. Certaines parties du code du *kernel*, trop bas niveau pour le langage Rust, ont été écrites en assembleur. Il faut donc compiler le code Rust avec le code assembleur. Tout le mécanisme de génération d'un *kernel* est détaillé dans le chapitre 4. Dans ce projet, une machine virtuelle est utilisée pour démarrer sur notre système d'exploitation. Cette machine virtuelle est QEMU. Elle peut émuler plusieurs architectures dont l'architecture i386 qui nous intéresse ici. QEMU peut démarrer sur un système d'exploitation depuis un fichier ISO. La figure 2 illustre ce comportement [1].

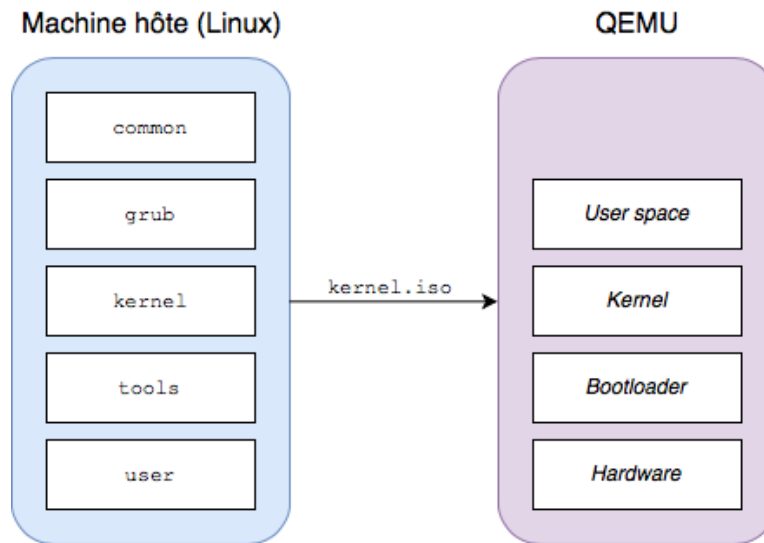


FIGURE 2 – Architecture globale du projet

D'un côté nous avons la machine de développement (ou machine hôte) contenant tous les fichiers servant à compiler le *kernel*. De l'autre il y a QEMU qui démarre le système d'exploitation à partir du *kernel*. Les fichiers côté machine de développement sont ordonnés d'une manière bien précise [1].

Répertoire	Description
<code>common</code>	Fichiers communs entre le <i>kernel</i> et les applications utilisateurs
<code>grub</code>	Fichiers de configuration du <i>bootloader</i> GRUB
<code>kernel</code>	Code source du <i>kernel</i>
<code>tools</code>	Codes source des différents outils développés pour l'OS
<code>user</code>	Codes source des des applications utilisateurs

TABLE 1 – Structure du projet

3 Langage Rust

3.1 Introduction

Rust est un langage de programmation système développé par Mozilla et conçu pour être sécurisé. Le but de Rust est de remplacer les autres langages système comme le C ou le C++. Les trois caractéristiques mises en valeur par l'équipe de Rust sont la rapidité, la sécurité (prévient des erreurs de segmentation) et la concurrence (garantit la sûreté entre *threads*) [2]. La première version de Rust est sortie en janvier 2012 [3]. Ce langage est donc très récent et emprunte beaucoup de concepts de programmation à d'autres langages. Une première partie de ce projet a été un travail de recherche afin d'apprendre ce langage. Beaucoup de documentation est disponible sur le site de Rust dont le livre de Rust, *The Rust Programming Language*, sorti en deux éditions différentes (*First Edition* et *Second Edition*). Ce livre est disponible en ligne et est une très bonne introduction au langage. Il donne beaucoup d'exemples et permet de comprendre rapidement le fonctionnement de Rust.

3.2 Installation et compilation

3.2.1 Installation de Rust

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. Rustup est un gestionnaire de version du langage Rust. Pour installer Rust ainsi que cet utilitaire, il faut exécuter la commande suivante [4].

```
$ curl https://sh.rustup.rs -sSf | sh
```

Le gestionnaire de version rustup peut maintenant être utilisé pour changer la version de Rust. Notre système d'exploitation utilise la version *nightly* de Rust, la commande suivante doit donc être exécutée.

```
$ rustup override add nightly
```

3.2.2 Compilation d'un programme en Rust

Le compilateur de Rust est rustc. Il est installé automatiquement lors de l'installation de Rust. Prenons le programme ci-dessous qui est contenu dans le fichier `hello.rs`.

```
1 fn main() {  
2     let hello = "Hello world!";  
3     println!("{}", hello);  
4 }
```

Listing 1 – Premier programme en Rust

Le mot-clé `let` indique qu'une variable est déclarée. On remarque qu'aucun type n'est donné dans la déclaration. L'inférence de types est l'un des nombreux concepts utilisés par Rust. L'équivalent en C serait `char hello[] = "Hello world!"`. Le programme affiche ensuite cette variable sur la sortie standard en utilisant la macro `println!`. Les macros seront vues plus en détail dans la suite de ce document. On peut les distinguer des fonctions

par la manière dont elles sont appelées. L'appel d'une macro se fait avec son nom suivi d'un '!'. Ce programme affiche donc "Hello world!" sur la sortie standard. Pour générer un exécutable à partir de ce code, le compilateur de Rust, `rustc` peut être utilisé avec la commande `rustc hello.rs`. Cette méthode fonctionne bien lorsqu'on ne veut compiler qu'un seul fichier comme dans cet exemple. Cela devient vite compliqué lorsqu'il faut gérer des dépendances entre plusieurs fichiers et utiliser des bibliothèques externes. Heureusement, une solution existe pour résoudre ce problème. Rust fournit un gestionnaire de paquets nommé Cargo.

3.2.3 Cargo

Le gestionnaire de paquets de Rust, Cargo, peut non seulement gérer les dépendances d'un projet mais s'occupe aussi de sa compilation [5]. Cargo est installé en même temps que Rust, il n'y a donc pas de manipulations supplémentaires à faire pour l'utiliser. Cargo fonctionne à l'aide d'un fichier de configuration du projet au format TOML. Ce fichier doit être appelé `Cargo.toml`. Le contenu du fichier `Cargo.toml` ressemble généralement au code dans le *listing 2* [4].

```
1  [package]
2  name = "hello"
3  version = "0.1.0"
4  authors = ["Your Name <you@example.com>"]
5
6  [dependencies]
```

Listing 2 – Contenu du fichier `Cargo.toml`

Cargo compile et gère automatiquement les dépendances de tous les fichiers contenus dans le répertoire `src`. Si on reprend l'exemple du *listing 1*, un projet utilisant Cargo aurait l'arborescence suivante.

```
hello
├── Cargo.toml
└── src
    └── main.rs
```

Des dépendances externes (appelées *crates*) peuvent être utilisées dans le projet en les ajoutant dans le fichier TOML, sous la section *dependencies*. Une *crate* peut être en local sur la machine hôte ou être disponible en ligne sur [Crate.io](https://crates.io). Ce site sert de registre pour toutes les bibliothèques Rust développées par la communauté. Cargo télécharge automatiquement les dépendances depuis [Crate.io](https://crates.io) [4]. Pour compiler un projet, `rustc` n'a plus à être utilisé. Cargo fait appel au compilateur de rust lui même. Pour demander à cargo de compiler le projet il faut exécuter la commande `cargo build`. Si le projet est un exécutable, il est lancé avec la commande `cargo run`. Notre système d'exploitation utilise un gestionnaire de paquets basé sur Cargo. Celui-ci se nomme Xargo. Il peut être installé directement depuis Cargo en utilisant la commande `cargo install xargo`. Xargo est fait pour compiler un projet Rust dans un environnement *bare metal*. Dans ce type d'environnement il n'y a aucune bibliothèque système étant donné que nous créons nous même le système. Le code doit donc être compilé sans dépendances à la bibliothèque standard. Rust a tout de même besoin d'une base pour être compilé. Cette base est fournie par la bibliothèque `core`. Xargo lie la bibliothèque `core` au projet automatiquement contrairement à Cargo [6].

3.2.4 Compilation croisée

Dans le cadre du projet, le code Rust a du être compilé pour une architecture Intel 32-bits (IA-32). On parle ici de compilation croisée car une machine hôte (*host* en anglais) compile du code pour une machine cible (*target* en anglais). Quand on fait de la compilation croisée, on se réfère à des caractéristiques nommées *triple*. Un *triple* a généralement le format architecture-vendeur-système-ABI. Par exemple pour *cross-compiler* pour une machine Apple, le format du *triple* est `x86_64-apple-darwin`. A noter que l'ABI n'est pas spécifiée. Rust propose un système natif de compilation croisée pour de nombreuses cibles. Ceci peut se faire avec le compilateur `rustc` ou bien avec Cargo en spécifiant la machine cible avec l'option `--target=` suivie du *target triple*. Il est possible de configurer son propre *target triple* dans un fichier JSON placé à la racine du projet. Dans ce cas, il faut donner le nom du fichier (sans l'extension `.json`) à l'option de Cargo. Dans notre OS, le nom du fichier est `i386-rust_os.json`. Notre projet a donc l'arborescence suivante.

```
kernel
├─ Cargo.toml
├─ i386-rust_os.json .2 src
└─ kernel.rs
```

Le fichier `i386-rust_os.json` configure la machine cible pour l'architecture i386 (IA-32) et sans système d'exploitation. Ci-dessous, le contenu de ce fichier.

```
1 {
2   "llvm-target": "i386-unknown-none",
3   "data-layout": "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128",
4   "linker-flavor": "gcc",
5   "target-endian": "little",
6   "target-pointer-width": "32",
7   "target-c-int-width": "32",
8   "arch": "x86",
9   "os": "none",
10  "disable-redzone": true,
11  "features": "-mmx,-sse,+soft-float",
12  "panic-strategy": "abort"
13 }
```

Listing 3 – Contenu du fichier `i386-rust_os.json`

Tous les champs du *target triple* sont décrits dans la documentation de Rust [7]. Le contenu de ce fichier est inspiré de plusieurs *target triples* trouvés sur internet. L'un dans une *issue* du *repository* de Rust sur GitHub [8] et l'autre dans un blog parlant de l'écriture d'un système d'exploitation en Rust [6]. La compilation croisée de notre *kernel* peut maintenant se faire avec la commande `xargo build --target=i386-rust_os`. A noter qu'il existe un *bug* avec l'option `target` quand on utilise notre propre *target triple* [9]. Pour ne pas qu'il se produise, il faut indiquer le chemin d'accès au répertoire contenant le fichier JSON en utilisant la variable d'environnement `RUST_TARGET_PATH`. La commande finale pour compiler le projet est la suivante :

```
$ RUST_TARGET_PATH=$(pwd) xargo build --target i386-rust_os
```

3.3 Bases du langage

3.3.1 Commentaires

Les commentaires en Rust ont la même syntaxe que les commentaires en C. Le *listing 4* montre les différents types de commentaire.

```
1 // Commentaire sur une seule ligne
2
3 /* Commentaire sur
4    plusieurs lignes */
```

Listing 4 – Commentaires en Rust

3.3.2 Variables et mutabilité

Les variables en Rust sont toutes constantes par défaut. Le code du *listing 5* ne pourra pas compiler.

```
1 let x = 0;
2 x += 1; // Erreur !
```

Listing 5 – Exemple de variable immutable

Pour rendre une variable mutable, le mot-clé **mut** doit être utilisé. Le code du *listing 6* corrige celui du *listing 5*.

```
1 let mut x = 0;
2 x += 1; // Plus d'erreur, la variable x est mutable
```

Listing 6 – Exemple de variable mutable

Une variable peut aussi être réécrite. On appelle ça le *shadowing*.

```
1 let x = 0;
2 let x = "zero";
```

Listing 7 – Exemple de *shadowing*

Le mot-clé **let** sans **mut** peut faire penser aux constantes en C. Rust permet aussi de déclarer des constantes avec le mot-clé **const**. Lors d'une déclaration de constante en Rust, le type de la variable doit être donné ainsi que sa valeur qui doit être constante (valeur fixée à la compilation et non au *runtime*). Une constante est déclarée globalement. Pour déclarer une variable globalement il faut utiliser le mot-clé **static**. Comme pour les constantes, quand une variable est déclarée statiquement, son type et sa valeur doivent être déterminés à la compilation.

```
1 const KHEAP_SIZE      : usize = 0x1000000;
2 static mut KHEAP_ADDR : u32   = 0;
```

Listing 8 – Déclaration d'une constante et d'une variable statique

3.3.3 Fonctions et macros

Les fonctions en Rust sont semblables aux fonctions en C. Leur déclaration se fait avec le mot-clé **fn**. Les macros de Rust reprennent le concept des macros du langage C. La particularité d'une macro par rapport à une fonction est qu'une macro va être remplacée à la compilation par le code qu'elle contient. De plus, le nombre d'arguments d'une macro n'est pas fixe. Une macro fonctionne par motifs. Prenons pour exemple l'implémentation de la macro `println!`.

```
1 macro_rules! println {  
2     () => (print!("\n"));  
3     ($fmt:expr) => (print!(concat!($fmt, "\n")));  
4     ($fmt:expr, $($arg:tt)*) => (print!(concat!($fmt, "\n"), $($arg)*));  
5 }
```

Listing 9 – Code source de la macro `println!`

Cette macro a trois motifs différents. Si aucun argument n'est donné à la macro, le premier motif va être utilisé. Nous pouvons voir qu'il affiche simplement un saut de ligne avec la macro `print!`. Dans une macro, la partie gauche contient un ensemble de symboles et la partie droite l'action à effectuer avec les symboles. Prenons maintenant la deuxième ligne de cette macro. Pour que ce motif soit appelé, une expression doit être donnée à la macro. Le symbole de cette expression est `$fmt` (comme le nom d'un argument dans une fonction, le symbole d'une macro est arbitraire). Ce *pattern* est appelé quand par exemple nous ne voulons afficher qu'une chaîne de caractères non formatée (`println!("Hello")`). La dernière ligne de cette macro représente le cas où un ou plusieurs arguments supplémentaires sont donnés à la macro. Dans cet exemple, c'est le cas où on veut formater la chaîne de caractères comme dans le *listing* 1. Quand un symbole est suivi d'un '*', cela veut dire qu'il peut être répété zéro ou plus de fois [10].

3.3.4 Structures de contrôle

Le langage Rust propose les mêmes structures de contrôle que le langage C. Une condition se fait avec l'expression **if** et du code peut être répété avec les expressions **for**, **while** et **loop**. Rust possède également une structure de contrôle pour faire du *pattern matching*. Elle est utilisée avec l'expression **match**. A noter qu'on parle ici d'expression. Une expression peut être utilisée dans la déclaration d'une variable. Le code suivant est donc possible [4].

```
1 let condition = true;  
2 let number = if condition {  
3     5  
4 } else {  
5     6  
6 };
```

Listing 10 – Utiliser l'expression **if** dans une déclaration de variable

Dans cet exemple, la variable **number** sera initialisée à la valeur 5.

3.3.5 Structures de données

Rust donne aussi accès à des structures de données, semblables aux structures C dans leur déclaration mais néanmoins différentes. Ces structures peuvent implémenter des méthodes. On peut donc aussi les comparer aux classes en Java. Ci-dessous, une structure implémentant deux méthodes.

```
1  #[derive(Clone)]
2  #[repr(C, align(4096))]
3  pub struct PageTable {
4      pub entries: [u32; 0x400]
5  }
6  impl PageTable {
7      fn null() -> PageTable {
8          PageTable {
9              entries: [0; 0x400]
10         }
11     }
12     pub fn as_ptr(&mut self) -> u32 {
13         self as *const PageTable as u32
14     }
15 }
```

Listing 11 – Exemple de structure en Rust

Ici, la structure `PageTable` ne contient qu'un seul champs mais il peut y en avoir plus. Les deux lignes commençant par un `#` sont des attributs appliqués à la structure [4]. Le premier attribut permet d'inclure automatiquement les fonctionnalités du trait `Clone` à notre structure. Le deuxième attribut indique que cette structure doit être représentée comme en C dans la mémoire et être alignée sur 4096 octets. L'implémentation de méthodes pour une structure se fait avec le mot-clé `impl`. La première méthode est un constructeur renvoyant simplement une structure initialisée avec des zéros. La deuxième méthode peut être appliquée à une structure, elle renvoie l'adresse de cette dernière. Le *listing* 12 montre comment appeler ces méthodes.

```
1  let table = PageTable::null();
2  let table_addr = table.as_ptr();
```

Listing 12 – Appels aux méthodes d'une structure

A noter que le mot-clé `pub` est utilisé à trois reprises dans le *listing* 11. Ce mot-clé contrôle la visibilité de la variable, la fonction, le champ ou la structure qui lui est associé. Il fait le contraire du mot-clé `private` en Java (de base tout est privé dans un fichier Rust) [4]. En plus des structures, Rust propose aussi les énumérations ou `enum`. Un `enum` permet de définir un type en énumérant toutes ses valeurs possibles [4]. Le *listing* 13 donne un exemple d'`enum` dont chaque valeur est sur huit bits.

```
1  #[repr(u8)]
2  pub enum Color {
3      Black = 0x0,
4      White = 0xf
5  }
```

Listing 13 – Déclaration d'un `enum`

3.4 Spécificités du langage

3.4.1 *Ownership* et références

L'*Ownership* est la fonctionnalité de Rust lui permettant de garantir la sécurité au niveau de la mémoire. Elle est décrite dans le livre de Rust comme la fonctionnalité fondamentale du langage. Chaque valeur en Rust est liée à une variable nommée propriétaire (*owner* en anglais). Une valeur ne peut avoir qu'un seul propriétaire à la fois. Quand on sort de la portée de la variable, la valeur n'est plus accessible [4]. Prenons pour exemple la structure `PageTable` décrite plus haut. Le code suivant provoquera une erreur.

```
1 let table1 = PageTable::null();
2 let table2 = table1;           // table2 prend l'ownership de la valeur de table1
3 let table_addr = table1.as_ptr(); // Erreur ! table1 n'a plus de valeur
```

Listing 14 – Changement d'*ownership*

Pour corriger cette erreur, il faut copier la valeur de la variable `table1` dans `table2` en utilisant la méthode `clone()`. C'est pour cette raison que nous avons spécifié l'utilisation des fonctionnalités du trait `Clone` lors de la déclaration de la structure dans le *listing* 11. Notre variable peut être stockée sur la pile car sa taille est connue à la compilation. Un autre trait offert par Rust peut être utilisé ici pour copier sa valeur. C'est le trait `Copy`. Les fonctionnalités de ce trait peuvent être rajoutées à une structure stockée dans la pile en l'ajoutant à l'attribut `derive`. Grâce à ce trait, il n'y a pas besoin d'appeler la méthode `clone()` pour copier la valeur. La copie est faite automatiquement. Pour ne pas perdre l'*Ownership* lors du passage d'une variable à une fonction, une référence à cette variable peut être donnée à la place (opérateurs `&` et `&mut`) [4].

3.4.2 Traits

Un trait en Rust ajoute des fonctionnalités à un type. Ces fonctionnalités peuvent être partagées entre plusieurs types. Il existe des traits déjà disponibles dans le code de Rust comme ceux utilisés précédemment (`Clone` et `Copy`). Un trait est déclaré avec le mot-clé `trait`. Pour implémenter un trait dans une structure, le mot-clé `impl` peut aussi être utilisé. Reprenons la structure `PageTable` du *listing* 11. Cette structure contient un tableau et il serait pratique de pouvoir récupérer un élément de ce tableau directement en faisant `table[i]` avec `i` contenant l'indice du tableau. Actuellement, ce n'est pas possible, il faut faire `table.entries[i]`. Les traits `Index` et `IndexMut` permettent ce comportement. Le *listing* 15 montre comment implémenter ces traits dans le cas de `PageTable`.

```
1 impl Index<usize> for PageTable {
2     type Output = u32;
3     fn index(&self, index: usize) -> &u32 {
4         &self.entries[index]
5     }
6 }
7 impl IndexMut<usize> for PageTable {
8     fn index_mut(&mut self, index: usize) -> &mut u32 {
9         &mut self.entries[index]
10    }
11 }
```

Listing 15 – Implémentation de traits pour une structure

3.4.3 Gestion des erreurs

Rust possède tout un mécanisme de gestion des erreurs. Les erreurs sont séparées en deux catégories. Les erreurs récupérables et non récupérables. Quand une erreur est récupérable, le type `Result<T, E>` est utilisé pour gérer l'erreur (en général dans une expression `match`). Dans le cas contraire, la macro `panic!` est appelée. Par défaut quand, un programme panique, Rust réinitialise la pile et nettoie la mémoire avant de quitter. Ce comportement peut être annulé en rajoutant la section du *listing 16* [4].

```
1 [profile.release]
2 panic = 'abort'
```

Listing 16 – Section à ajouter au fichier TOML

3.4.4 Tests unitaires

Rust offre tout un mécanisme de tests unitaires. Ce mécanisme est utilisé avec Cargo en exécutant la commande `cargo test`. Quand cette commande est exécutée, Cargo cherche le module `tests` du projet. Un module en Rust contient des variables, des fonctions et des structures. Il permet de diviser le code en parties plus petites. Un module peut être dans le fichier principal du projet ou dans un fichier différent. Dans le cas des tests unitaires un module basique ressemblerait à celui décrit dans le *listing* suivant.

```
1 #[cfg(test)]
2 mod tests {
3     #[test]
4     fn it_works() {
5         assert_eq!(2 + 2, 4);
6     }
7 }
```

Listing 17 – Module `tests`

L'attribut `#[cfg(test)]` dit à Rust de ne compiler le code que si `cargo test` est exécuté. Lors de l'exécution des tests unitaires, Cargo appelle toutes les fonctions préfixées de l'attribut `test` [4]. Malheureusement, les tests unitaires n'ont pas pu être beaucoup utilisés dans le cadre de ce projet car ils sont exécutés sur la machine hôte. Notre système d'exploitation est compilé sur la machine hôte mais est exécuté sur une autre machine (dans notre cas une machine virtuelle, QEMU). Il n'est donc pas possible d'utiliser ce mécanisme de tests pour tester les différentes parties du système.

3.4.5 *Unsafe* Rust

Nous avons vu que Rust garantit la sécurité de la mémoire à la compilation. Le compilateur ne nous laisse par exemple pas déréférencer un pointeur, modifier une variable statique ou encore appeler du code d'un autre langage. Le problème est que nous utilisons Rust dans le cadre de la programmation système. Rust est utilisé dans ce projet pour créer un système d'exploitation. Il est nécessaire d'accéder à la mémoire sans restrictions et d'appeler du code assembleur. Rust donne la possibilité de supprimer la protection du compilateur avec le mot-clé `unsafe`. Quand un code est `unsafe`, plus aucune protection n'est appliquée. Il est donc très déconseillé de l'utiliser. En effet, en utilisant du code `unsafe`, on perd beaucoup d'avantages de Rust [4].

4 Exécution du *kernel*

4.1 Introduction

Quand un ordinateur est allumé, un signal est envoyé à la carte mère qui démarre l'alimentation. Le processeur démarre alors en mode réel (16 bits). Un signal est ensuite envoyé pour charger le BIOS. Le BIOS est le *firmware* du PC, il est localisé en mémoire flash de la carte mère. Il a pour rôle de vérifier que chaque périphérique est alimenté et qu'il n'y a pas de problème avec la mémoire puis il initialise le *hardware*. Ensuite, le BIOS charge le MBR depuis le premier disque dur. Le MBR est situé dans les 512 premiers octets du disque et il contient le *bootloader* du système qui est un morceau de code exécutable. Le *bootloader* a pour tâche de localiser l'image du *kernel* pour le charger en mémoire. Il passe ensuite le CPU en mode protégé (32 bits) puis exécute le code du *kernel*. La figure ci dessous résume tout ce processus appelé *boot* en anglais [1].

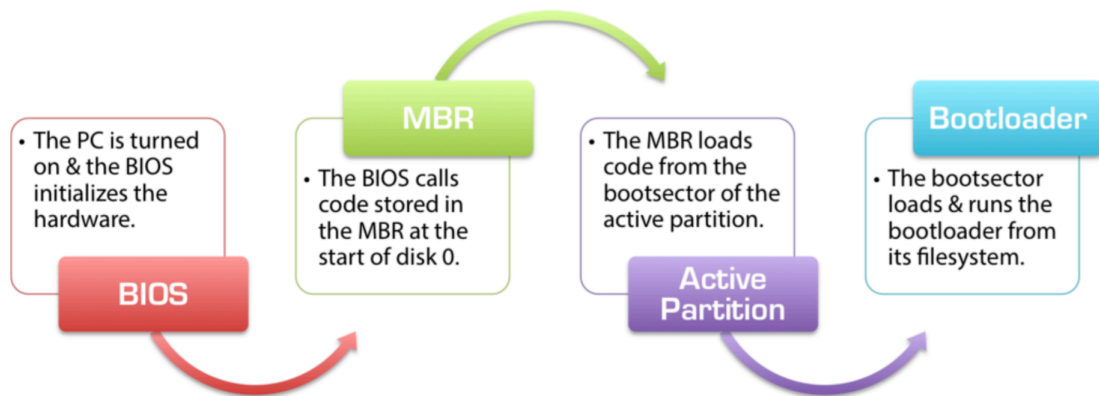


FIGURE 3 – *Boot* d'une machine à base de BIOS

Pour rappel, nous utilisons le logiciel QEMU, qui émule le comportement d'une machine physique. Pour exécuter notre *kernel* sur QEMU, il faut d'abord compiler ses sources en un exécutable au format ELF. Une image ISO est ensuite générée à partir de ce fichier. Cette image doit contenir le *kernel* et le *bootloader* de notre système d'exploitation. Il est possible d'écrire son propre *bootloader* mais c'est une démarche assez complexe. Il existe de nombreux *bootloaders open source*. Celui utilisé dans ce projet est GRUB. L'exécution du *kernel* se déroule donc en deux parties. Il faut dans un premier temps compiler tout le code du *kernel* puis créer un fichier image à partir du *kernel* compilé et de GRUB.

4.2 Compilation du *kernel*

4.2.1 Appel du code Rust depuis l'assembleur

Lorsqu'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 finalement de lier les différents fichiers objets et générer un exécutable [1]. Notre *kernel* est composé de code écrit en assembleur et de code écrit en Rust. Dans le chapitre 2.2, il a été expliqué que du code assembleur est utilisé pour les parties trop bas niveau du *kernel* (utilisation d'instructions spécifiques). Le code assembleur est compilé avec NASM (Netwide Assembler). Ce compilateur génère

des fichiers objets à partir de programmes écrits en assembleur x86. La compilation du code Rust est décrit en détail dans le chapitre 3.2.2. Etant donné que nous voulons lier le code assembleur au code Rust, il faut que ce dernier soit compilé en fichier objet. Ceci peut être fait en rajoutant une section dans le fichier `Cargo.toml` (voir *listing 18*).

```
1 [lib]
2 name = "kernel"
3 path = "src/kernel.rs"
4 crate-type = ["staticlib"]
```

Listing 18 – Section `lib` du fichier `Cargo.toml`

Cette section indique à Xargo de compiler le code en une librairie statique. Une librairie statique est un ensemble de fichiers objets. Le fichier indiqué dans `path` (`kernel.rs`) est le fichier principal du projet. Il contient le point d'entrée dans le code Rust. Quand notre *kernel* est appelé par le *bootloader*, il commence son exécution dans un bout de code assembleur. Ce code est le *bootstrap*. Son rôle est d'initialiser la pile et d'appeler le point d'entrée au code Rust. On doit donc s'assurer que ce point d'entrée sera visible par le code assembleur avant de lier les fichiers objets car dans le cas contraire nous aurons une erreur. Rust permet d'exporter une fonction grâce au mot-clé `extern`. Ci-dessous, le contenu du fichier `kernel.rs`.

```
1 #![feature(lang_items)]
2 #![no_std]
3
4 #[no_mangle]
5 pub extern fn kmain() {}
6
7 #[lang = "panic_fmt"]
8 #[no_mangle]
9 pub extern fn panic_fmt() -> ! {
10     loop {}
11 }
12
13 #[no_mangle]
14 pub extern "C" fn __floatundisf() {
15     loop {}
16 }
```

Listing 19 – Fichier principal du *kernel*

La fonction `panic_fmt()` est utilisée par Rust quand le programme panique. La fonction `__floatundisf()` est utilisée pour faire des calculs avec nombres flottants [11]. Sans librairie standard, il faut les déclarer. On constate que beaucoup d'attributs sont utilisés. Les attributs préfixés de `"#!"` sont appliqués à l'intégralité de la *crate*. Un attribut simplement préfixé d'un `'#'` ne s'applique qu'à la fonction qui lui est associée. L'attribut `no_std` indique au compilateur que la *crate* n'utilise pas la librairie standard. L'attribut `no_mangle` qui précède toutes les fonctions du *listing* indique au compilateur de ne pas modifier le nom de la fonction. C'est une méthode que Rust utilise pour avoir des noms de fonction uniques [6]. Ces fonctions sont appelées par le code assembleur, leurs noms ne doivent pas changer. Enfin, la fonctionnalité `lang_items` permet d'utiliser l'attribut `lang = "panic_fmt"` afin de déclarer la fonction `panic_fmt()`. Au niveau du *bootstrap*, il suffit d'inclure la fonction `kmain()` en écrivant `extern kmain` dans le fichier. Le code Rust peut ainsi être appelé depuis le *bootstrap* en assembleur.

4.2.2 Linking

Nous avons vu que NASM et Xargo sont utilisés pour générer des fichiers objets. Nous avons d'un côté les fichiers objets du code assembleur et de l'autre les fichiers objets du code Rust. Comme expliqué précédemment, ces fichiers doivent être liés en utilisant un *linker*. Le compilateur GCC est utilisé comme *linker* pour notre *kernel*. Pour faire l'édition des liens, GCC utilise un fichier spécifique nommé *linker script*. Si ce fichier n'est pas donné, il en utilise un par défaut. C'est en général ce qui est fait quand on compile un simple programme C. Le *linker script* permet de structurer le code par sections. Notre OS doit avoir une structure bien précise, nous avons donc besoin de notre propre *linker script*. Le *listing 20* contient le *script* utilisé pour ce projet.

```
1 ENTRY(entrypoint)
2
3 SECTIONS {
4     . = 1M;
5     .boot ALIGN(4) : { *(.multiboot) }
6     .stack ALIGN(4) : { *(.stack) }
7     .text ALIGN(4K) : { *(.text*) }
8     .rodata ALIGN(4K) : { *(.rodata*) }
9     .data ALIGN(4K) : { *(.data*) }
10    .bss ALIGN(4K) : { *(COMMON) *(.bss*) }
11 }
```

Listing 20 – *Linker script* du *kernel*

L'appel à `ENTRY` permet de spécifier l'entrée du *kernel*. Pour un simple programme en C l'entrée serait le *main*. Ici, c'est l'entrée de notre *kernel* donc le code du *bootstrap* qui appelle ensuite le code Rust. `SECTION` va dire au linker où placer les parties du code. La table 2 explique la signification de chaque section [1, 12, 13, 14].

Section	Description
<code>.boot</code>	Informations du <i>bootloader</i>
<code>.stack</code>	Pile du <i>kernel</i>
<code>.text</code>	Code du <i>kernel</i>
<code>.rodata</code>	Données en lecture seule (<i>read only data</i> , comme les constantes)
<code>.data</code>	Données en lecture/écriture
<code>.bss</code>	Données non initialisées (comme les variables statiques)

TABLE 2 – Description des sections d'un programme

A noter que les sections commencent avec un *offset* d'un mégaoctet. Il est nécessaire de placer le *kernel* après cette adresse car le premier mégaoctet est une zone réservée [1, 15]. Par exemple, la mémoire vidéo (VRAM) se situe dans cette zone. La commande pour lier les fichiers objets avec le *linker script* est la suivante.

```
$ gcc -T <linker> -static -m32 -ffreestanding -nostdlib <asm> <rust> -o <elf>
```

Ici, `<linker>` est le *linker script*, `<asm>` représente les fichiers objets générés par NASM, `<rust>` la librairie statique générée par Xargo et `<elf>` la sortie au format ELF.

4.3 Boot de l'OS

4.3.1 GRUB

GRUB est un *bootloader* faisant partie du projet GNU, il est donc *open source*. Il utilise le standard *multiboot* qui décrit comment un *bootloader* peut charger un *kernel* sous architecture x86 [16]. GRUB peut donc charger tout OS compatible avec les spécifications de *multiboot*. Notre système étant sous architecture IA-32, il fait partie de cette catégorie. Un autre objectif de GRUB est de permettre de démarrer plusieurs systèmes d'exploitation sur une même machine. Nous avons vu précédemment que le MBR contient le *bootloader* du système. GRUB fait partie de la catégorie des *chain bootloader*, ce qui veut dire que son initialisation se fait par étapes. Le MBR étant une zone mémoire très restreinte (seulement 512 octets), il ne va contenir que la première partie de GRUB. Le BIOS charge donc la première partie depuis le MBR puis chaque étape charge la suivante [1]. Les étapes de l'initialisation de GRUB sont décrites dans la table 3.

Etape (<i>stage</i>)	Description
<i>Stage 1</i>	Contient le code pour charger le <i>Stage 1.5</i>
<i>Stage 1.5</i>	Contient les drivers nécessaires à l'accès au <i>Stage 2</i>
<i>Stage 2</i>	Permet de sélectionner et charger un OS

TABLE 3 – Etapes d'initialisation de GRUB

Une fois ces étapes passées, GRUB passe le CPU en mode 32 bits. Il place ensuite le `multiboot_magic` dans le registre `eax` et l'adresse d'une structure contenant les informations sur le système dans le registre `ebx`. Ces informations peuvent être envoyées au code Rust lors de l'appel à ce dernier. La configuration de GRUB se fait avec les fichiers `menu.lst` et `stage2_eltorito` [1].

4.3.2 Image ISO

Pour exécuter notre *kernel* avec la machine virtuelle QEMU, nous avons besoin de créer un fichier image au format ISO. ISO 9660 est un standard qui définit le système de fichiers utilisé sur un support optique. Pour qu'une image ISO soit *bootable*, il est nécessaire que GRUB soit installé dans les huit premiers kilooctets du disque [1]. Pour générer l'image ISO à partir du *kernel* et des fichiers de configuration de GRUB, l'utilitaire `genisoimage` est utilisé. Ci-dessous, un exemple de création d'une image ISO à partir du *kernel* compilé (`kernel.elf`) et des fichiers de configuration de GRUB (`menu.lst` et `stage2_eltorito`).

Prenons l'arborescence suivante :

```
isofiles
├─ boot
│  └─ grub
```

Le fichier ELF contenant le code du *kernel* ainsi que les fichiers de configuration de GRUB cités plus haut doivent être copiés de manière à obtenir l'arborescence qui suit :

```
isofiles
├── boot
│   ├── grub
│   │   ├── menu.lst
│   │   └── stage2_eltorito
│   └── kernel.elf
```

L'image ISO peut maintenant être créée avec la commande ci-dessous :

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

Cette commande générera une image ISO *bootable* nommée `os.iso` [1]. Pour charger cette image avec QEMU et exécuter notre *kernel* une dernière commande est utilisée :

```
$ qemu-system-i386 -cdrom os.iso
```

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 également de 4Go. 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 tâche est exécutée, elle est chargée en mémoire et est définie par la paire base et limite. La base est son adresse physique dans la RAM et la limite est sa taille. La figure 4 donne un exemple d'adressage de plusieurs processus [1].

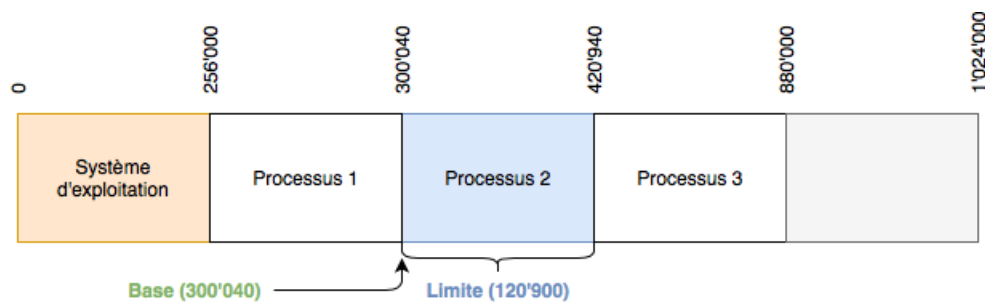


FIGURE 4 – Exemple d'adressage mémoire

Une tâche possède son propre espace d'adressage dit virtuel. Pour le processus 2 de la figure 4, l'adresse 0 est en fait à l'adresse physique 300'040. Il y a donc besoin de traduire l'adresse virtuelle en adresse physique. C'est là qu'entre en jeu le MMU (*Memory Management Unit*). Le MMU est un dispositif matériel permettant de faire cette translation d'adresses. A chaque référencement mémoire, il va convertir l'adresse virtuelle en adresse physique et regarder si elle ne dépasse pas la limite du processus. Le MMU permet donc aussi de protéger la mémoire car il va empêcher toute référence à une zone extérieure au processus (voir figure 5) [1].

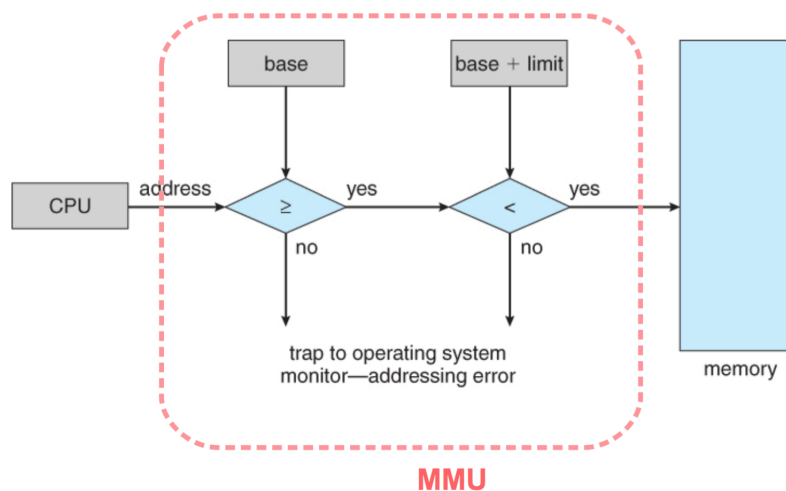


FIGURE 5 – Protection mémoire avec un MMU

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.

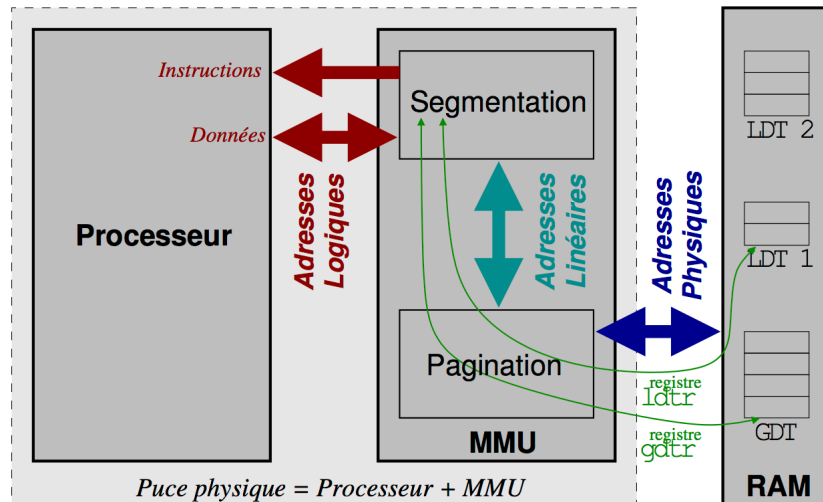


FIGURE 6 – Translation d'adresse

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 6. 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) [17].

5.2 Segmentation

5.2.1 Principe général

Comme expliqué précédemment, 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 [18].

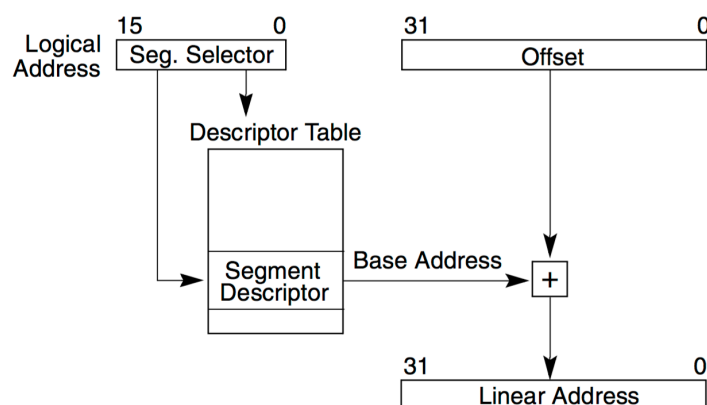


FIGURE 7 – Conversion d’une adresse logique en adresse linéaire

La figure 7 résume bien la conversion d’adresse logique 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 8) [1].

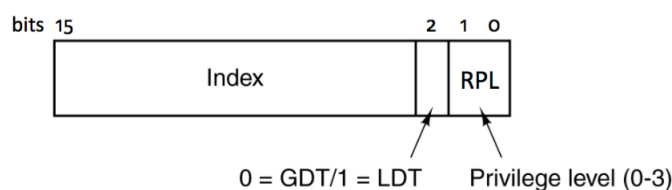


FIGURE 8 – Structure d’un sélecteur de segment

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, 19].

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

TABLE 4 – Registres de segment

En mode protégé (32-bits), ces registres doivent pointer sur des descripteurs de segment de la GDT. Au minimum les trois premiers registres décrits doivent être utilisés en mode protégé (CS, DS, et SS). Les opérations adressant le code (décodage des instructions en mémoire, sauts, etc...) référencent le descripteur de segment sur lequel pointe le registre CS. Les opérations adressant les données (adressage de variables ou d’adresses mémoires) référencent le descripteur de segment sur lequel pointe le registre DS. Les opérations adressant la pile (**push** et **pop**) référencent le descripteur de segment sur lequel pointe le registre SS. 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 traduire 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 descripteurs est composée, comme son nom l'indique, de descripteurs. Chaque descripteur décrit une zone mémoire qui est définie 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 9 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 utilisée dans la version actuelle de l'OS.

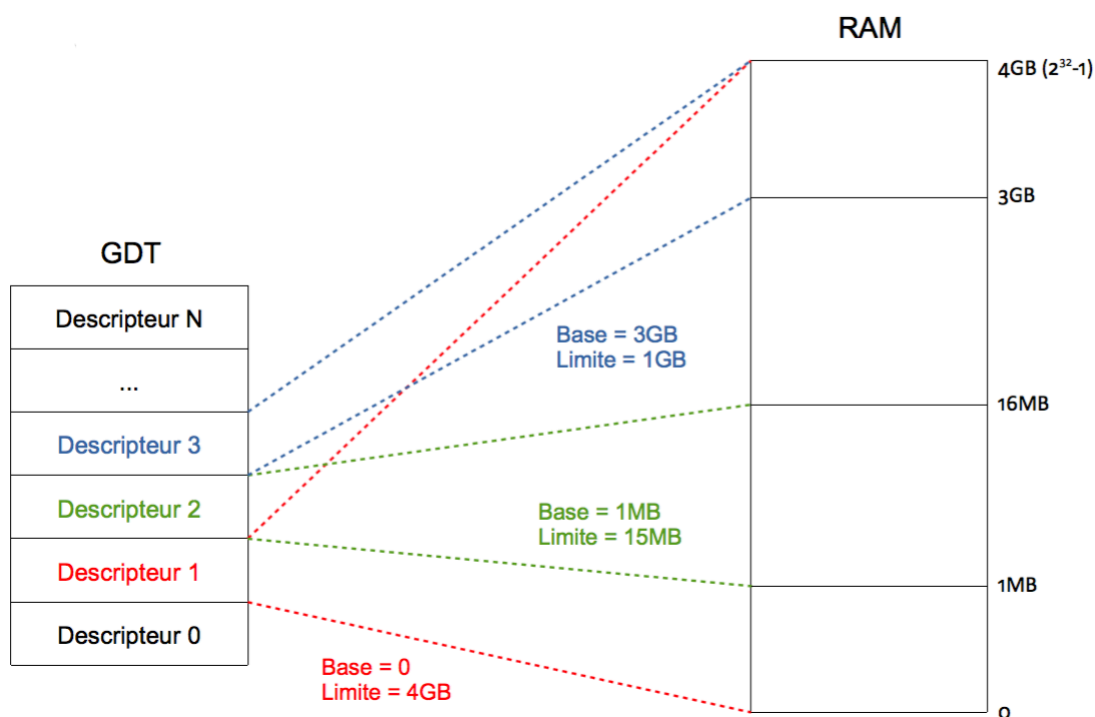


FIGURE 9 – Exemple d'une GDT

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 privilèges, droit d'écriture/lecture, ...). Voir la figure 10 pour plus de détails [1, 20].

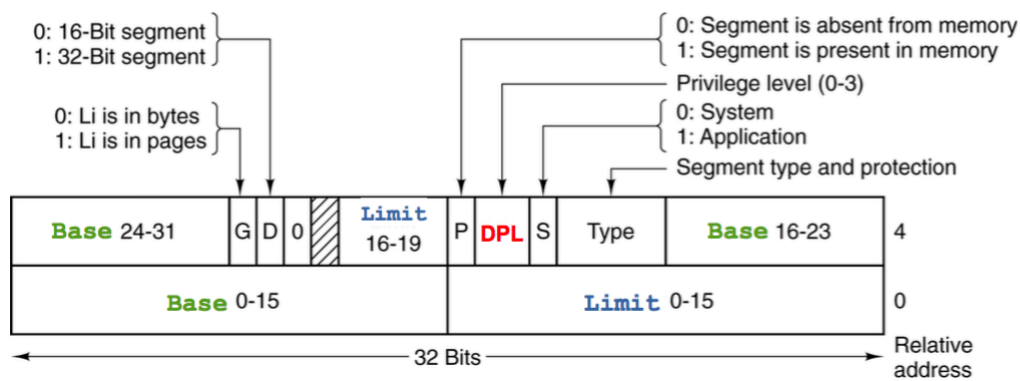


FIGURE 10 – Structure d’une entrée dans la GDT

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 tous les modèles de segmentation), un segment de code couvrant toute la mémoire et un segment de données couvrant aussi toute la mémoire. Les segments de code et de données adressent ainsi les mêmes zones mémoire. On verra par la suite que d’autres entrées ont été ajoutées à la GDT pour la gestion des tâches.

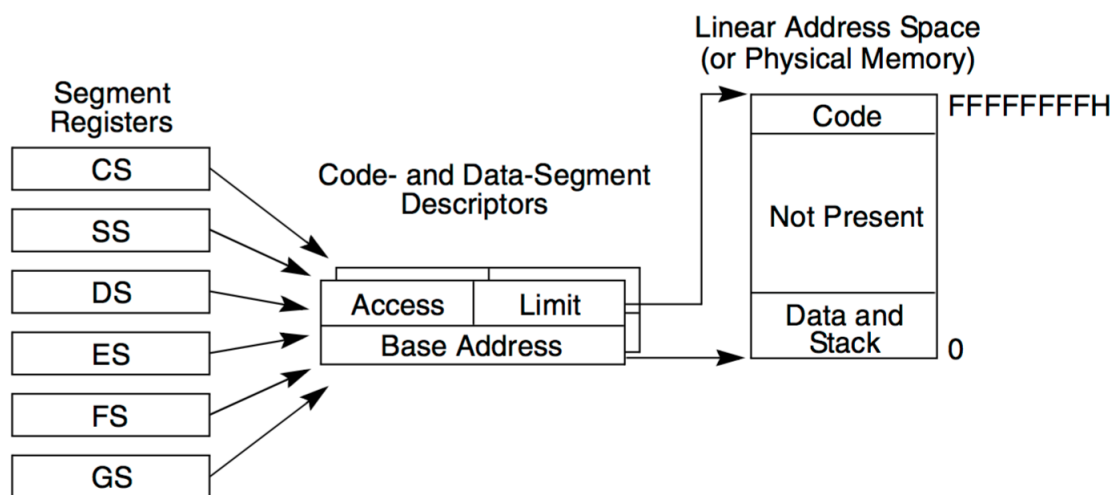


FIGURE 11 – Modèle de segmentation de type *flat*

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 réelle de $0x100000 \times 0x1000$ 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.

```

1 fn new(base: u32, limit: u32, type: u8, s: u8, d: u8, g: u8, dpl: u8) -> GdtEntry;
2
3 pub fn make_code_segment(base: u32, limit: u32, dpl: u8) -> GdtEntry {
4     GdtEntry::new(base, limit, 0xB, 0x1, 0x1, 0x1, dpl)
5 }
6
7 pub fn make_data_segment(base: u32, limit: u32, dpl: u8) -> GdtEntry {
8     GdtEntry::new(base, limit, 0x3, 0x1, 0x1, 0x1, dpl)
9 }

```

Listing 21 – Constructeurs d’une entrée dans la GDT

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 [18]. 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 [18]. 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 12 [20].

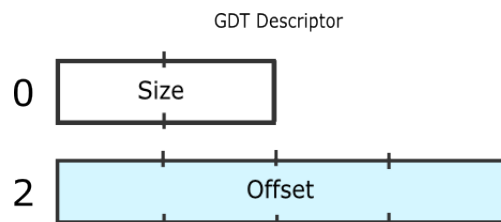


FIGURE 12 – Descripteur de GDT

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 précédemment chargée. Pour récupérer l’index d’un segment dans la GDT à partir de l’index de son descripteur, il faut faire un décalage à gauche de 3 bits (voir figure 8). Prenons 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 ($2 \ll 3 = 16$). 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 décalage.

5.3 Pagination

5.3.1 Principe général

La pagination est une autre 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 [21]. Quand une tâche fait référence à une adresse logique en mémoire, cette adresse est convertie en adresse linéaire grâce au mécanisme de segmentation et c'est le mécanisme de pagination qui permet de traduire cette adresse linéaire en adresse physique (comme vu précédemment dans la partie sur la segmentation). Quand la pagination est activée, l'adresse linéaire est divisée en deux parties lorsque des pages de 4Mo sont utilisées et en trois parties lorsque des pages de 4Ko sont utilisées. Le *kernel* développé utilise des pages de 4Ko, une adresse linéaire est donc sous la forme suivante :

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

On dit que cette pagination est une pagination à trois niveaux. En général, une pagination à trois niveaux est utilisée mais il peut exister des systèmes utilisant plus ou moins de niveaux. Le système d'exploitation doit créer un répertoire de pages (*Page Directory*) et au moins une table 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 [22].

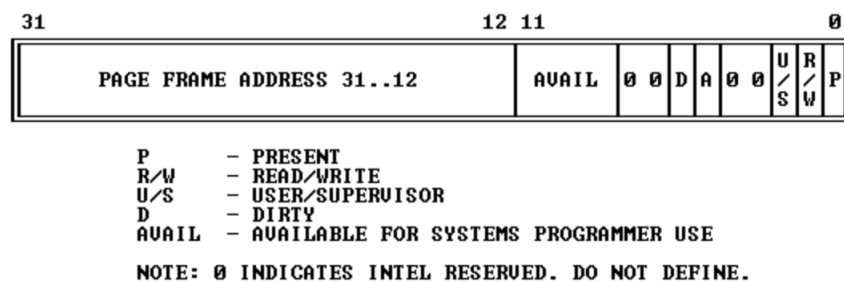


FIGURE 13 – Structure d'une *Page Entry*

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 14 résume bien ce mécanisme [18]. 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 [23].

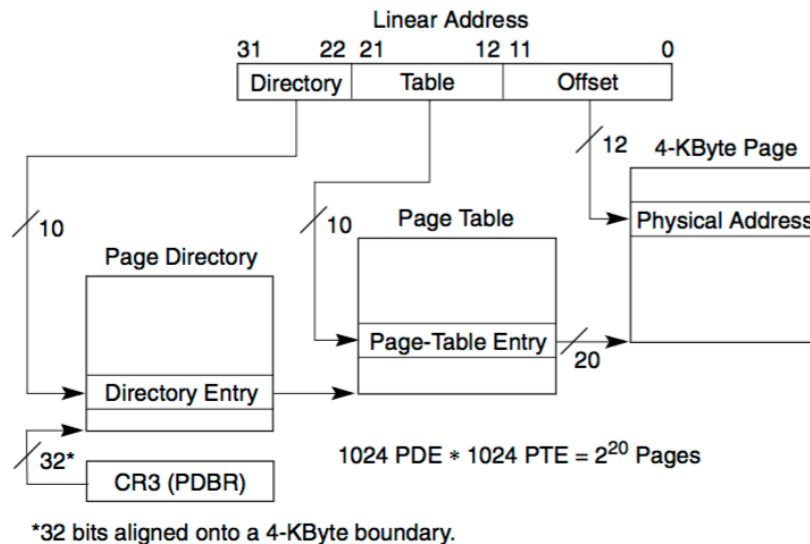


FIGURE 14 – Exemple de pagination à 3 niveaux

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. Grâce à la pagination, ceci peut se faire assez simplement, il suffit de compléter le répertoire de pages ainsi que ses tables de pages correctement. Pour rappel, le *kernel* commence à l'adresse 0x100000 (1Mo) mais il faut aussi rendre accessible le premier Mo de RAM. Il faut donc déplacer les adresses physiques allant de 0x0 à la fin du *kernel* (qui n'est pas fixe). Dans un premier temps, le *linker* doit être modifié de cette manière :

```

1  SECTIONS {
2      /* Low memory Kernel */
3      . = 0x00100000;
4      .boot ALIGN(4) :      { *(.multiboot) }
5      .low_text ALIGN(4K) : { *(.low_text) }
6      .low_data ALIGN(4K) : { *(.low_data) }
7      .low_bss ALIGN(4K) :  { *(.low_bss) }
8      /* 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 }

```

Listing 22 – *Linker script* du *higher-half kernel*

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 précédemment, 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 schématisé dans la figure 15.

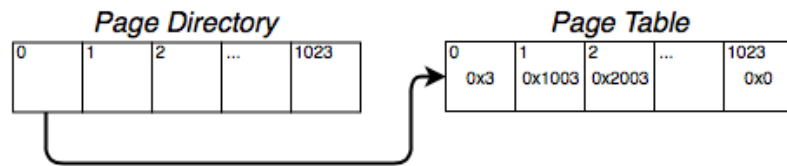


FIGURE 15 – Répertoire de pages adressant le *kernel* au début de la RAM

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 13). 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 décalage 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, $0xC0000000 \gg 22 = 0x300$ (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.

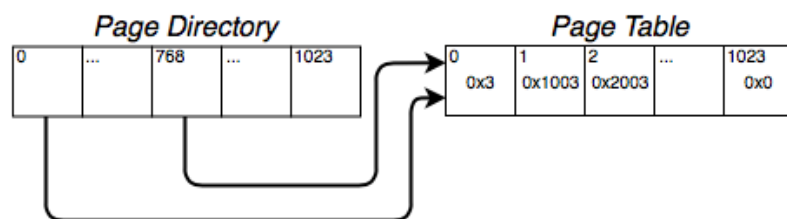


FIGURE 16 – Répertoire de pages adressant le *kernel* à la fin de la RAM

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 la 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 étaient déclarées statiquement se retrouvant donc dans la zone mémoire du *kernel*, plus précisément dans le segment bss (voir 4.2.2). 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 zones mémoires 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.

```
1 . += 0xC0000000;  
2 ...  
3 kernel_end = .;
```

Listing 23 – Modification apportée au *linker*

```
1 extern kernel_end  
2 get_kernel_end:  
3     mov eax, kernel_end  
4     ret
```

Listing 24 – Code assembleur rendant accessible l'expression `kernel_end`

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* alignée 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 utilise 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.

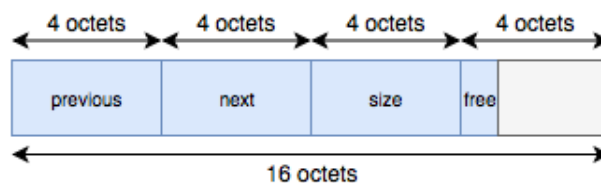


FIGURE 17 – Entête d'un bloc de mémoire dans le tas

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 bloc 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 18). 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.

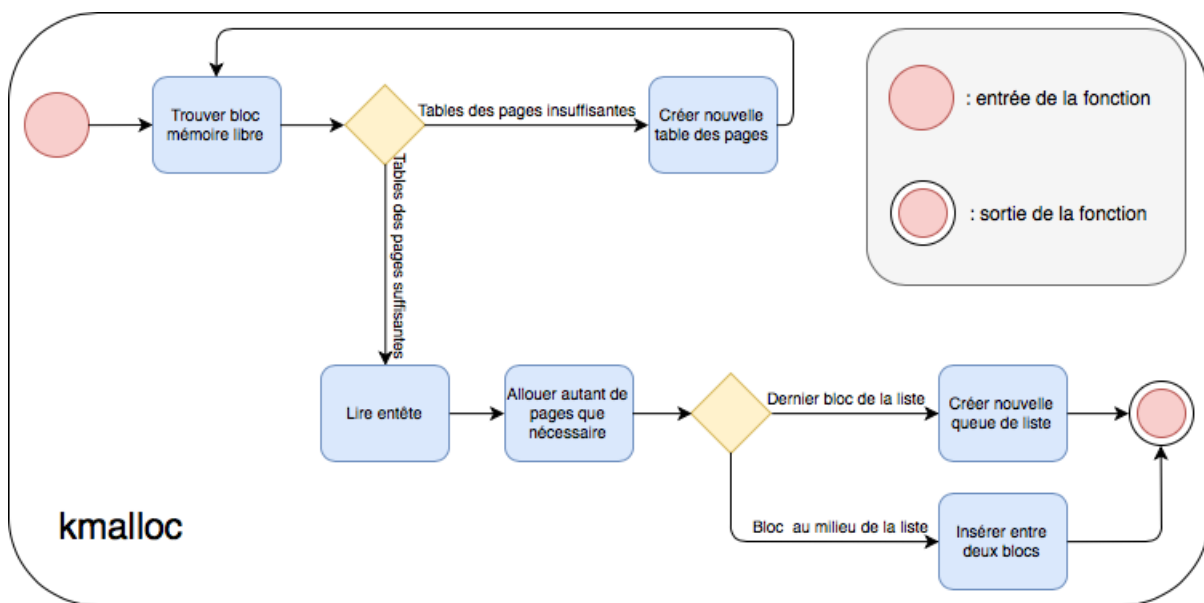


FIGURE 18 – Algorithme utilisé pour l'allocation dynamique dans le *kernel*

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 supposer 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 première entrée de la liste et par la suite de la parcourir.

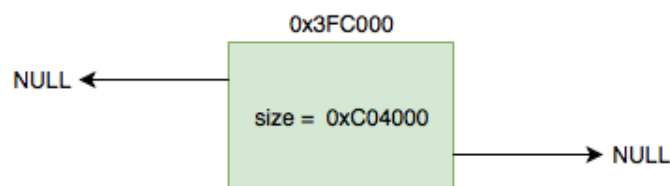


FIGURE 19 – Etat initial de la chaîne d'entêtes

Si le *kernel* a besoin d'allouer une nouvelle page maintenant, la fonction 18 va être appelée qui va parcourir les blocs libres. Dans ce cas précis, l'intégralité du tas est disponible donc la fonction va simplement créer un nouveau bloc. Dans la figure 20, les blocs libres 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 compter l'entête de 16 octets, il faudra toujours allouer une page de plus.

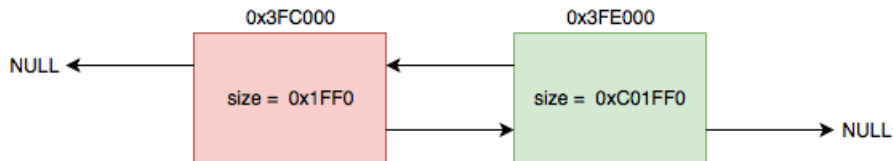


FIGURE 20 – Allocation d'une page

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 18, la fonction d'allocation va se charger 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 21, 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.

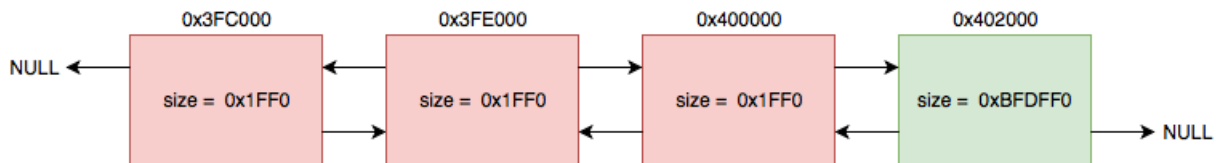


FIGURE 21 – Allocation d'une page et d'une table des pages

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 20 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. Ils 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 périphérique à partir de registres de contrôle (par exemple avec un contrôleur de disque) [24]. Étant 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 spéciales pour accéder aux PIO. Ces instructions sont les instructions `IN` et `OUT`. `IN` permet de lire tandis que `OUT` permet d'écrire. À 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 :

Écrire 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 réservée 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 événements 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 réservées aux exceptions (voir figure 22) [1, 18].

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 <i>n</i> 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 22 – Table des interruptions et exceptions sur IA-32

Comme vu précédemment, 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 contenant la routine d'interruption correspondant au numéro d'interruption logicielle spécifiée. 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 synchrones étant donné qu'elles sont exécutées par le processeur, contrairement aux interruptions matérielles qui sont asynchrones.

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 événement 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 contrôleur d'interruption (PIC). Le PIC va faire correspondre une IRQ à un numéro d'interruption (voir figure 23). 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	Interrupt
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 23 – Table de correspondance des IRQs

En comparant la figure 22 avec la figure 23, 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 [25].

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 [18] décrits dans la figure 24. 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.

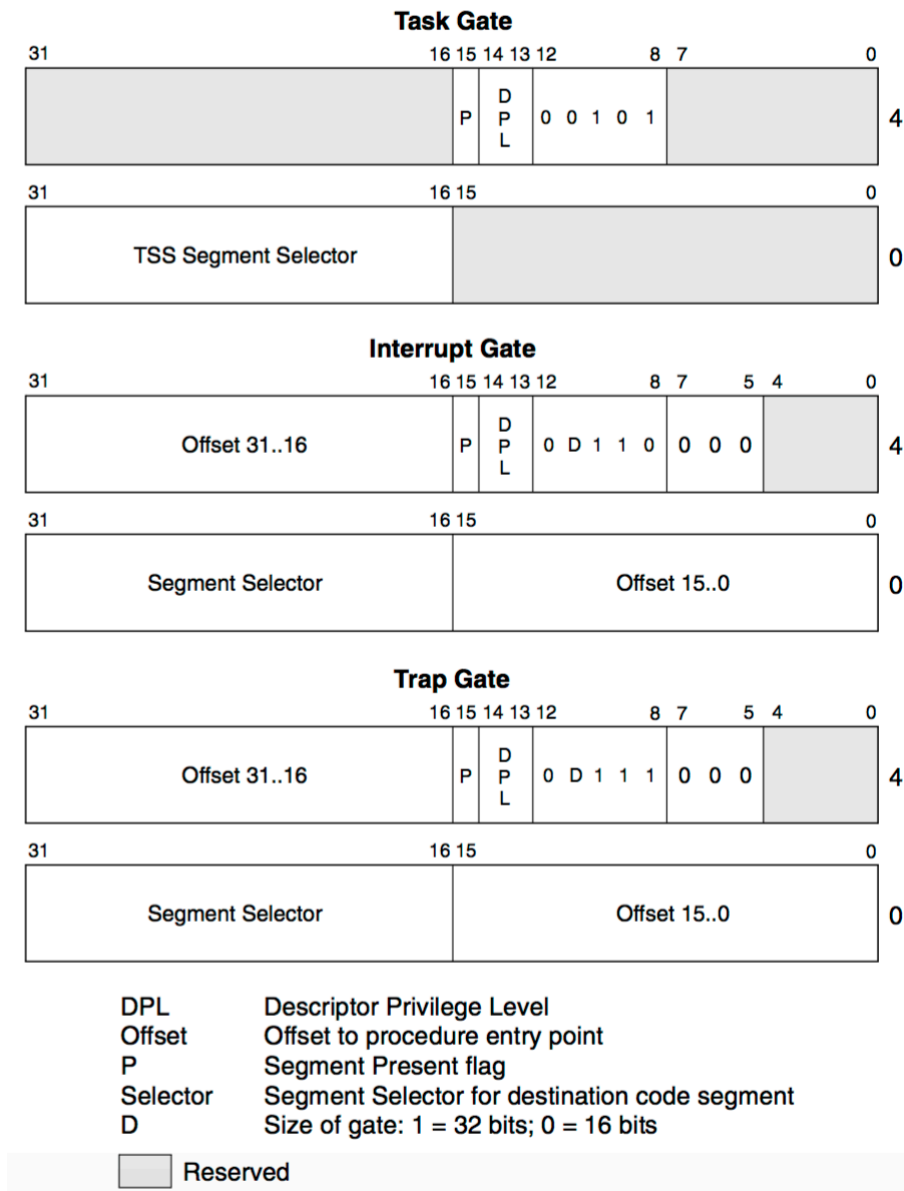


FIGURE 24 – Différents types de descripteur d'interruption

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

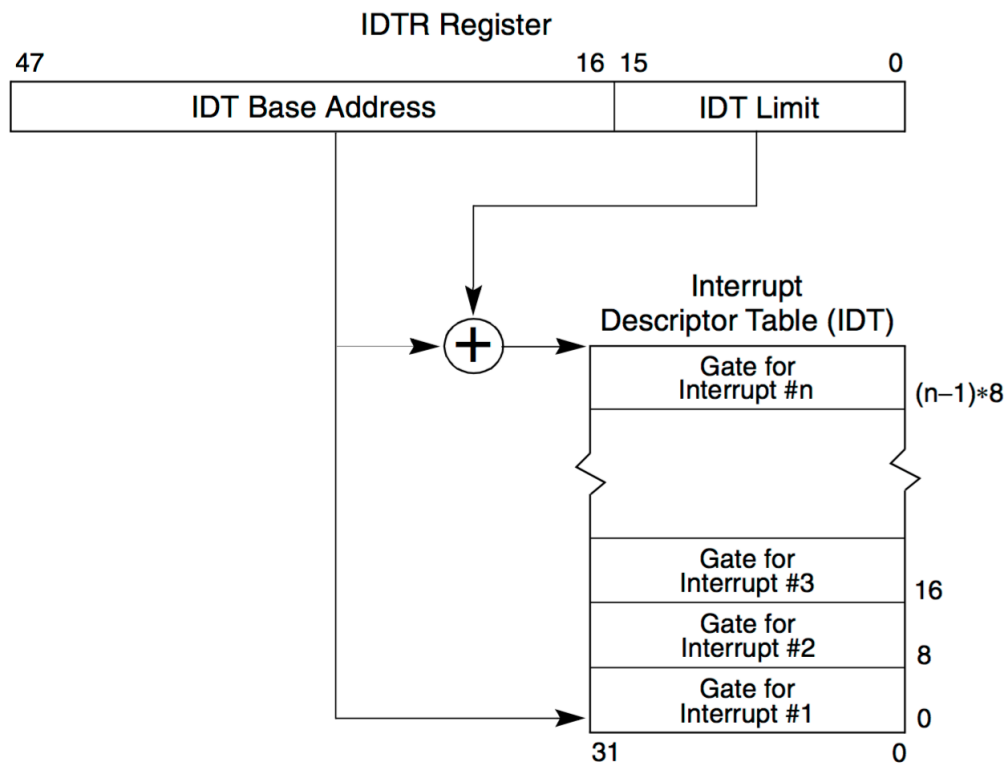


FIGURE 25 – Relation entre le registre IDTR et l'IDT

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 nécessaire 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 réservée 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 26) [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 27 [26].

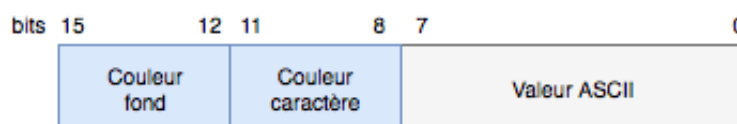


FIGURE 26 – Structure d’un caractère en mode texte VGA

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 27 – 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 [27]. 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 registres 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 spécifiée 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).

```

1 static mut SCREEN: Screen = Screen {
2     buffer: 0xC00B8000 as *mut _,
3     attribute: ColorAttribute::new(Color::White, Color::Black),
4     cursor_x: 0,
5     cursor_y: 0
6 };

```

Listing 25 – Structure Screen

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 variables. La macro **print** possède un nombre d'arguments variables, nous pouvons donc implémenter cette macro ainsi que la macro **println** comme suit :

```

1 macro_rules! print {
2     ($($arg:tt)*) => (vga_write_fmt(format_args!($($arg)*)));
3 }
4
5 macro_rules! println {
6     () => (print!("\n"));
7     ($fmt:expr) => (print!(concat!($fmt, "\n")));
8     ($fmt:expr, $($arg:tt)*) => (print!(concat!($fmt, "\n"), $($arg)*));
9 }

```

Listing 26 – Implémentation des macros **print** et **println**

A noter que la fonction **vga_write_fmt** est simplement une fonction interfaçant la méthode **write_fmt** appliquée à la structure **Screen** 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 mesurer 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`.

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

Listing 27 – Programmation du *timer* en assembleur

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. Ce compteur est incrémenté à chaque fois qu'une interruption a lieu sur l'IRQ0. Comme pour la structure statique permettant d'interfacer l'affichage texte VGA, toute modification de celle du *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).

```
1 pub fn sleep(ms: u32) {
2     let duration = get_ticks() + (ms * get_freq() / 1000);
3     loop {
4         if get_ticks() >= duration {
5             break;
6         }
7         halt();
8     }
9 }
```

Listing 28 – Implémentation de la fonction `sleep`

Ici, `get_ticks` et `get_freq` sont des fonctions qui renvoient les attributs `ticks` et `freq` de la structure statique précédemment décrite. 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 architecture 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 28). 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*.

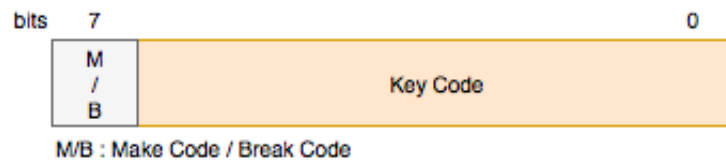


FIGURE 28 – Structure d'un *scan code*

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 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*.

```
1 pub fn keyboard_handler() {
2     let state = inb(0x64) & 1;
3     if state == 1 {
4         let key = inb(0x60);
5         if key >> 7 == 0 {
6             buffer_write(KEY_MAP[key as usize])
7         }
8     }
9 }
```

Listing 29 – Routine d'interruption du clavier

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 contrôleur de disque dur. Actuellement, tout le *kernel* et ses dépendances 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-system-i386 -cdrom kernel.iso -hda fs.img
```

L'option `-cdrom` spécifie l'image ISO contenant le *kernel* (voir partie 4.3.2) 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 fallu 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). À 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.

Secteur de boot	Table d'allocation de fichiers (FAT)	Répertoire racine	Données des fichiers
-----------------	--------------------------------------	-------------------	----------------------

FIGURE 29 – Structure d'un système de fichiers de type FAT

La figure 29 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 réservés 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)

TABLE 5 – Structure du *superblock*

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

TABLE 6 – Structure d'une entrée dans l'espace de métadonnées

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, on peut facilement accéder à un fichier dans le disque. Les informations contenues 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 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.

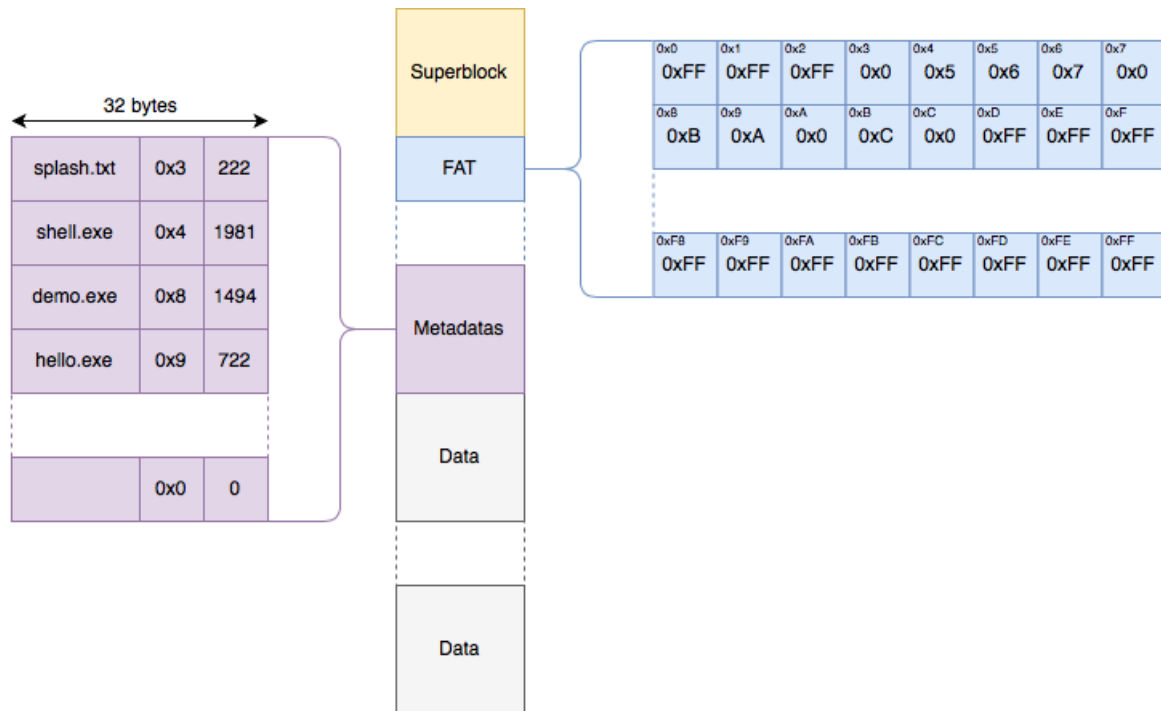


FIGURE 30 – Système de fichiers de l'OS

La figure 30 donne un exemple de contenu du 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 (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. Ils manipulent 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 31. 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).

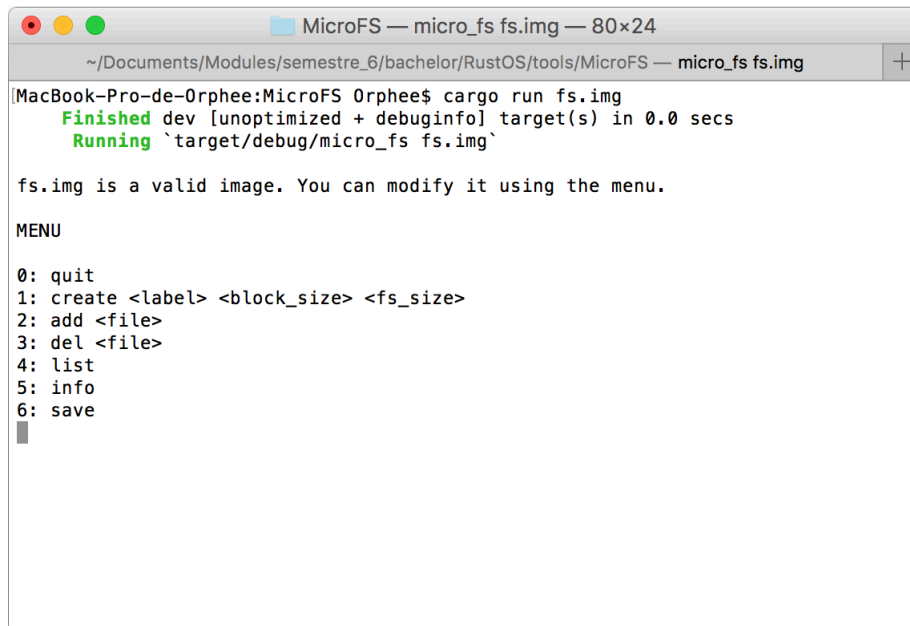


FIGURE 31 – Menu du gestionnaire du système de fichiers

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 contrôleur 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 [28]. Deux fonctions ont été implémentées pour communiquer avec le contrôleur de disque dont les prototypes sont ci-dessous.

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

Listing 30 – Prototypes des fonctions d’écriture/lecture dans le disque dur

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 réalisation 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 utilisateurs

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].

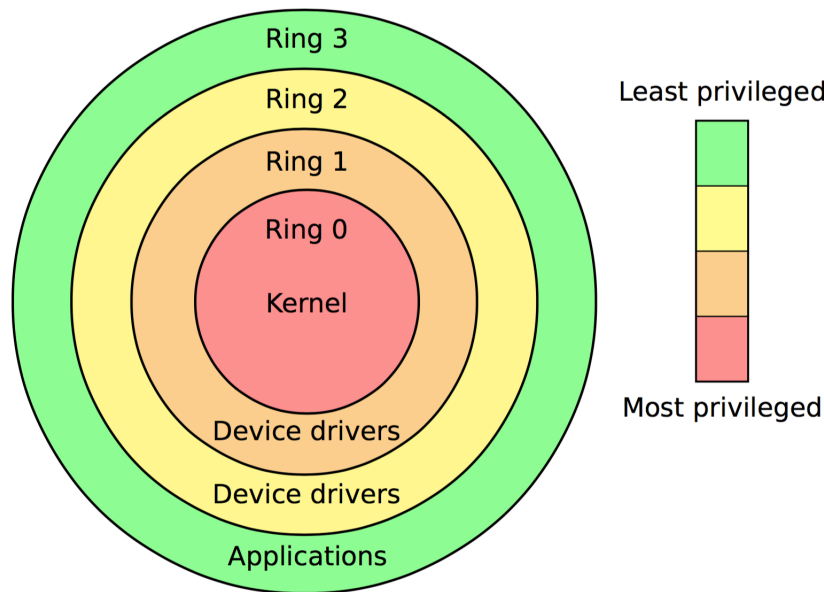


FIGURE 32 – Niveaux de privilèges sur architecture IA-32

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 *kernel* 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 utilisateurs 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 segments 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 33) [18].

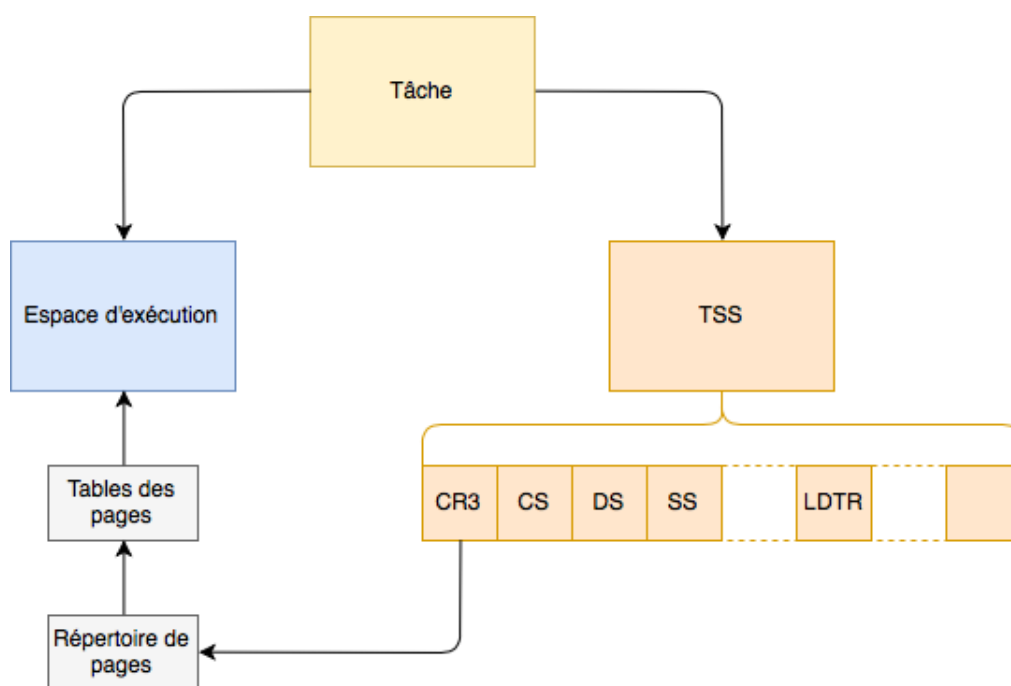


FIGURE 33 – Structure d'une tâche avec la pagination

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 fournit 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 comme 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.

```
1 pub struct Tss {
2     previous_task_link : u16, reserved0      : u16,
3     esp0               : u32,
4     ss0               : u16, reserved1      : u16,
5     esp1               : u32,
6     ss1               : u16, reserved2      : u16,
7     esp2               : u32,
8     ss2               : u16, reserved3      : u16,
9     cr3               : u32,
10    eip               : u32, eflags          : u32, eax: u32, ecx: u32, edx: u32,
11    ebx               : u32, esp            : u32, ebp: u32, esi: u32, edi: u32,
12    es                : u16, reserved4      : u16,
13    cs                : u16, reserved5      : u16,
14    ss                : u16, reserved6      : u16,
15    ds                : u16, reserved7      : u16,
16    fs                : u16, reserved8      : u16,
17    gs                : u16, reserved9      : u16,
18    ldt_selector       : u16, reserved10    : u16,
19    reserved11         : u16,
20    iomap_base_addr    : u16
21 }
```

Listing 31 – Champs de la structure TSS

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.

8.3 Appels systèmes

Les appels systèmes sont des fonctions exposées aux applications utilisateurs 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 au 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.

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

Listing 32 – Entrée dans l'IDT pour les appels systèmes

La structure des entrées de l'IDT est décrite dans la figure 24. 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 une 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`).

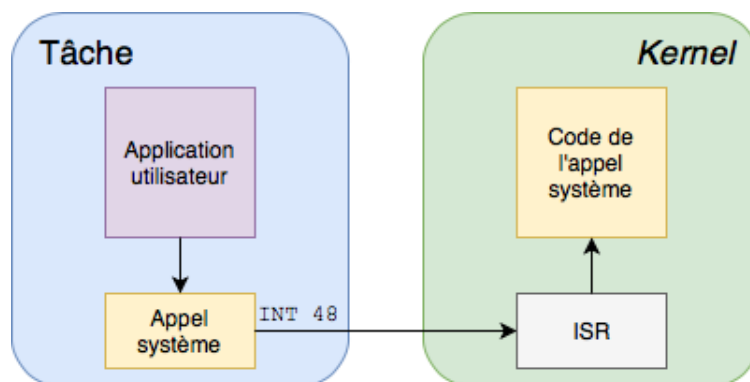


FIGURE 34 – Fonctionnement des appels systèmes

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'allouer de la mémoire en mode *kernel*. Les tâches utilisateurs 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 implémentées les tâches dans une version précédente de l'OS. Le problème qu'il y a avec l'utilisation de structures statiques est que la même quantité de mémoire va être utilisée pour une application faisant 200 octets et une application faisant 200'000 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 36. 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 possibles les appels systèmes (comme expliqué dans la partie précédente).

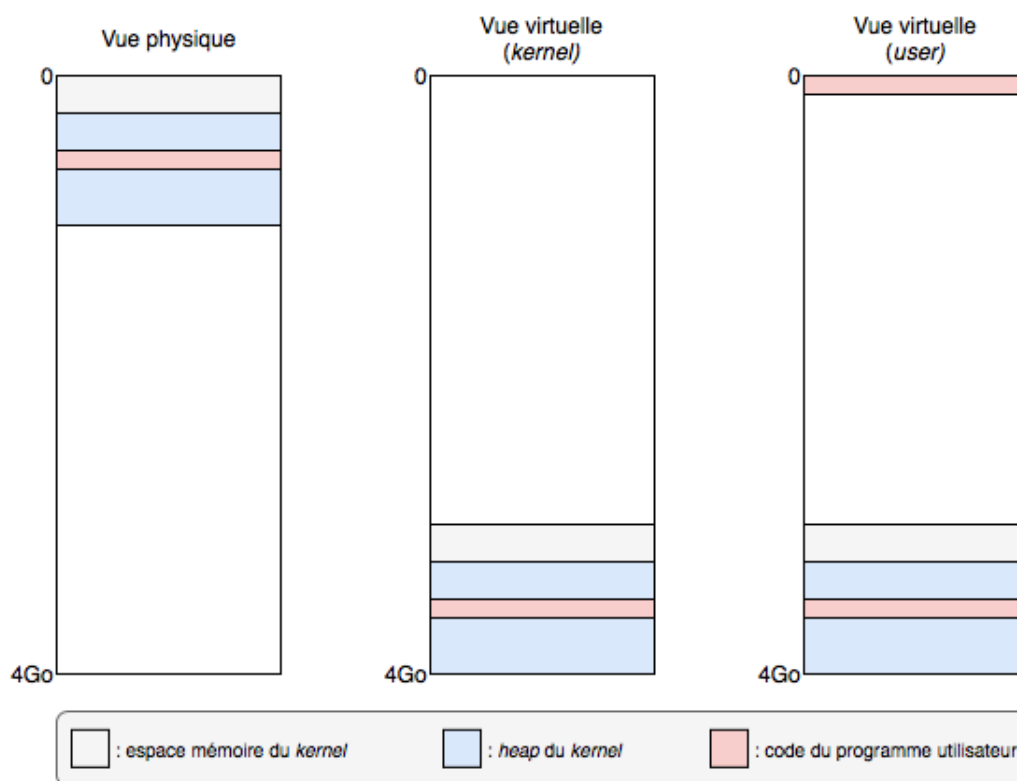


FIGURE 35 – Vues de la mémoire après allocation d'un code utilisateur

8.5 Librairie système et applications

8.5.1 Compilation d’une application utilisateur

Le *kernel* pouvant charger et exécuter des programmes, les appels systèmes étant mis en place et l’allocation dynamique fonctionnelle en mode *user*, des programmes utilisateurs peuvent maintenant être écrits. Ces applications sont compilées séparément du *kernel* avec un *linker* qui leur est propre. D’ailleurs, une application peut être codée dans le langage voulu tant que le *linker* fourni est utilisé. Un programme utilisateur est compilé en format binaire. Sous ce format, le code du programme commence directement à l’adresse 0x0. Il n’y a pas d’entête comme avec le format ELF. Ci-dessous, les lignes importantes du *linker* pour compiler en format binaire. Ce *linker* appelle une fonction assembleur très simple qui appelle la fonction `main` du programme [1].

```
1 OUTPUT_FORMAT("binary")
2 SECTIONS {
3     . = 0x0;
4     ...
5 }
```

Listing 33 – *Linker* pour un exécutable en format binaire

Le langage rust a été utilisé pour développer une librairie système et quelques applications. La compilation de rust est légèrement différente que pour le *kernel* car d’une part nous sommes dans un format binaire, et ensuite il faut se défaire de tous les mécanismes de protection de rust qui prennent une place considérable dans l’exécutable. Pour compiler un programme rust en fichier objet destiné à un format binaire, le *flag* `-C relocation-model=static` doit être passé au compilateur. Par défaut, le compilateur cherche la `GLOBAL_OFFSET_TABLE` qui n’existe pas lorsque le format de l’exécutable est un format binaire. Ce *flag* va dire au compilateur d’ignorer ce mécanisme [29]. De cette façon, un programme utilisateur peut être compilé mais sa taille sera excessivement grande (dans l’ordre des 100’000 octets). Ceci est dû à plusieurs raisons. Tout d’abord, rust est compilé par défaut sans aucune option d’optimisation. Le *flag* `-C opt-level=3` fixe un niveau d’optimisation de 3 (plus haut niveau d’optimisation). Un peu plus d’optimisation peut être fait en disant à xargo de compiler en mode *release* et non *debug* avec l’option `--release`. Le niveau d’optimisation a un impact sur le code écrit mais ce sont les librairies qui prennent le plus de place dans un exécutable. Pour rappel, nous compilons sans librairie standard mais la librairie `core` est tout de même utilisée. Quand on compile un code rust, l’intégralité de cette librairie est compilée aussi, même si certaines parties sont inutiles. Pour résoudre ce problème il faut cette fois-ci modifier le fichier `cargo.toml` et lui rajouter les lignes suivantes [30].

```
1 [profile.release]
2 lto = true
3 panic = 'abort'
```

Listing 34 – Options ajoutées au fichier `cargo.toml`

A noter que les *flags* passés au compilateur de rust (avec l’option `-C`), sont appliqués au compilateur `rustc` et non à xargo. Pour dire à xargo (ou cargo) de rajouter des options lors de la compilation, il faut mettre ces options dans la variable d’environnement `RUSTFLAGS`. Comme pour le *kernel*, GCC est utilisé pour *linker* les fichier objets en un exécutable.

8.5.2 Librairie système

Pour faciliter le développement d'applications pour l'OS, une librairie système a été développée. Cette dernière agit comme une mini librairie C offrant les fonctionnalités de base à un programme utilisateur. Cette librairie offre un niveau d'abstraction plus élevé cachant l'utilisation d'appels systèmes à l'utilisateur. Ainsi, aucune application n'a besoin de faire appel des appels systèmes. C'est la librairie système qui s'en charge. La librairie développée offre les fonctionnalités suivantes.

Fonction	Description
<code>print/println</code>	Macros permettant d'afficher une chaîne de caractères formatée
<code>clear</code>	Efface l'écran
<code>puts</code>	Affiche une chaîne de caractères
<code>putc</code>	Affiche un caractère
<code>exec</code>	Exécute un programme utilisateur contenu dans le disque dur
<code>keypressed</code>	Indique si une touche du clavier a été appuyée
<code>getc</code>	Fonction bloquante. Retourne la touche appuyée par l'utilisateur
<code>file_stat</code>	Retourne les métadonnées d'un fichier
<code>file_open</code>	Ouvre un fichier donné en paramètre et retourne son descripteur
<code>file_close</code>	Ferme un fichier à partir de son descripteur
<code>file_read</code>	Lit les octets d'un fichier depuis son descripteur
<code>file_seek</code>	Avance le curseur d'un fichier depuis son descripteur
<code>file_iterator</code>	Retourne un itérateur des fichiers du disque dur
<code>file_next</code>	Itère sur le prochain fichier du disque
<code>get_ticks</code>	Retourne l'état actuel du <i>timer</i>
<code>sleep</code>	Arrête l'exécution pendant un temps donné
<code>set_cursor</code>	Fixe la position du curseur sur l'écran
<code>get_cursor</code>	Retourne la position du curseur sur l'écran
<code>cursor_disable</code>	Active ou désactive le curseur
<code>copy_scr</code>	Copie un <i>frame buffer</i> sur l'écran
<code>malloc</code>	Alloue de la mémoire
<code>free</code>	Libère de la mémoire

TABLE 7 – Fonctions proposées par la librairie système

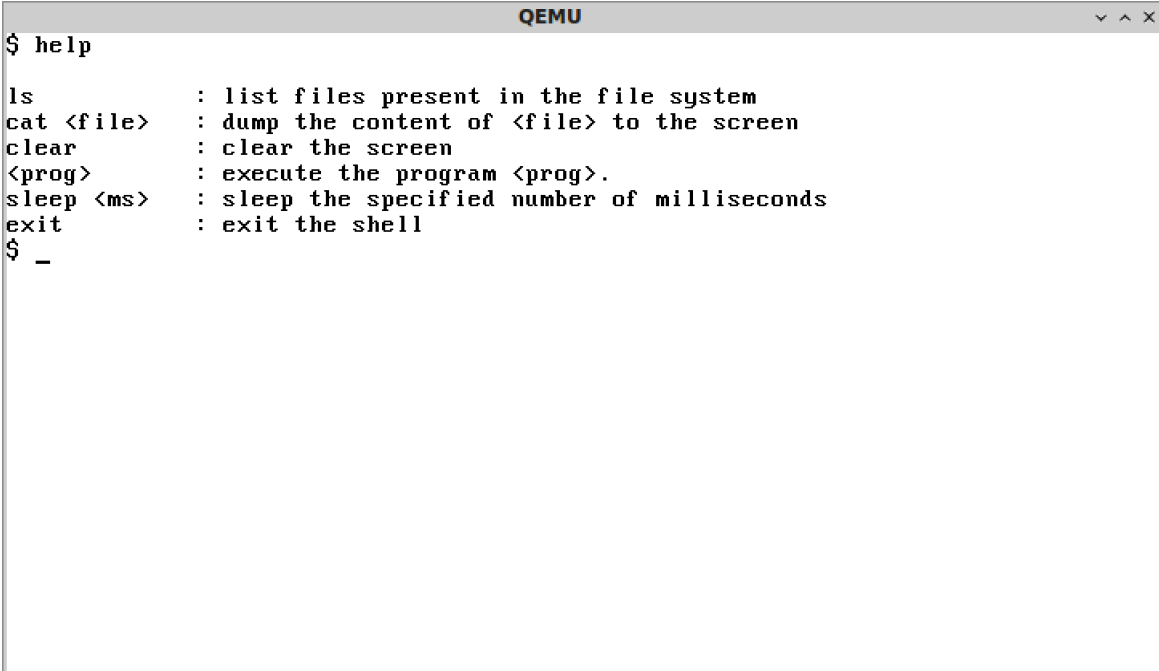
La fonction `malloc` utilise un appel système allouant une nouvelle page dans le répertoire de pages de la tâche. Cet appel système appelle lui même la fonction `umalloc` décrite dans la partie 8.4. la fonction `malloc` gère la mémoire de la même manière que le *kernel*. Un tas est utilisé au niveau utilisateur. Quand un utilisateur alloue de la mémoire, `malloc` va regarder s'il y a assez de place dans la dernière page allouée. Si c'est le cas, aucun appel système n'est appelé, une partie de la dernière page va être réservée. Dans le cas contraire, l'appel système pour allouer une page est appelé autant de fois que nécessaire. Cette gestion utilise des entêtes qui fonctionnent exactement comme ceux du *heap* du *kernel* décrits en 5.4. La seule différence est que l'espace alloué n'est pas aligné sur 4096 octets (la taille d'une page) car l'utilisateur aura tendance à allouer des espaces plus petits et cela produirait une trop grande perte de mémoire. De plus, si l'espace alloué était aligné sur la taille d'une page, un appel système aurait lieu à chaque allocation ce qui n'est pas souhaitable.

8.5.3 Applications développées

Shell

Actuellement, les applications utilisateurs ne peuvent être appelées que depuis le *kernel*. Ceci n'est pas très pratique d'un point de vue utilisateur. Afin d'avoir un système d'exploitation un peu plus interactif, un shell très simple a été développé. Ce shell permet de lire les fichiers du disque dur et d'exécuter d'autres applications utilisateurs. Le shell implémenté propose les fonctionnalités suivantes.

- `ls` : liste les fichiers du disque dur
- `cat <file>` : affiche le contenu d'un fichier
- `clear` : efface l'écran
- `<prog>` : exécute un programme
- `sleep <ms>` : attend pendant un temps donné
- `exit` : sort du shell
- `help` : affiche la liste des commandes disponibles



```
$ help
ls          : list files present in the file system
cat <file>  : dump the content of <file> to the screen
clear       : clear the screen
<prog>      : execute the program <prog>.
sleep <ms>  : sleep the specified number of milliseconds
exit        : exit the shell
$ _
```

FIGURE 36 – Shell développé

Démo

En plus du shell, une démo a été développée afin de valider toutes les fonctionnalités de l'OS. Cette démo appelle toute les fonctions de la librairie système et teste l'allocation mémoire dynamique.

9 Résultats

10 Discussions

10.1 Problèmes rencontrés

10.2 Améliorations possibles

11 Conclusion

Références

- [1] Florent Glück. Programmation système avancée, 2017.
- [2] Langage de programmation rust. <https://www.rust-lang.org/fr-FR/index.html>.
- [3] The rust compiler 0.1 is unleashed. <https://mail.mozilla.org/pipermail/rust-dev/2012-January/001256.html>.
- [4] Rust book second edition. <https://doc.rust-lang.org/book/second-edition>.
- [5] Cargo book. <https://doc.rust-lang.org/cargo>.
- [6] Writing an os in rust. <https://os.phil-opp.com>.
- [7] Target option. https://doc.rust-lang.org/1.1.0/rustc_back/target/struct.Target.html.
- [8] Target i386 example. <https://github.com/rust-lang/rust/issues/33497>.
- [9] Custom target path issue. <https://github.com/rust-lang/cargo/issues/4905>.
- [10] Rust book first edition. <https://doc.rust-lang.org/book/first-edition>.
- [11] `__floatundisf` issue. <https://users.rust-lang.org/t/kernel-modules-made-from-rust/9191>.
- [12] Linker scripts (osdev). https://wiki.osdev.org/Linker_Scripts.
- [13] Linker scripts (scoberlin). http://www.scoberlin.de/content/media/http/informatik/gcc_docs/ld_3.html.
- [14] Linker scripts (math.utah.edu). https://www.math.utah.edu/docs/info/ld_3.html.
- [15] Memory map (x86). [https://wiki.osdev.org/Memory_Map_\(x86\)](https://wiki.osdev.org/Memory_Map_(x86)).
- [16] Multiboot specifications. <https://www.gnu.org/software/grub/manual/multiboot/multiboot.html>.
- [17] David Decotigny et Thomas Petazzoni. Segmentation et interruptions. <http://sos.enix.org/wiki-fr/upload/SOSDownload/sos-texte-art2.pdf>.
- [18] Intel. Ia-32 intel architecture - software developer's manual - volume 3 : System programming guide. <http://flint.cs.yale.edu/cs422/doc/24547212.pdf>, 2003.
- [19] Segmentation. <https://wiki.osdev.org/Segmentation>.
- [20] Gdt. <https://wiki.osdev.org/GDT>.
- [21] David Decotigny et Thomas Petazzoni. Mise en place de la pagination. <http://sos.enix.org/wiki-fr/upload/SOSDownload/sos-texte-art4.pdf>.
- [22] Page translation. https://pdos.csail.mit.edu/6.828/2011/readings/i386/s05_02.htm.
- [23] Jean Gareau. Advanced embedded x86 programming : Paging, june 1998.
- [24] 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.
- [25] 8259 pic. https://wiki.osdev.org/8259_PIC.
- [26] Text ui. https://wiki.osdev.org/Text_UI.

- [27] Crt controller registers. <https://web.stanford.edu/class/cs140/projects/pintos/specs/freevga/vga/crtcreg.htm>.
- [28] Ata pio mode. https://wiki.osdev.org/ATA_PIO_Mode.
- [29] rustc - linux man pages. <https://www.systutorials.com/docs/linux/man/1-rustc/>.
- [30] Why is a rust executable large? <https://lifthrasiir.github.io/rustlog/why-is-a-rust-executable-large.html>.