

RustOS

Système d'exploitation en Rust

Orphée Antoniadis

Projet de Bachelor - Prof. Florent Glück

Hepia ITI 3ème année

Semestre de Printemps 2017-2018

Résumé

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

Table des matières

1	Introduction	8
1.1	Contexte	8
1.2	Objectif	8
2	Analyse	9
3	Conception	10
3.1	Environnement de développement	10
3.2	Technologies	10
3.2.1	Nasm	10
3.2.2	Rustup	10
3.2.3	Cargo et Xargo	10
3.2.4	QEMU	10
3.3	Architecture	10
4	Rust	11
5	Exécution du <i>kernel</i>	12
5.1	Compilation	12
5.2	<i>Linking</i>	12
5.3	<i>Boot</i>	13
5.3.1	Principe général	13
5.3.2	GRUB	13
5.3.3	Image ISO	14
6	Gestion mémoire	15
6.1	Introduction	15
6.2	Tables de descripteurs	16
6.2.1	GDT	16
6.2.2	IDT	16
6.3	Segmentation	16
6.4	Pagination	16
7	Peripherals	17
7.1	Ports	17
7.2	VGA	17
7.3	<i>Timer</i>	17
7.4	Clavier	17
8	Système de fichiers	18
8.1	Introduction	18
8.2	Structure	18
9	Résultats	19
10	Discussions	20
10.1	Problèmes rencontrés	20
10.2	Améliorations possibles	20
11	Conclusion	21
12	Références	22

Table des figures

1	Strucutre du fichier ELF	13
2	<i>Boot</i> d'une machine à base de BIOS	13
3	Exemple d'adressage mémoire	15
4	Protection mémoire avec un MMU	15
5	Couleurs du mode texte VGA	17

Remerciements

Convention typographique

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

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

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

Acronymes

BIOS *Basic Input Output System.* 13

CPU *Central Processing Unit.* 17

ELF *Executable and Linkable Format.* 10, 12, 13

GCC *GNU Compiler Collection.* 10, 12

GDT *Global Descriptor Table.* 16

GRUB *GRand Unified Bootloader.* 10, 13, 14

IA-32 *Intel Architecture, 32-bit.* 15

IDT *Interrupt Descriptor Table.* 16

ISO *International Organization for Standardization.* 10, 14

MBR *Master Boot Record.* 13

MMU *Memory Management Unit.* 15

OS *Operating System.* 10, 12–14, 17

PC *Personal Computer.* 13

PIO *Port Input/Output.* 17

RAM *Random Access Memory.* 15

VGA *Video Graphics Array.* 17

VRAM *Video Random Access Memory.* 13

1 Introduction

1.1 Contexte

1.2 Objectif

2 Analyse

3 Conception

3.1 Environnement de développement

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

3.2 Technologies

3.2.1 Nasm

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

3.2.2 Rustup

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

3.2.3 Cargo et Xargo

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

3.2.4 QEMU

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

3.3 Architecture

4 Rust

5 Exécution du *kernel*

5.1 Compilation

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

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

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

5.2 Linking

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

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

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

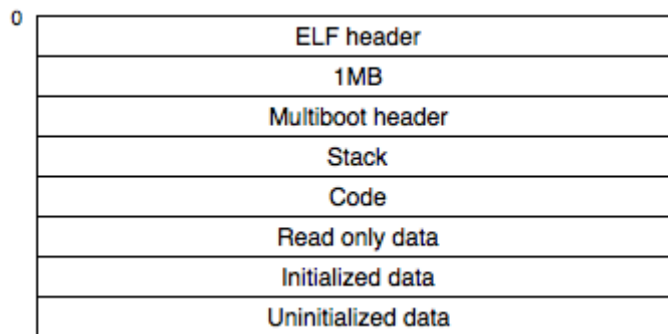


FIGURE 1 – Structure du fichier ELF

A noter que les sections commencent avec un *offset* de 1MB. Nous avons eu besoin de faire ça car les premiers 1MB dans un OS sont réservés [4]. La mémoire vidéo (VRAM) se situe par exemple dans cette zone.

5.3 Boot

5.3.1 Principe général

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

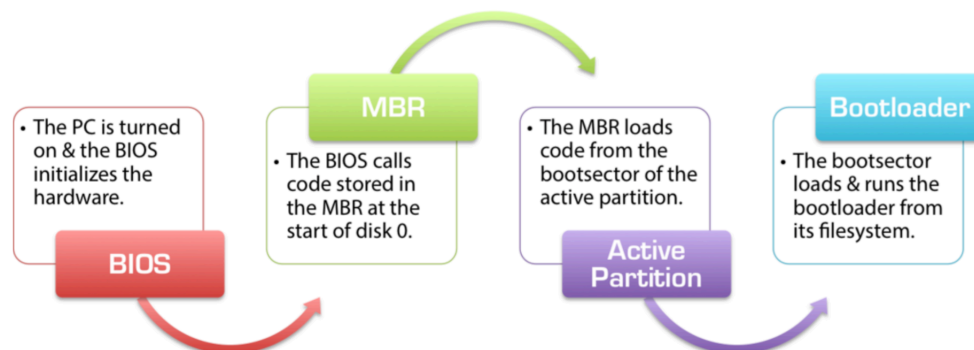


FIGURE 2 – Boot d'une machine à base de BIOS

5.3.2 GRUB

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

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

GRUB permet de charger n'importe quel type de système d'exploitation grace au standard *Multiboot*. Ce standard permet à tout *bootloader* de charger tout OS compatible [5].

5.3.3 Image ISO

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

```
isofiles
  boot
    grub
```

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

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

Pour finir, il faut exécuter la commande :

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

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

6 Gestion mémoire

6.1 Introduction

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

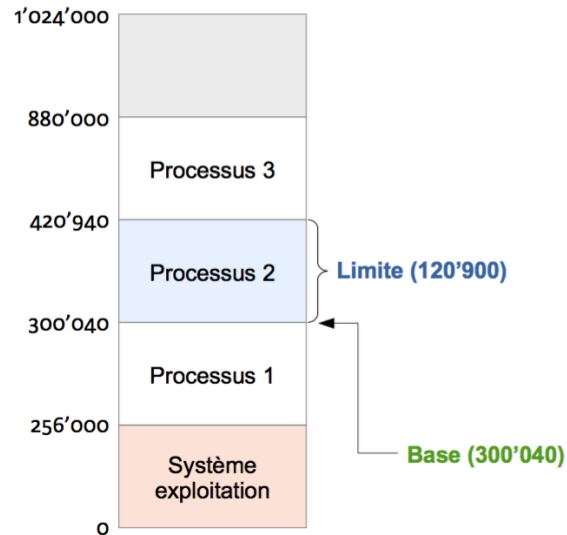


FIGURE 3 – Exemple d'adressage mémoire

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

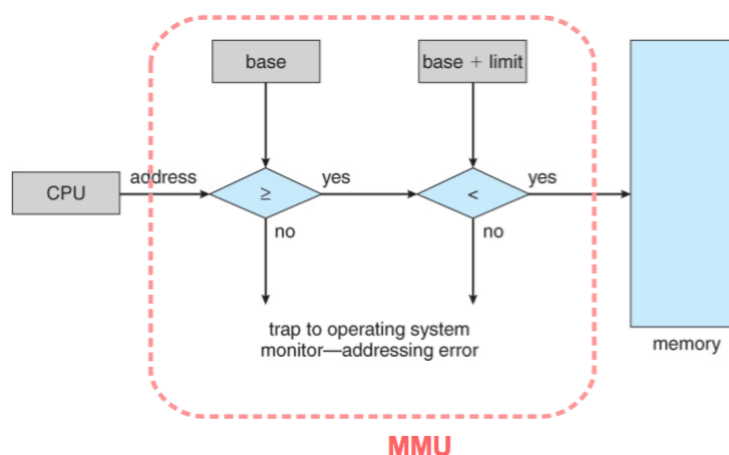


FIGURE 4 – Protection mémoire avec un MMU

Dans une architecture IA-32, la translation d'adresses se fait à l'aide de descripteurs définissant des segments de mémoire. Ces descripteurs sont contenus dans une table de descripteurs.

6.2 Tables de descripteurs

6.2.1 GDT

6.2.2 IDT

6.3 Segmentation

6.4 Pagination

7 Périphériques

7.1 Ports

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

7.2 VGA

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

















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 5 – Couleurs du mode texte VGA

7.3 *Timer*

7.4 Clavier

8 Système de fichiers

8.1 Introduction

8.2 Structure

9 Résultats

10 Discussions

10.1 Problèmes rencontrés

10.2 Améliorations possibles

11 Conclusion

12 Références

- [1] Linker scripts (osdev). https://wiki.osdev.org/Linker_Scripts.
- [2] Linker scripts (scoberlin). http://www.scoberlin.de/content/media/http/informatik/gcc_docs/ld_3.html.
- [3] Linker scripts (math.utah.edu). https://www.math.utah.edu/docs/info/ld_3.html.
- [4] Memory map (x86). [https://wiki.osdev.org/Memory_Map_\(x86\)](https://wiki.osdev.org/Memory_Map_(x86)).
- [5] Multiboot specifications. <https://www.gnu.org/software/grub/manual/multiboot/multiboot.html>.
- [6] Rust book first edition. <https://doc.rust-lang.org/book/first-edition>.
- [7] Rust book second edition. <https://doc.rust-lang.org/book/second-edition>.
- [8] Cargo book. <https://doc.rust-lang.org/cargo>.
- [9] Target option. https://doc.rust-lang.org/1.1.0/rustc_back/target/struct.Target.html.
- [10] Target i386 example. <https://github.com/rust-lang/rust/issues/33497>.
- [11] ___floatundisf issue. <https://users.rust-lang.org/t/kernel-modules-made-from-rust/9191>.
- [12] Writing an os in rust. <https://os.phil-opp.com>.
- [13] Writing an os in rust (second edition). <https://os.phil-opp.com/second-edition>.