

# Rapport OS From Scratch

---

- [Rapport OS From Scratch](#)
  - [1. Introduction](#)
  - [2. Prérequis](#)
    - [2.1. Machine virtuelle](#)
    - [2.2. Qemu](#)
    - [2.3. Nasm](#)
    - [2.4 Environnement](#)
  - [3 Etape du projet](#)
  - [4. Bootloader \(DONE\)](#)
  - [4. Crosscompiler](#)
    - [4.1 Implémenter le cross-compiler](#)
  - [5. Kernel](#)
  - [6. Problèmes rencontrés](#)
    - [6.1. Destruction d'une machine virtuelle](#)
    - [6.2. Installation du cross-compiler](#)
  - [Mode d'emploi](#)
    - [Nasm](#)
    - [QEmu](#)
  - [7. Conclusion](#)
  - [Référence](#)

## 1. Introduction

---

Nous employons tous un système d'exploitation (ou au moins avons employé une fois) surtout dans notre travail d'ingénieur en informatique. En effet, nous avons créé déjà des programmes qui sont exécutés sur des OS (Windows, Linux).

Dans le cours de conception OS nous avons choisi un projet qui consiste

à créer notre propre OS tournant sur une machine virtuelle QEMU (émulateur d'architecture processeur). Avec ce projet nous voulons donc mieux comprendre comment fonctionne du côté hardware qu'au côté software notre OS, notamment les points suivants :

- Comment boot un ordinateur
- Comment écrire des programmes bas niveau alors qu'il n'y a pas encore d'OS
- Comment relier le CPU avec tous les composants
- Comment passer du code assembleur à un langage haut niveau.
- Comment créer des caractéristiques d'un OS comme un filesystem, un shell, des drivers, des tâches

## 2. Prérequis

---

### 2.1. Machine virtuelle

Les différents collaborateurs du projet ont décidé de travailler sur une machine virtuelle Linux.

### 2.2. Qemu

Qemu est un logiciel qui permet de créer une machine virtuelle permettant de simuler une architecture sur laquelle nous pouvons instancier notre OS.

Nous émuloons un processeur i386 comme architecture.

### 2.3. Nasm

Nasm est un compilateur qui permet notamment de transformer nos fichiers assembleurs en fichiers binaire.

## 2.4 Environnement

Le projet a été réalisé sur un système UNIX plus précisément une distribution Ubuntu. Mais le projet peut se faire également sur un système MSDOS.

## 3 Etape du projet

---

- Créer un boot loader.
- Entrer dans un mode 32-bit.
- Pouvoir passer du langage bas niveau (assembleur) au haut niveau (exemple langage C).
- Gestion des I/O comme l'écran et le clavier.
- Associer un libc pour avoir des commandes de bases (par exemple utiliser BusyBox).
- Le MM (Memory Management).
- Créer un système de fichier (filesystem)
- Créer un shell de commande.

## 4. Bootloader (DONE)

---

Le bootloader est le programme qui sera lancé une fois que le BIOS a fini de s'initialiser.

Le Bios ne sait pas comment lancer l'OS, c'est le boot sector qui s'en occupe. Il est placé dans le 1er secteur du disque dur (cylindre 0, head 0, secteur 0). Il prend 512 bytes.

Ce boot sector est en 16-bit.

Dans un cas où nous n'utiliserions pas une architecture virtuelle mais directement initialiser l'OS sur un disque physique, il faudrait enregistrer le bootloader dans la MBR (Master Boot Record).

## 4. Passage de 16-bits(real mode) à 32-bits(protected mode)

---

Dans notre OS, notre CPU de base qui est initialisé par le BIOS démarre en real mode (le BIOS ne fonctionne qu'en 16-bit). Ce mode a quelques inconvénients par contre, il n'utilise pas toute la puissance du CPU. Dans notre cas, on voudrait pouvoir compiler un langage de haut niveau par la suite et pour cela, il nous faut passer notre OS en protected mode. Le mode 32-bit nous permet de travailler sur plusieurs adressages mémoires virtuelles qui ont chacun une taille maximum de 4GB de mémoire adressable. Ça permet aussi au système de renforcer la mémoire et les protections au niveau des I/O avec des instructions disponibles.

## 4. Crosscompiler

---

Un cross-compiler est un compilateur qui est exécuté sur un host platform (pour nous ça sera notre système émulé sur QEMU). Ensuite la plateforme cible est l'OS que nous sommes en train de réaliser (CPU, OS). Il est important de comprendre que la plateforme host et target ne sont pas les mêmes. Notre OS que nous faisons sera toujours différent de notre système actuelle. On doit utiliser un cross-compiler à moins qu'on développe notre vrai operating système. Mais pour notre projet nous ne voulons pas créer un nouveau langage, on veut juste pouvoir utiliser un langage de haut niveau déjà existant pour pouvoir éviter de tout coder en assembleur.

### 4.1 Implémenter le cross-compiler

Tout d'abord nous allons récupérer le path de la version actuelle de gcc, avec la commande suivante :

```
which gcc
```

```
export CC=/usr/bin/gcc-4.9
export LD=/usr/bin/gcc-4.9

export PREFIX="/usr/local/i386elfgcc"
export TARGET=i386-elf
export PATH="$PREFIX/bin:$PATH"
```

```
mkdir /tmp/src
cd /tmp/src
curl -O http://ftp.gnu.org/gnu/binutils/binutils-2.24.tar.gz
tar xf binutils-2.24.tar.gz
mkdir binutils-build
cd binutils-build
../binutils-2.24/configure --target=$TARGET --enable-interwork --enable-multilib --disable-nls --disable-werror --prefix=$
make all install 2>&1 | tee make.log
```

```
cd /tmp/src
curl -O https://ftp.gnu.org/gnu/gcc/gcc-4.9.2/gcc-4.9.2.tar.bz2
tar xf gcc-4.9.2.tar.bz2
mkdir gcc-build
cd gcc-build
../gcc-4.9.2/configure --target=$TARGET --prefix="$PREFIX" --disable-nls --disable-libssp --enable-languages=c --without-h
```

A noter qu'un script est disponible (gcc-4.9.x/contrib/download\_prerequisites) pour télécharger les dépendances + créer les symlink automatiquement :

Dans le dossier source :

```
./contrib/download_prerequisites
```

```
make all-gcc
make all-target-libgcc
make install-gcc
make install-target-libgcc
```

## 5. Kernel

Le Kernel est le coeur d'un Operating System. C'est lui qui est responsable de la gestion de la mémoire (MM), des I/O, de la gestion des interruptions, et encore d'autres choses.

## 6. Problèmes rencontrés

### 6.1. Destruction d'une machine virtuelle

Bien que l'erreur n'est pas réellement lié au projet, nous tenions à rendre hommage à l'un des collaborateurs du projet qui en voulant effacer un dans un dossier c'est un peu précipité et à tapé la ligne suivante :

```
rm -rf /*
```

Nous vous laissons imaginer la suite.

## 6.2. Installation du cross-compiler

### Problème rencontrés

---

### Mode d'emploi

---

#### Nasm

Pour passer du code assembleur à du code brute :

```
$nasm boot_sect.asm -f -o boot_sect.bin
```

#### QEmu

Pour émuler notre os.

```
$qemu your-os-boot-disk-image-file.bin
```

## 7. Conclusion

---

Ce projet à été un véritable défi, du fait qu'il soit très bas niveau, nous avons du apprendre les bases d'un OS, réutiliser de l'assembleur et du C, recherché le fonctionnement d'un os par rapport au BIOS.

Bien que nous n'avons pas pu implémenter toutes les fonctionnalités du cahier des charges, nous avons réussi à appeler un kernel basique à partir d'un bootloader. Ce qui n'était déjà pas une mince affaire.

## Référence

---

- [Writing a Simple Operation System - from Scratch](#) Document by Nick Blunde
- [os-tutorial](#) Github by Carlos Fenollosa
- [wiki.osdev.org](http://wiki.osdev.org) - Wiki for creation of operation system