Rapport OS From Scratch

1. Introduction

Nous employons tous un système d'éxploitation (ou au moins avons employé une fois) surtout dans notre travaille d'ingénieur en informatique. En effet, nous avons créé déjà des programmes qui sont exécuté 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 virtuel 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 suivant :

- Comment boot un ordinateur
- Comment écrire des programes 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 caractéristique d'un OS comme un filesystem, un shell, des drivers, des tâches

2. Prérequis

2.1. Machine virtuelle

Les différents collaborateurs du projets ont décidés de travailler sur une machine virtuel linux.

2.2. Qemu

Qemu est un logiciel qui permet de créer une machine virtuel permetant de simuler une architecture sur laquelle nous pouvons instancier notre os. Nous émulons un processeur i386 comme architécture.

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 réaliser sur une système UNIX plus précisement une distribution Ubuntu. Mais le projet peut se faire également sur un système MSDOS.

3 Etape du projet

- Crée un boot loader.
- Entrer dans un mode 32-bit.
- Pouvoir passer du langage bas niveau (assembleur) au haut niveau (exemple langaege 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 à finit 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 dr (cylindre 0, head 0, secteur 0). Il prend 512 bytes. Ce boot sector est en 16-bit. Dans un cas ou 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 initalisé 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. Ca permet aussi au système de renforcer la mémoire et les protections au niveaux des I/O avec des instructions disponibles.

4. Crosscompiler

Un cross-compiler est un compilateur qui est executé sur un host platform (pour nous ça sera notre système émulé sur QEMU). Ensuite la plateforme cible est l'OS que nous sommes entrain 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 différents 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 par 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 alons 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=$TARGET --enable-interwork --enable-multilib -
-disable-nls --disable-werror --prefix=$PREFIX 2>&1 | tee configure.log
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-headers
```

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.

Pour le projet nous avons choisi d'utiliser la libraire C, car elle fournit toutes les fonctions standard de C et les fournit en une forme binaire approprié pour le linkage avec les applications utilisateurs. En résumé, la librairie C est la mieux appropriée pour gérer notre OS.

Il fonctionne par compilation / assemblage. Un assembleur prends un code source et le converti en code machine (binaire), plus précisement, il converti le code source code en code object. Le compilateur prends le code source haut niveau, le converti convertie direcement en code object or dans notre cas avec GCC, converti direcement le source code en code source assembleur et invoque l'assemblage pour la partie final.

6. FileSystem

Le filesytem est enregistré sur la RAM, à l'initialisation du Kernel le fichier **LEKEBAB** est crée et va servir de rootfile et de fichier de base de notre liste chaîné. Après quoi on peut y ajouter des fichiers, qui contiennent un nom et un id.

Ce qui nous as permis de faire des fonctions de recherche par id et de listage des fichiers.

7. Problèmes rencontrés

7.1. Crosscompilator

Lors d'une mauvaise installation du GCC, on a essayé de passé du mode real en protected. On voulait compiler du code en C mais ceci ne fonctionnait.Il s'est avéré que le problème venait de la versionn du GCC (4.8). Nous avons du réinstaller une version plus récente de GCC (version 4.9.2) pour palier à ce problème. Il est important de noter qu'il faut toujours prendre la version la plus haute de GCC, car si on exécute du code C d'une version supérieure à la notre, ceci ne fonctionnera pas.

7.2. Filesystem

Nous étions partis sur l'idée d'enregistrer notre filesystem sur le disque dur, mais fort malheureusement, arpès beaucoup de recherche et de tentatives, nous n'avons pas réussi à appeler le *handling interrupt* du BIOS qui permet de pouvoir accèder au HDD.

Résolution : Ce problème a été résolu en créant notre filesystem à chaque redemarrage. C'est à dire qu'on stocke notre filesystem sur la RAM. Ce qui pour conséquence que à chaque fois que l'on redemarre notre OS,

on pert notre filesystem.

7.3. 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 éffacer un dans un dossier c'est un peu précipité et à tapé la ligne suivante :

```
rm -rf /*
```

Nous vous laissons imaginer la suite.

8. Mode d'emploie

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
```

9. 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 reussi 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 - Wiki for creation of operation system