The MOS Project

Lab Tutorial

Moody Liu

November 2022

Contents

Lab Introduction

The primary goal of MOS is to provide an platform-independent environment for beginners to learn OS.

Users of MOS will be able to experiment and get them familiar with with these following features of an operating system:

Note: The **bold** items are the main objectives of the project.

- Basic OS Knowledges
- Memory Management
  - **Paging**, and a Brief of Segmentation
  - Common Memory Management Practices in Modern Kernels
- File Systems
  - Underlying File Operations
  - VFS Framework (a file-system abstraction layer)
- Process Management
  - Allocating a New Process
  - The Famous `fork()` Syscall
  - **Process Scheduler**
  - **Threads**
  - **Threads Synchronization** via `mutex`

Chapter 1

Lab 1 - Setting Up the Development Environment

In this chapter, we'll set up the development environment for the rest of our labs. Firstly I'll introduce you to the tools we'll be using, and then, in the second part, we'll actually install them.

1.1 Introduction to the Tools

MOS is an operating system, thus, preparing for a development environment is already not an easy task (bruh). Several efforts have been made to make the process easier.

We are currently targeting the 32-bit `x86` architecture, the tools in table 1.1 are the ones we'll be using.

Tool	Description	Installation
`CMake`	1.1.1	1.2.1
`i686-elf` toolchain	1.1.5	1.2.5
`NASM`	1.1.2	1.2.2
`cpio`	1.1.3	1.2.3
`qemu-system-i386`	1.1.4	1.2.4

Table 1.1: Tools used in this lab

1.1.1 CMake

> CMake is an open-source, cross-platform family of tools designed to build, test and package software[1]

MOS uses CMake as the build system generator, it supports many build systems like 'Make', 'Ninja', 'Visual Studio' and 'Xcode'.

Note:

- It's the actual build system (e.g. 'Make') that starts the compiler, linker, etc., CMake is only to **generate** the configuration files for such build system.

We'll use 'Make' as the build system for MOS in this tutorial, but **you can use 'Ninja' if you want**.

[1] https://cmake.org/

1.1.2 NASM

NASM is an assembler for x86 architecture. There are several files under 'arch/x86' that are written in NASM. It has a cleaner syntax than the GNU assembler (i.e. `as`).

1.1.3 cpio

Cpio is the format of an archive, and also the tool to create such archives. MOS uses cpio as the initial root filesystem.

1.1.4 qemu-system-i386

QEMU is an open-source emulator, it also provides a gdb stub for debugging. It can be installed via your Linux's package manager.

(e.g. `apt install qemu-system-i386` or `pacman -S qemu-system-x86`, . . .).

See its download page for more details.

1.1.5 i686-elf Toolchain

As its name suggests, this is a cross toolchain for 'i686-elf' target. 'i686' means the 32-bit x86 architecture, and 'elf' is the executable format. They together form the 'target-triple' of the toolchain.

Tip:

- Most 64-bit Linux OSes have 'target triple' of `x86_64-pc-linux-gnu`, or `x86_64-linux-gnu`.
- Read more about 'target triple' at `https://wiki.osdev.org/Target_Triplet`.

Unlike other applications (e.g. `bash` or `vim`) that they run on an existing operating system and a standard C library (say, `glibc` or `musl`). MOS itself is the operating system, thus there's not an existing OS for it to run on, neither a standard libc (i.e. no `printf`, no `malloc` etc.) for it to use, considering you're directly interacting with the CPU and the hardware.

This is called 'bare-metal' environment, or 'freestanding' environment, a 'bare-metal' compiler toolchain is exactly for this situation.

Warning:

- One should never use a hosted compiler (e.g. the `gcc` installed on the lab machine) to cross-compile for a bare-metal environment when they are targeting a different architecture, (`x86` and `x86_64` are different), it ***may sometimes*** work, but you'll have to add a lot of unnecessary flags.
- See `https://wiki.osdev.org/Why_do_I_need_a_Cross_Compiler%3F` for more information.

1.2 Installating the Tools

1.2.1 CMake

CMake should come with your Linux distribution's package manager, MOS requires at least version 3.20, but any newer version is recommended.

1.2.2 NASM

NASM can be installed via your Linux's package manager, the minimum version of NASM tested is `2.15.03`. A pre-built binary from NASM's website is also available.

1.2.3 cpio

cpio can also be installed via your Linux's package manager, the minimum version of cpio tested is `2.12`.

1.2.4 qemu-system-i386

TODO

1.2.5 i686-elf Toolchain

Installing the cross-compiler toolchain is probably the most difficult part of this lab, but it's once and for all, you don't have to do it again (unless you want to upgrade the toolchain).

There are majorly two ways to get this toolchain, either by building it from source or by downloading a pre-built binary.

Downloading a pre-built binary

If you don't want to build the toolchain from source, you can download a pre-built binary from moodyhunter/i686-elf-prebuilt (choose the i686 one).

Warning:

- Using pre-built binary saves time, but please consider doing so **only** if you trust the author.
- The above pre-built binary is built with GitHub Actions, and is built on Ubuntu 20.04.5 LTS (Image `ubuntu-20.04` version `20221027.1`).

Building From Source

The source code of binutils and gcc can be found at GNU Binutils's Website and GNU GCC's Website respectively.

Note:

- For Arch Linux users, checkout i686-elf-binutils, i686-elf-gcc and i686-elf-gdb.

The script located at `docs/assets/i686-elf-toolchain.sh` downloads, compiles and installs them into the directory specified by the `PREFIX` variable.