TODO: write a nice title here

Thomas Hybel

Aarhus University
October 2017

Abstract

TODO: write an abstract here.

Contents

Introduction

- introduction - what are we trying to do - how are we doing it - following the course from MIT - why are we doing it - overview of the coming chapters

Lab 1

In this lab we wrote initialization code for our operating system. This code sets up a rudimentary page table and switches the processor from 16-bit real mode to 32-bit protected mode. We also wrote a boot loader, which is a small program that loads the main kernel from disk and transfers control to it.

The boot process

To understand the purpose of a boot loader, it is first necessary to have an overview of the process which an x86 machine goes through upon startup.

When an x86 machine starts, its BIOS code is run. The BIOS initializes some of the system's hardware components (e.g., keyboard, graphics card, and hard drive). The BIOS will then load one sector (512 bytes) from the boot medium into memory at a hard-coded address (0x7C00). Once this first sector is loaded into memory, the BIOS will transfer execution to the loaded code.

The first sector will typically contain a small program known as the boot loader. Its purpose is to load the main kernel from disk and transfer execution to it.

Before the boot loader loads the kernel, we first run some initialization code which sets up a more comfortable environment for the boot loader and kernel to work in.

Initialization code

When the BIOS jumps into our code, the processor is running in 16-bit real mode. However the code produced by a modern compiler expects to run in 32-bit protected mode. We therefore needed to write code which performs this switching of processor modes.

The main difference between real mode and protected mode is how address resolution is performed. In real mode, accessing a physical address is done using a segment selector register and a general-purpose register. In contast, protected mode allows (but does not require) the use of *paging*, i.e., the mapping from virtual from physical addresses using a page table.

Since we want our operating system to use virtual memory, we certainly want to switch to protected mode. We do not immediately enable paging, however.

To switch to protected mode, a bit needs to be set in the status register `cr0`. We do so with the following assembly code:

```
# protected mode enable flag
.set CR0_PE_ON, 0x1
```

```
movl    %cr0, %eax
orl     $CR0_PE_ON, %eax
movl    %eax, %cr0
```

To switch to 32-bit mode, we must update the `cs` (code segment) register. This register is an offset into a table called the Global Descriptor Table (GDT) which contains a number of descriptors. Each descriptor has a bit which determines whether the chosen segment is a 32-bit or a 16-bit segment. We use the `ljmp` instruction to update the `cs` register.

Once our code has switched to 32-bit protected mode, it executes the boot loader. At this point the processor is in a state where it can execute compiled C code, which means that we no longer need to hand-write assembly.

The file `boot/boot.S` contains the code that performs the described tasks.

The boot loader

It is the task of the boot loader to load the main kernel and transfer execution to it. The boot loader must do this using no more than 512 bytes of code, minus the bytes used by the initialization code.

Our kernel will be compiled into an Executable and Linkable Format (ELF) file. The boot loader will parse this ELF file, using the information therein to load the code and data of the kernel to the right physical addresses.

Once the boot loader has finished loading the kernel, it determines the entry point address from the ELF file and jumps there, transferring execution to the kernel.

Minimal kernel code

At this point our kernel can finally run. However it does not yet have much functionality. Unlike user-mode programs, the kernel does not have access to a C standard library, unless we write one ourselves.

As a start, we would like to have the ability to input and output text. Our kernel can use the `inb` and `outb` instructions to communicate with the outside world via a serial connection. We used this to write out a message and confirm that the kernel runs.

Lab 2

- lab 2 - alloc and free physical pages - page table management

Lab 3

- lab 3 - goal: make user-space process run - functions for keeping track of proceses - ELF loader - context switching - handling exceptions - handling syscalls

Lab 4

- lab 4 - preemptive multitasking - starting multiple processors - simple scheduler with yields - preemption (clock signals) - process forking with CoW - simple fork - CoW fork - IPC

Lab 5

- lab 5 - file system - shell

Lab 6

- lab 6 - networking: writing an e1000 card driver

Graphics Lab

- graphics

Hardware Lab

- hardware
    - conclusion

References