实验 1：体验真实的操作系统

2020 年 11 月 26 日 黄志勇

1 背景知识

本课程实验以开源教学操系统 pintos 为基础，下一次实验以此次实验知识为前提。此次实验是本课程所有实验中最简单的一个，每一位同学都需要独立完成此次操作实验中的所有操作步骤，并认真完成课后作业。

1.1 启动过程

系统的启运过程是指按下电源键(Power)后，计算机上电并启动 BIOS 系统自检，随后加载硬盘上的系统加载器(loader)至内存，并将 CPU 控制权限交给加载器，加载器加载磁盘上的操作系统内核至内存空间，执行完加载工作后，将 CPU 的控制权限交给操作系统内核的过程。

The process of loading the operating system into memory for running after a PC is powered on is commonly known as **bootstrapping**. The operating system will then be loading other software such as the shell for running. Two helpers are responsible for paving the way for bootstrapping: BIOS (Basic Input/Output System) and bootloader. The PC hardware is designed to make sure BIOS always gets control of the machine first after the computer is powered on. The BIOS will be performing some test and initialization, e.g., checking memory available and activating video card. After this initialization, the BIOS will try to find a bootable device from some appropriate location such as a floppy disk,

hard disk, CD-ROM, or the network. Then the BIOS will pass control of the machine to the bootloader who will load the operating system.

While BIOS and the bootloader have a large task, they have very few resources to do it with. For example, IA32 bootloaders generally have to fit within 512 bytes in memory for a partition or floppy disk bootloader (i.e., only the first disk *sector*, and the last 2 bytes are fixed signatures for recognizing it is a bootloader). For a bootloader in the Master Boot Record (MBR), it has to fit in an even smaller 436 bytes. In addition, since BIOS and bootloader are running on bare-metals, there are no standard library call like `printf` or system call like `read` available. Its main leverage is the limited BIOS interrupt services. Many functionalities need to be implemented from scratch. For example, reading content from disk is easy inside OSes with system calls, but in bootloader, it has to deal with disk directly with complex hardware programming routines. As a result, the bootloaders are generally written in assembly language, because even C code would include too much bloat!

To further understand this challenge, it is useful to look at the PC's physical address space, which is hard-wired to have the following general layout:

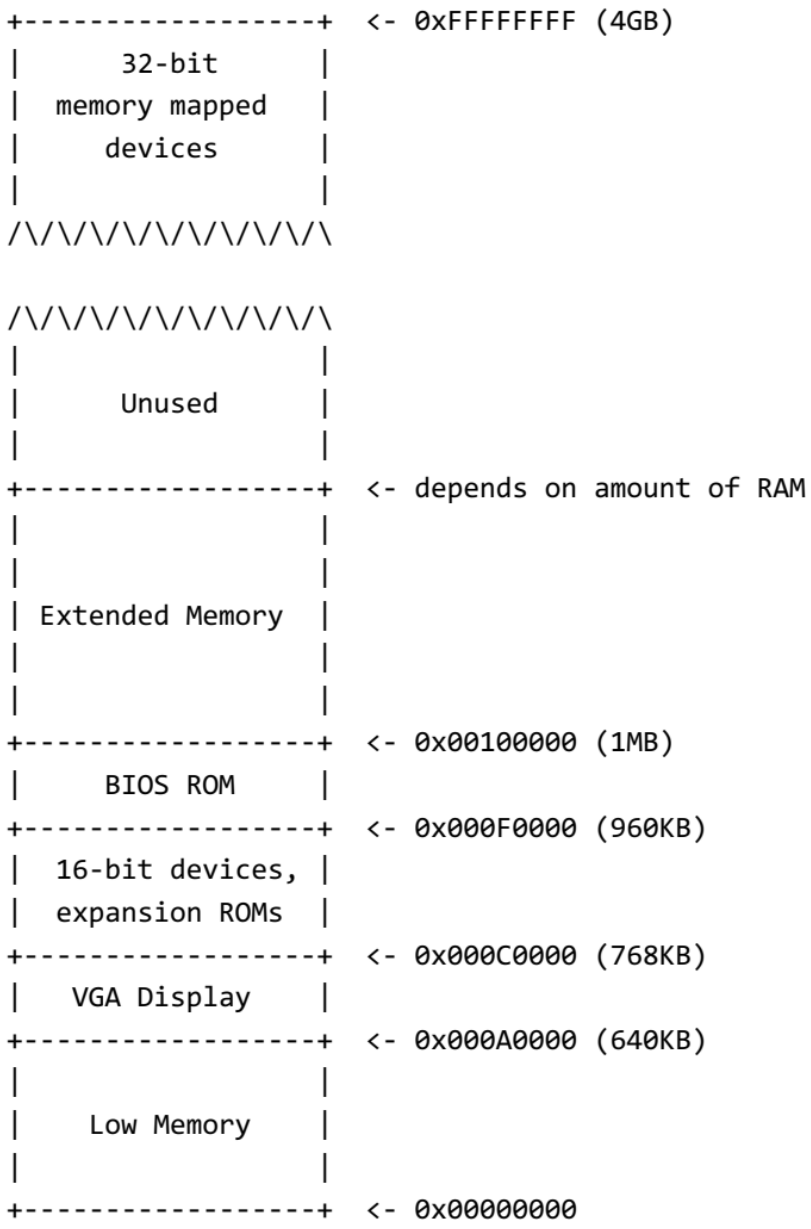
The first PCs, which were based on the 16-bit Intel 8088 processor, were only capable of addressing 1MB of physical memory. The physical

address space of an early PC would therefore start at 0x00000000 but end at 0x000FFFFF instead of 0xFFFFFFFF. The 640KB area marked "Low Memory" was the only random-access memory (RAM) that an early PC could use; in fact the very earliest PCs only could be configured with 16KB, 32KB, or 64KB of RAM!

The 384KB area from 0x000A0000 through 0x000FFFFF was reserved by the hardware for special uses such as video display buffers and firmware held in non-volatile memory. The most important part of this reserved area is the BIOS, which occupies the 64KB region from 0x000F0000 through 0x000FFFFF. In early PCs the BIOS was held in true read-only memory (ROM), but current PCs store the BIOS in updateable flash memory.

When Intel finally "broke the one megabyte barrier" with the 80286 and 80386 processors, which supported 16MB and 4GB physical address spaces respectively, the PC architects nevertheless preserved the original layout for the low 1MB of physical address space in order to ensure backward compatibility with existing software. Modern PCs therefore have a "hole" in physical memory from 0x000A0000 to 0x00100000, dividing RAM into "low" or "conventional memory" (the first 640KB) and "extended memory" (everything else). In addition, some space at the very top of the PC's 32-

bit physical address space, above all physical RAM, is now commonly reserved by the BIOS for use by 32-bit PCI devices.



1.2 操作系加载器(boot loader)

Floppy and hard disks for PCs are divided into 512 byte regions called sectors. A sector is the disk's minimum transfer granularity: each read or

write operation must be one or more sectors in size and aligned on a sector boundary. If the disk is bootable, the first sector is called the boot sector, since this is where the boot loader code resides. When the BIOS finds a bootable floppy or hard disk, it loads the 512-byte boot sector into memory at physical addresses 0x7c00 through 0x7dff, and then uses a jmp instruction to set the CS:IP to 0000:7c00, passing control to the boot loader.

IA32 bootloaders have the unenviable position of running in **real-addressing mode** (also known as "real mode"), where the segment registers are used to compute the addresses of memory accesses according to the following formula: *address* = 16 * *segment* + *offset*. The code segment CS is used for instruction execution. For example, when the BIOS jump to 0x0000:7c00, the corresponding physical address is 16 * 0 + 7c00 = 7c00. Other segment registers include SS for the stack segment, DS for the data segment, and ES for moving data around as well. Note that each segment is 64KiB in size; since bootloaders often have to load kernels that are larger than 64KiB, they must utilize the segment registers carefully.

Pintos bootloading is a pretty simple process compared to how modern OS kernels are loaded. The kernel is a maximum of 512KiB (or 1024 sectors), and must be loaded into memory starting at the address 0x20000. Pintos does require a specific kind of partition for the

OS, so the Pintos bootloader must look for a disk partition of the appropriate type. This means that the Pintos bootloader must understand how to utilize Master Boot Records (MBRs). Fortunately they aren't very complicated to understand. Pintos also only supports booting off of a hard disk; therefore, the Pintos bootloader doesn't need to check floppy drives or handle disks without an MBR in the first sector.

When the loader finds a bootable kernel partition, it reads the partition's contents into memory at physical address 128 kB. The kernel is at the beginning of the partition, which might be larger than necessary due to partition boundary alignment conventions, so the loader reads no more than 512 kB (and the Pintos build process will refuse to produce kernels larger than that). Reading more data than this would cross into the region from 640 kB to 1 MB that the PC architecture reserves for hardware and the BIOS, and a standard PC BIOS does not provide any means to load the kernel above 1 MB.

The loader's final job is to extract the entry point from the loaded kernel image and transfer control to it. The entry point is not at a predictable location, but the kernel's ELF header contains a pointer to it. The loader extracts the pointer and jumps to the location it points to.

The Pintos kernel command line is stored in the boot loader (using about 128 bytes). The pintos program actually modifies a copy of the boot

loader on disk each time it runs the kernel, inserting whatever command-line arguments the user supplies to the kernel, and then the kernel at boot time reads those arguments out of the boot loader in memory. This is not an elegant solution, but it is simple and effective.

1.3 内核

The bootloader's last action is to transfer control to the kernel's entry point, which is start() in threads/start.S. The job of this code is to switch the CPU from legacy 16-bit "**real mode**" into the 32-bit "**protected mode**" used by all modern 80*x*86 operating systems.

The kernel startup code's first task is actually to obtain the machine's memory size, by asking the BIOS for the PC's memory size. The simplest BIOS function to do this can only detect up to 64 MB of RAM, so that's the practical limit that Pintos can support.

In additional, the kernel startup code needs to to enable the A20 line, that is, the CPU's address line numbered 20. For historical reasons, PCs boot with this address line fixed at 0, which means that attempts to access memory beyond the first 1 MB (2 raised to the 20th power) will fail. Pintos wants to access more memory than this, so we have to enable it.

Next, the kernel will do a basic page table setup and turn on protected mode and paging (details omitted for now). The final step is to call into

the C code of the Pintos kernel, which from here on will be the main content we will deal with.

2 实验内容

1.1 准备工具链(Tool Chain)

◆ 根据项目 osutils 准备工具链

https://gitee.com/ctguhzy/osutils

◆ 直接获取工具链

git clone https://gitee.com/ctguhzy/OSToolChain.git

cd OSToolChain

source ./setup.sh

1.2 编译项目源代码

```
$ git clone https://gitee.com/ctguhzy/pintos.git


$ cd pintos/src/threads
$ make qemu
```

```
$ cd pintos/src/threads
$ make
$ cd build
$ pintos --bochs -- run alarm-zero
```

1.3 调试内核

根据网页内容，练习调试内核，完成 E.5.2 Example GDB Session

https://cs.jhu.edu/~huang/cs318/fall20/project/pintos_11.html#SEC161

3 课后作业

在 pintos 系统中启动系统并运行程序的方式如，pintos -- run alarm-zero. 系统启动并执行完任务后结束退出。这不符合我们使用操作系统的习惯，通常我们启动系统后会进入 Shell 模式(命令行模式)等待用户输入命令。现在我们启动系统后需要进入类似模式，并显示提示符 PINTOS>，然后等待输入命令。当输入命令是 whoami 时，打印出你的姓名，并进行下一行显示提示符。

当输入 quit 时，系统允许执行完内核并退出。

当输入其它命令时，打印出无效信息。