COS318 Project 1
Bootup Mechanism

Wei Dong

Sept. 20, 2006

Basic information

- This semester: 6 projects, a tiny OS kernel
  - Wei Dong: 1, 3, 4
  - Zhe Wang: 2, 5, 6
- Lab: Friend Center 010
- Login with OIT username/password
- Feel free to contact the TA if there's a problem with the machines

What's the problem?

- What is a kernel?
- How is a program normally run?
  - Supported by the OS
  - (Also restricted by the OS)
- But how does the OS itself get loaded?
  - Bad news: In the beginning, there is bare machine…
  - (Not totally bare, we still have the BIOS)
  - Good news: we can do anything with it

3

PC Bootup Process

- Start from FFFF:0000 (ROM BIOS)
- Self test and initialization
- Search for boot device
  - Bootblock: end with "0x55,0xAA"
- Load the 1st sector to 07C0:0000
- Jump to 07C0:0000

4

Dealing with the limitation

- BIOS loads only the first sector (512 bytes)
- But the kernel is larger
- Bootblock:
  - Reside in the boot sector
  - Load the whole kernel into memory
- Where is the kernel?
  - Real life: kernel as an executable file in the disk file system
    - Linux: /boot/vmlinuz
    - Bootloader understands the file system and executable file format
  - Our projects
    - Memory layout pre-calculated
    - The Memory Image stored in the sectors right after the bootblock

5

What You Must Do

- bootblock.s
  - Load the kernel
  - Setup stack, data segments
  - Transfer control to kernel
- createimage.c
  - Extract code and data from executables
  - Pack into boot disk (bootblock image + kernel image)

6

What We Provide

- bootblock_example.s
- bootblock.s
- createimage.c
- createimage.given
- kernel.s
- Makefile

You can start with either part

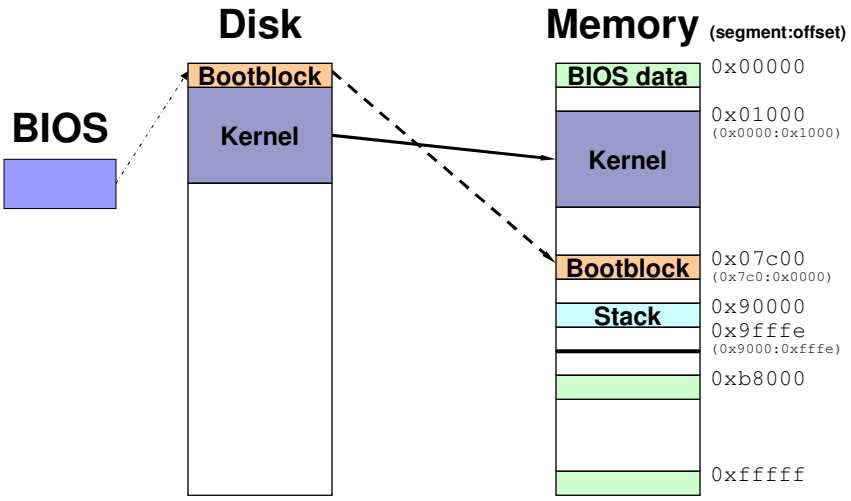
7

The Makefile

```
kernel:         kernel.s

bootblock:      bootblock.s

createimage:    createimage.c

image:          createimage bootblock kernel
        ./createimage bootblock kernel

boot:           image
        cat ./image > /dev/sda
```

8

Bootstrapping Layout



9

bootblock.s

10

When BIOS gives control to you

- CS = 0x07C0
- IP = 0x0000
- DL = Boot device number
  - You will load the kernel from this device
- Don't assume things to be there…
  - You'll have to setup the segments and stack if you want to use them

11

bootblock.s

Read the kernel into memory (INT13)

- Kernel starts at 0x0:0x1000
- Use hardcoded kernel size
  - (os_size: number of sectors)

- Set the kernel data segment and stack
  - DS=0x0
  - SS=0x9000, ESP=0xFFFE

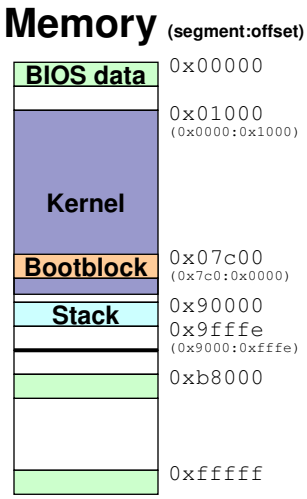
  (The kernel is 32bit, so setup the 32bit environment)
- Transfer control to kernel
  - Jump to 0x0:0x1000

12

Problem

What if a big kernel overwrites the bootblock?

(Extra credit)



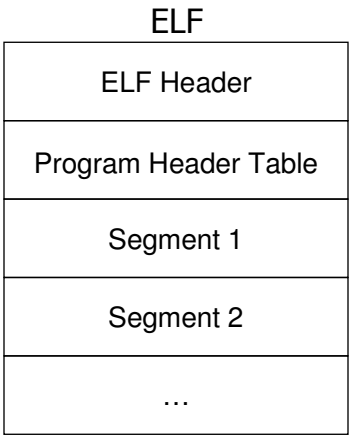
13

createimage.c

14

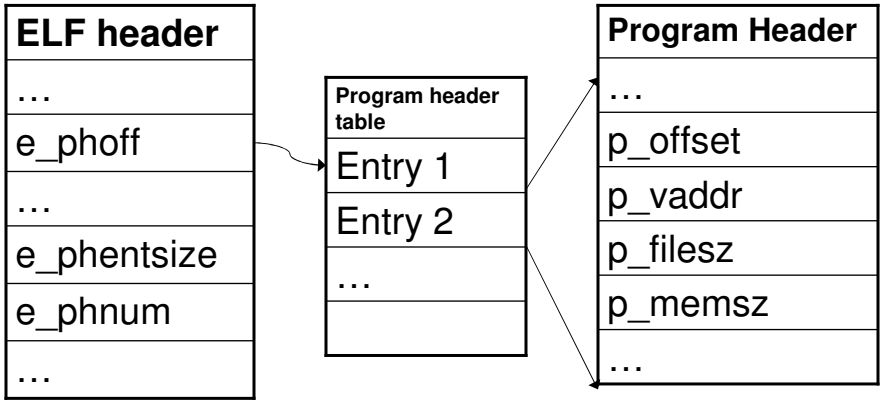
createimage.c

- Read a list of executable files (ELF)
- Write segments (real code) into bootblock + kernel image file
- Utilities: objdump, readelf



15

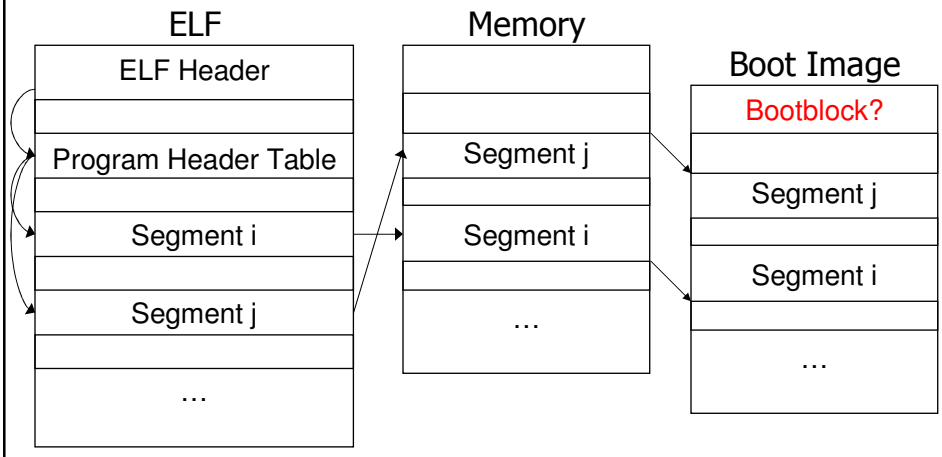
A look into the ELF format



Include elf.h for the data type definition

16

ELF Data Layout



File Offset = Memory Address – OS Start + sizeof(bootblock)

17

createimage.c

- Read ELF header to find offset of program header *table*
- Read program header to find start address, size and location of segment
- Pad and copy segment into image file
  - Note that bootloader is treated differently
- Write kernel size to hardcoded location in image file
- Write the boot signature

18