CS 326: Operating Systems

Booting Up

Lecture 4

Today

- Discussing buggy C functions and lab submission
- The boot process

Booting Up

- When you turn on your computer, it starts the **boot process**
  - Named for “pulling oneself over a fence by one’s bootstraps”
    - Or in other words, doing the impossible
- Booting is a series of tasks that ultimately get the operating system running
- The first thing you (may) see is the POST
  - Power-on Self Test


Award Modular BIOS v6.00PG, An Energy Star Ally
Copyright (C) 1984-99, Award Software, Inc.
BIW1M/BIW2M BIOS V1.3
Main Processor : PENTIUM II 910MHz
Memory Testing : 131072K OK + 1024K Shared Memory
Award Plug and Play BIOS Extension v1.0A
Copyright (C) 1999, Award Software, Inc.
energy
EPA POLLUTION PREVENTER

BIOS / UEFI

- After initializing the hardware, your basic input-output system (BIOS) will start iterating through the disks connected to your machine
  - When installing a new OS, you may change the **boot order** (what disk to boot from)
  - On a Mac, you can do this by holding down Option during boot (other machines: F12, Del…)
- Once a bootable disk is found, we proceed to the next boot phase

Master Boot Record

- The first 512 bytes on a hard drive contain the Master Boot Record (MBR)
- The MBR has two parts:
  - **Boot code**
    - Responsible for continuing the boot process
  - **Partition table**
    - Partition: segment of a disk
    - Partition 1: Windows; Partition 2: Linux; etc.
    - Basically contains a pointer to where each partition starts

Finding a Bootable Partition (x86)

- The BIOS looks for a disk partition that ends with 0x55AA
  - (Bootable flag)
- It copies the MBR into memory at `0x0000:0x7C00`
  - Segment 0, address 0x7C00
- And finally, it starts executing the first instruction at `0x7C00`.



MBR Contents

Start Address	Contents	Size (bytes)
0x0000	Boot Code	446
0x01BE	Partition Table (16 bytes each)	64
0x01FE	0x55	1
0x01FF	0xAA	1

- Observations:
  - 512 bytes is not a lot of bytes
  - Only four partitions possible

Demo: Booting

Let’s take a look at a minimal boot code.

Demo: mini OS

…and now, a more complicated “OS”

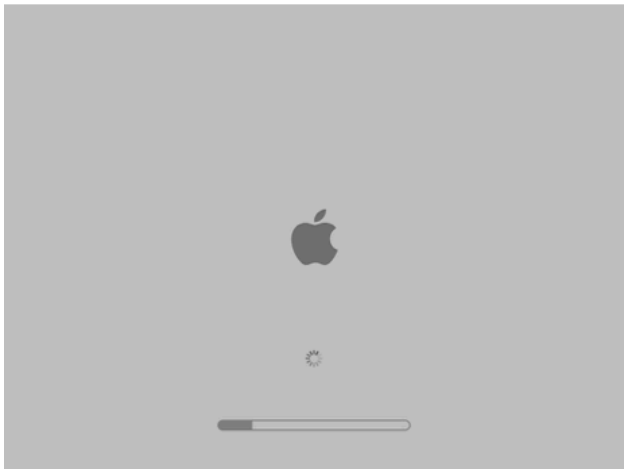
Hard-Coded Offsets

- You might be surprised that these offsets are just hard-coded into the system
- However, the BIOS is not particularly smart
  - Originally was a read-only chip (modern motherboards have flashable BIOS)
  - Making the BIOS smarter would make it more complex and more expensive
  - Tend to be specialized for the hardware, unlike OS
- Modern EFI systems add more complexity, but the principle remains the same

Continuing the Boot Process

- We began executing instructions, so we're all done starting the OS, right?
  - Unfortunately, no…
- The first instructions executed are part of the **bootloader**
- The bootloader is a bit smarter than the BIOS, and handles the next steps in the boot process





Bootloader

- The bootloader can understand a variety of file systems
  - Invent a new file system? Add support to the bootloader.
- It can also provide a list of operating systems available in a multi-boot configuration
- …And it can handle larger disks!
  - The BIOS is limited to a fixed number of partitions and disk sizes

Bootloader Restrictions

- You can only have one bootloader per disk
- Installing one will overwrite another
  - *Windowsssssssssss!!!*

Finishing the Boot Process

- The bootloader starts the OS
- The kernel initializes its data structures and prepares to run the first **process**
  - Process: a running instance of a program
- PID 1, often **init**, starts the rest of the processes
  - Gets loaded by the kernel, put in the process table, and then run by the **scheduler**
  - Init is responsible for services, startup tasks, etc.
    - (`systemd` on Linux)

PID 1, a.k.a. init

- PID 1 is the direct or indirect ancestor of all other processes
  - When one process launches another, it is that process's **parent**
    - The newly-launched process is the **child**
    - Unfortunately there are no uncle or aunt processes…
  - The kernel or scheduler is usually considered PID 0, but they technically aren't processes.
- Some init implementation can do more or less:
  - System V Init, **systemd**, upstart, launchd, etc.
- You can even write your own!
  - Boot Linux with flag: `init=/path/to/your/init`

Linux: init/main.c

```
/* We try each of these until one succeeds. The Bourne shell can be
 * used instead of init if we are trying to recover a really broken machine. */
if (execute_command) {
  ret = run_init_process(execute_command);
  if (!ret)
    return 0;
  panic("Requested init %s failed (error %d).", execute_command, ret);
}
if (!try_to_run_init_process("/sbin/init") ||
    !try_to_run_init_process("/etc/init") ||
    !try_to_run_init_process("/bin/init") ||
    !try_to_run_init_process("/bin/sh"))  return 0;

panic("No working init found.  Try passing init= option to kernel. "
      "See Linux Documentation/admin-guide/init.rst for guidance.");
```

Process Lineage

```
[malensek@ruby:~]$ ps -ef

UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Feb21 ?        00:00:57 /usr/lib/systemd/systemd
root         2     0  0 Feb21 ?        00:00:00 [kthreadd]
root         3     2  0 Feb21 ?        00:00:00 [ksoftirqd/0]
root         5     2  0 Feb21 ?        00:00:00 [kworker/0:0H]
root         7     2  0 Feb21 ?        00:01:21 [rcu_sched]
root         8     2  0 Feb21 ?        00:00:00 [rcu_bh]
root         9     2  0 Feb21 ?        00:00:13 [rcuos/0]
root        10     2  0 Feb21 ?        00:00:00 [rcuob/0]

...

malensek    235     1  0 Feb21 tty1     00:00:00 login -- malensek
malensek    282   235  0 Feb21 tty1     00:00:00 bash
```

xv6 Bootloader

- On the RISC-V version of xv6, the emulated hardware provided by QEMU includes a boot ROM
  - This provides a simple bootloader – no need for us to write or install one.

Extra Credit

- There will be two extra credit opportunities today:
  - Tracing the boot process
  - Tetris competition

Tracing the Boot Process [1/2]

When you start your OS, it looks like this:

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$
```

- How did we get there?
  - We started with QEMU's "bootloader"
  - The final `$` is displayed by `sh.c`
  - What files were involved in between?

Tracing the Boot Process [2/2]

- You'll perform a human backtrace through the code to find the missing links between the bootloader and `sh`
- Provide a list of files, e.g.:
  - `kernel/file1.c`
  - `kernel/file2.c`
  - `user/file3.c`
- Hint: start with `kernel/kernel.ld`
- Write your answer on the board. Closest answer gets the extra credit!

Tetris Competition

- `cd /home2/tetris-os`
- `cat NOTES.txt`
- Go!