# Operating Systems

## Booting

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

January, 2023

# The chicken and egg problem!!!

**The computer starts the OS**

*OR*

**The OS starts the computer**

A computer without an OS is like a body without a soul.

# The chicken and egg problem!!!

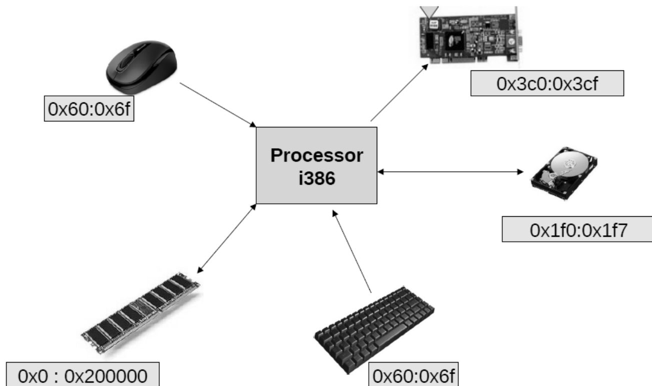
The first thing a computer does after getting switched on is to start up the OS. The OS helps other computer programs to work by handling the details of controlling the computer's hardware.

**Controlling the computer's hardware is a bit tricky!!!**

A bit about hardware management

The Booting Process
- Basics
- Booting in Linux
- Booting in xv6

# How OS recognizes the hardware?




Everything within and connected to a computer has an address!!!

# Types of addresses

1. Memory addresses
2. I/O addresses
3. Memory mapped I/O addresses

# Memory addresses

- Range: 0 to RAM size ($2^{32}-1$ or $2^{64}-1$)
- Mapping of main memory
  – Used to store data for code, heap, stack, OS, etc.
- Mapping of low and extended memory
  – Used by video buffers, expansion ROMS, BIOS ROMs, etc.
  – Used by latest OSs
- Accessed by load/store instructions

```
/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

# I/O addresses

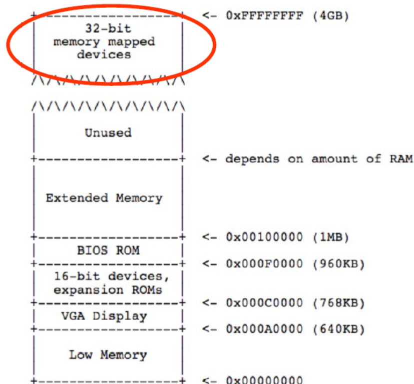
- Range: 0 to $2^{16} - 1$
- Used to access devices
- Uses a different bus compared to RAM memory access
  – Completely isolated from memory
- Accessed by in/out instructions

| I/O address range (hexadecimal) | device |
|---|---|
| 000–00F | DMA controller |
| 020–021 | interrupt controller |
| 040–043 | timer |
| 200–20F | game controller |
| 2F8–2FF | serial port (secondary) |
| 320–32F | hard-disk controller |
| 378–37F | parallel port |
| 3D0–3DF | graphics controller |
| 3F0–3F7 | diskette-drive controller |
| 3F8–3FF | serial port (primary) |

**See:** `https://bochs.sourceforge.io/techspec/PORTS.LST`

# Memory mapped I/O addresses

- Provides additional space
- Devices and RAM share the same address space
- Instructions used to access RAM can also be used to access devices.
  – E.g., load/store



# Who defines address ranges?

- Standard components
  - Industry standards (e.g., IBM PC standard)
  - Fixed for all PCs
  - Ensures BIOS and OS to be portable across platforms
- Plug and Play devices
  - Address range set by BIOS or OS
  - A device address range may vary every time the system is restarted

# What is booting

**Definition (Booting)**

The process of bringing up the OS is called booting.

The computer knows how to boot because instructions for booting are built into one of its chips, namely the BIOS chip.

The term booting came from bootstrapping that refers to the process of pulling it up by the bootstraps.

# How booting works?

Power initialization with Power-on Reset (PoR)

Basic Input/Output System (BIOS) startup

Execution of Master Boot Record (MBR)

Boot loader in action

Kernel in action

Running OS with *init*

# Power initialization with PoR

Every register is initialized to 0 except CS = 0xf000 and IP = 0xfff0 as follows.

- First instruction fetched from the physical address = (CS << 4) + IP = 0xffff0
- Processor in real mode (backward compatible to 8088)
  - Limited to 1MB addresses
  - No protection / privilege levels
  - Direct access to all memory
  - No multi-tasking
- First instruction is at right on top of accessible memory
  - Should jump to another location

# BIOS startup

- BIOS is present in a small chip connected to the processor
  - Flash/EPROM/E2PROM
- BIOS performs the following
  - Power-On Self-Test (POST)
  - Initialize video card and other devices
  - Display BIOS screen
  - Perform brief memory test
  - Set DRAM memory parameters
  - Configure Plug  Play devices
  - Assign resources (DMA channels  IRQs)
  - Identify the boot device

  (i) Read sector 0 from boot device into memory location 0x7c00

  (ii) Jumps to 0x7c00

```
/\/\/\/\/\/\/\/\/\/\
|      Unused      |
+------------------+
| Extended Memory  |
+------------------+ <- 0x00100000
|     BIOS ROM     |
+------------------+ <- 0x000F0000
| 16-bit devices,  |
| expansion ROMs   |
+------------------+
|   VGA Display    |
+------------------+
|    Low Memory    |
+------------------+
```

686 AMIBIOS® ©1999 UN01 5986

# BIOS startup

When you first turn on your computer it tests itself to make sure everything is in working order. This is called the Power-on self-test. Then a program called the bootstrap loader, located in the ROM BIOS, looks for a boot sector. A boot sector is the first sector of a disk and has a small program that can load an operating system. Boot sectors are marked with a magic number 0xAA55 = 43603 at byte 0x1FE = 510. These are the last two bytes of the sector. This is how the hardware can tell whether the sector is a boot sector or not.

The boot loader is pulled into memory and started. The boot loader's job is to start the real OS.

**Note:** Two popular boot loaders in Linux are LInux LOader (LILO) and GRand Unified Bootloader (GRUB).

# Execution of MBR

- Sector 0 in the disk is called MBR
- Contains code that boots the OS or another boot loader
- Copied from disk to RAM (@0x7c00) by BIOS and then begins to execute
- Size 512 bytes
  - 446 bytes bootable code
  - 64 bytes disk partition information (16 bytes per partition)
  - 2 bytes signature
- Typically the MBR code looks through partition table and loads the bootloader (such as Linux or Windows)
- or, it may directly load the OS

# Boot loader in action

The BIOS chip tells it to look in a fixed place, usually on the lowest-numbered hard disk (the boot disk) for a special program called a boot loader. Notably, the boot loader may be present in the MBR (sector 0) itself.

The loader looks for the kernel, loads it into memory, and start it. When you boot Linux and see LILO / GRUB on the screen followed by a bunch of dots, it is loading the kernel.

The boot loader also performs the following.

- Disabling the interrupts
  - Interrupts are re-enabled when the OS is ready
- Setting up Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT)
- Switching from real mode to protected mode
- Reading OS from the disk

# Let's think!!!

**Why does not the BIOS load the kernel directly?**

The BIOS was originally written for primitive 8-bit PCs with tiny disks, and literally cannot access enough of the disk to load the kernel directly. The boot loader step also lets you start one of several operating systems off different places on your disk, in the unlikely event that Unix is not good enough for you. In fact, Linux does not use the BIOS at all after the booting.

# The kernel in action

Once the kernel starts, it has to look around, find the rest of the hardware, and get ready to run programs. It does this by poking not at ordinary memory locations but rather at I/O ports – special bus addresses that are likely to have device controller cards listening at them for commands.

The kernel does not poke at random; it has a lot of built-in knowledge about what it is likely to find where, and how controllers will respond if they are present. This process is called autoprobing.

# The kernel in action

Most of the messages you see at boot time are the kernel autoprobing your hardware through the I/O ports, figuring out what it has available to it and adapting itself to your machine.

The Linux kernel is extremely good at this, better than most other Unixes and much better than DOS or Windows. In fact, many Linux old-timers think the cleverness of Linux's boot-time probes (which made it relatively easy to install) was a major reason it broke out of the pack of free-Unix experiments to attract a critical mass of users.

Getting the kernel fully loaded and running completes the first stage (sometimes called run level 1) of booting.

# Running OS with *init* – Step 0

After completing run level 1, kernel hands over the control to a special process called *init* which spawns several housekeeping processes.

The init is a daemon process that continues running until the system is shut down.

# Running OS with *init* – Step 1

The first job of *init* process is usually checking to make sure your disks are alright. Disk file systems are fragile things; if they have been damaged by a hardware failure or a sudden power outage, there are good reasons to take recovery steps before the OS is up.

# Running OS with *init* – Step 2

The next step of init is to start several daemons. A daemon is a program like a print spooler, a mail listener or a WWW server that lurks in the background, waiting for things to do. These special programs often have to coordinate several requests that could conflict.

They are daemons because it is often easier to write one program that runs constantly and knows about all requests than it would be to try to make sure that a flock of copies (each processing one request and all running at the same time) do not step on each other. The particular collection of daemons your system starts may vary, but will almost always include a print spooler (a gatekeeper daemon for your printer).

# Other setups

- Setting up virtual memory
- Initializing interrupt vectors
- Initialize timers, monitors, hard disks, consoles, filesystems, etc.
- Initializing other processors (if any)
- Starting up user processes

# Logging in

The next step is to prepare for users. The init starts a copy of a program called *getty* to watch your console (and maybe more copies to watch dial-in serial ports). This program is what issues the login prompt to your console. At this level, you can log in and run programs.

**Note:** Once all daemons and getty processes for each terminal are started, we are at run level 2.

# Connecting to network and other services

The next step is to start up various daemons that support networking and other services. Once that is done, we are at run level 3 and the system is fully ready for use.

# Booting in xv6

The boot loader in xv6 is designed as follows.

- It is present in sector 0 of disk.
- It comprises 512 bytes having two parts
  - bootasm.S (8900)
    - Enters in 16 bit real mode, leaves in 32 bit protected mode
    - Disables interrupts
      - BIOS ISRs are not used
    - Enables A20 line
    - Loads GDT (only segmentation, no paging)
    - Sets stack to 0x7c00
    - Invokes bootmain
    - Never returns
  - bootmain.c (9017)
    - Loads the xv6 kernel from sector 1 to RAM starting at 0x100000 (1MB)
    - Invokes the xv6 kernel entry
      - _start present in entry.S
      - This entry point is known from the ELF header