Chapter 8 | Operating Systems

• Operating System ● Introduction (What is an Operating System?) ● Loading a program / How does a program run ● Boot Sequence ● Hardware Abstraction ● Virtual Machine and Resource Manager

CS102 Computer Environment

Operating system

Introduction (What is an Operating System?)

Think about some examples of useful programs you have used in your life. Maybe you think of Web browsers, games, Word processing etc. Web browsers should able to respond to keyboard and mouse inputs and communicate with the Internet through the network card. Games are even more interactive than Web Browsers (though it's possible to have games running within Web Browsers). What we observe is that any useful programs in the end involve some form of I/O. Here is a list of resources/functionality that any useful program might need:

- Memory Space
  - Specifically, space to store the code and data, without which no program can run.

- I/O Operations
  - To read inputs from keyboard/mouse
  - To print output to consoles or display graphics on screen.
  - To send data/information to another computer through the internet
  - Interface with webcams, printers, facsimile etc

- File Systems
  - Read/write files
  - Files can be on any kind of storage – USB drive, network drive, CD, DVD, hard disks etc

- Initialization
  - How does a program get to start?

Given all these considerations, together with the fact that most modern OSs support what is commonly known as multi-tasking i.e., multiple programs apparently running at the same time. We get an insight into what an Operating System is. Specifically, it is a program that acts as an intermediary between a user of a computer and the computer hardware. The following sections describe the various role/tasks of the OS does.

Loading a program/How does a program run

As described in the previous chapter, the CPU is an automatic machine i.e., once things are set in place. The Fetch—Decode—Execute—Writeback cycle is activated all the time. Thus what the computer does is dependent on the value of the program counter. The CPU simply executes the instruction that is found in the address pointed to by the PC. Now it should be noted that the "instructions" contained in the address pointed to by the PC may be rubbish and meaningless from our point of view. The "meaning" of instructions executed is irrelevant to the CPU. We need to get the CPU to "load" a program for it to perform meaningful tasks.

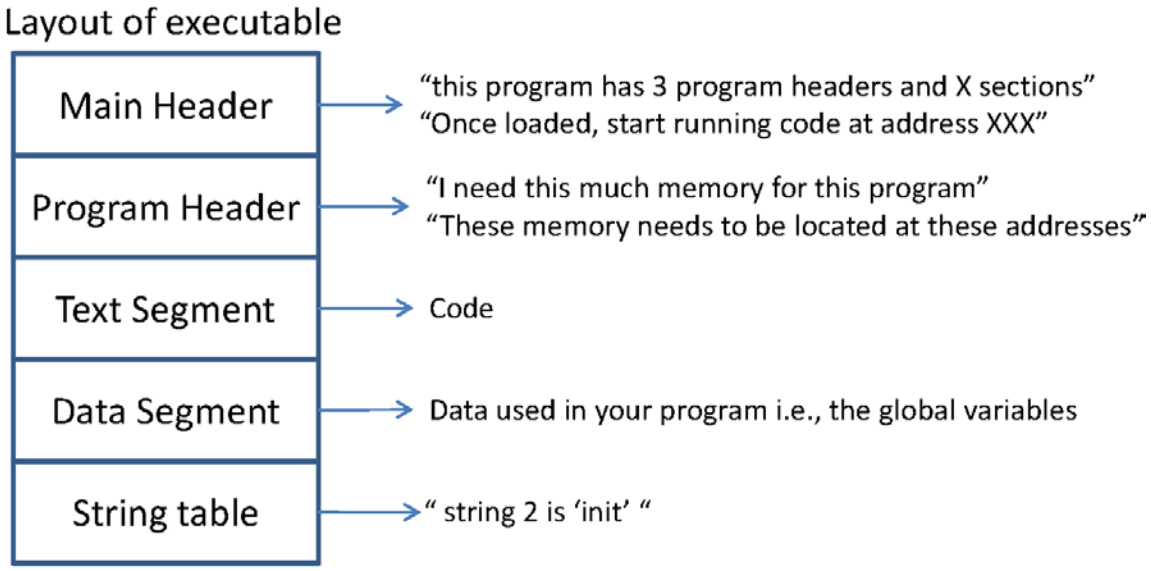

Figure 1: Executable layout

Before we understand how to load a program, we briefly describe the layout of an executable, which is the final form of a program after going through the stages of compilation and linking. An executable can be split into a few different sections, as shown in Figure 1. It should be noted that the executable would be in binary i.e., all 1s and 0s, not *human-readable*. However, a loader program would understand it. So there are these few sections that are basic to most executable formats: main header, program header, text segment, data segment and the string table.

- Main Header
  - Informs the loader of the layout i..e, how many sections, where is the starting instruction etc.
- Program Header
  - Informs the loader of memory usage requirements and the memory addresses the text segment and data segment should occupy

- Text Segment
    - As shown in Figure 1, it is the code of your program, including the libraries used by your program.
- Data Segment
    - This is where all the global variables are stored.
- String Table
    - Really an extension of the data segment, where all the strings used in the program are stored. For example, your code may contain a line printf("Hello World!\n");. The string "Hello World\n" will be stored in the string table.

So a loader program needs to do the following to run a program:

1. understand the executable format
2. Copy the text and data segments into the correct memory addresses.
3. Allocate space for Stack and heap for the new running program
4. Set the Program Counter value to the address of the starting instruction of the loaded program.
5. The newly loaded program runs

Now, having done all of that, a chicken and egg question remains: who shall load the loader program then? We turn our attention to how a computer system boots up.

Boot Sequence

1. Power-On

   The internal power supply turns on and initializes. It usually takes some time to initialize because it has to generate enough power for the rest of the computer. If it does not initialize correctly, it might lead to damage. Therefore, the chipset will generate a reset signal to the processor (the same as if you held the reset button down for a while on your case) until it receives the Power Good signal from the power supply.

2. Run BIOS-program

   The processor is pre-programmed to always look at the same place in the system BIOS ROM for the start of the BIOS boot program. This is normally located at FFFFFFFF0h. It should be noted that the BIOS program need not be loaded onto the RAM., instead the address FFFFFFF0h is mapped to be pointing to the first instruction of the BIOS program (which may jump to other parts of the BIOS program during execution).

3. What BIOS do

   - The BIOS performs the power-on self test (POST). If there are any fatal errors, the boot process stops.
   - The BIOS looks for the video card's built in BIOS program and runs it. The system BIOS executes the video card BIOS, which initializes the video card.
   - The BIOS then looks for other devices' ROMs to see if any of them have BIOSes and executes them as well.
   - The BIOS displays its startup screen.
   - The BIOS does more tests on the system, including the memory count-up test which you see on the screen.
   - The BIOS performs a "system inventory" of sorts, doing more tests to determine what sort of hardware is in the system.
   - If the BIOS supports the Plug and Play standard, it will detect and configure Plug and Play devices at this time and display a message on the screen for each one it finds.
   - The BIOS will display a summary screen about your system's configuration.
   - The BIOS begins the search for a drive to boot from looking searching through all drives (Floppy, hard disk, CD…).

4. What the BIOS do last – load a boot loader
    - After locating the boot drive, the BIOS looks for boot information to start the operating system boot process. Usually, it looks for a master boot record at cylinder 0, head 0, and sector 1 (the first sector on the disk). The boot loader (very small, only 512 bytes) is loaded copied into a location on the RAM and the program counter is set to point to the first instruction of the boot loader.
    - If no boot device at all can be found, the system will normally display an error message and then freeze up the system.
5. What the boot loader does
    - Typically, it loads an OS loader program that in turns loads the rest of the Operating System.
6. Operating System runs
    - The Operating System initializes by setting up necessary basic device drivers, detecting hardware and housekeeping activity. Typically, the OS finally loads and presents the user with either a command prompt or a GUI for login. After that, the OS is just waiting for user input. (Perhaps some scheduled periodic tasks gets to run from time to time).

We have come back in one full-circle. Once we get how a program is loaded, the operating system can be loaded in a similar fashion. The key is that the CPU starts executing from a specific address at initialization. Of course, we expect the operating system to be able to perform the task of a loader. That's how applications such as the web browser, games get to run in the OS i.e., they are loaded into RAM and executed as described above.

Hardware Abstraction

One of the main roles of the OS is to provide user software with hardware abstraction. Usually, the user software uses the hardware I/O through a defined set of Application Program Interface (API) e.g., a printf call leads somewhere down the line to a call to the Operating System to write the string on the screen. There are three main reasons why we need the OS to perform this role:

1. Interaction with hardware is complex

As indicated in the previous chapter, the CPU can interact with the I/O devices through either special I/O instructions or memory mapping. Allowing user software to interact with I/O devices directly require programmers to have this sort of knowledge. Furthermore, interaction with each I/O device requires certain handshaking protocols that are unique to each device type.

2. Reduce the need to duplicate code

Without the OS providing the API, not only does every programmer needs to write his own device driver code, the code that interacts with device I/O is duplicated for every program that runs in the computer system, thereby wasting precious memory resources.

3. Security issues

When user programs are allowed to interact with the hardware I/O directly, we run a security risk, exposing the system to be attacked by either malicious programs or simply badly written programs. OS acts as a security manager as well to prevent the system going down due to rogue software behavior.

Virtual Machine and Resource Manager

Every running program behaves as if it is the only program using the computer. While multiple programs can run at the same time, programmers do not need to consider their existence when programming. Rather, programmers assume that their program is the only one running in the entire system. In view of this, the operating system must be able to share and distribute resources among the running programs. Specifically, the execution of the CPU and the memory resources needs to be shared. However, it should be noted that the operating systems need to facilitate all these while maintaining security (see above, protection from rogue programs), high CPU utilization (the CPU spends mosts of its time doing actual useful work), fast response time (GUI programs respond quickly) etc. These goals are actually held in tension e.g., having security versus high CPU utilization.