<u>Linux Kernel Booting Process and System Initialization</u>

Today we are going to talk about a very famous term and that is booting process of an operating system. We all know that an operating system must itself be loaded into memory and it must be running before a user can load and execute different programs. The details may vary depending on the h/w platform and operating sytem, but roughly we can describe booting process into five phases:

- <u>BIOS/UEFI:</u> Basic Input Output System is now replaced by Unified Extensible Firmware Interface which performs POST and then as per the prioritized boot order mentionend in CMOS selects the boot disk to execute MBR.
- <u>MBR/GPT:</u> Master boot Record or Globally Unique Identifier Table (GPT) usually resides in the zero sector of the boot drive which is also called the Stage-1 boot loader.
- <u>GRUB:</u> GRand Unified Boot loader is the default boot loader of Linux, which is responsible for loading the kernel into memory.
- <u>Kernel:</u> After the kernel is loaded into memory it initializes itself, loads the initial RAM disk image containing a temporary file system with different Loadable Kernel Modules (LKM) that are used to load the actual/permanent root file system. Kernel then forks and executes system daemon (`systemd/init`) the grand-dady of all user space processes.
- <u>Systemd:</u> The system daemon then executes different processes/scripts to bring the system in its default state which is normally `graphical.tgt`. Thus, giving a user interface to the user of the computer to interact with it.

Note: Please do go through the man page of `boot`

1. <u>BIOS/UEFI Initialization</u>

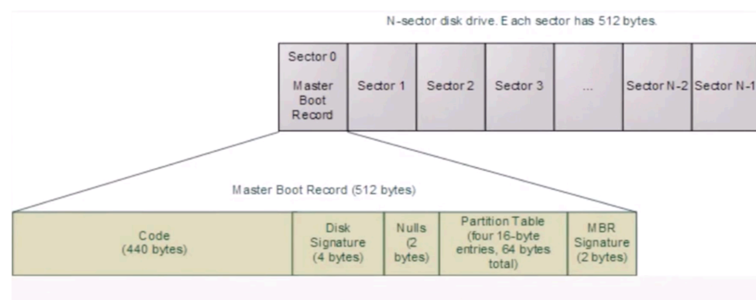
Things start rolling when we press the power button of our computer. Once the motherboard is powered up it initializes its own firmware (chipset and other tidbits), and tries to get the CPU running. In case of a 32-bit Intel CPU, the `eip` register holds a magical address that is called the reset-vector. For all modern x86 processors this is the

physical address `0xFFFFFFF0` (16 bytes below 4GiB). The instruction at this address is a jump instruction to the memory location mapped to the BIOS entry point (`0xF0000`), containing the BIOS code. BIOS code is there in Complementary Metal Oxide Semi-conductor (CMOS), a battery powered memory chip on the mother board. This contains the prioritized boot order which can be altered by user if desired. It also contains the hard ware clock of the system. The BIOS code then executes and perform the following tasks:

a. Performs Power-On-Self-Test (POST), which tests various h/w components attached with the system. A working video card takes us on monitor screen. In case of lack of video card / memory, it emits beep codes, e.g., repeated long beeps indicate memory problem.
b. After POST, the BIOS wants to boot up an OS, from hard disk, USB flash drive, Optical disk or may be from Network.
c. Finally, BIOS reads the first 512 bytes of bootable hard disk (zero sector) and loads the contents into memory location `0x7C00`

2. The Master Boot Record

The contents of the zero sector of hard disk (512 bytes) contains the code that is known as Stage-1 boot loader (446 bytes). Can be Linux specific, Windows specific and can be even a virus. Please note that MBR is not located inside any partition, rather precedes the first partition. The first 440 bytes contain bootable code, followed by disk signature of 4 bytes and then two bytes containing NULL. Then we have four entries of 16 Bytes each containing information about primary partitions, and finally there is MBR signature of 2 bytes.



- To view the Stage-1 boot loader, give following command:
```
$ dd if=/dev/sda bs=440 count=1 | hexdump -C
```
- To view the 2 Bytes MBR signature give following command:
```
$ dd if=/dev/sda bs=512 count=1 | tail -c 2 | hexdump -C
55 aa
```

- To view the 4 Bytes Disk signature, give following command:

```
$ dd if=/dev/sda bs=446 count=1 | tail -c 6 | hexdump -C
98 64 9b af 00 00
```

Now the problem is that this small piece of code inside MBR cannot load the kernel, as its is unaware of file system concept and requires a boot loader with file system driver. So, this 440 bytes of code is actually used to laod the actual boot loader (GRUB).

3. The Boot Loader

A boot loader is the first software program that runs when a computer starts. Different operating systems have support of different boot loaders. Linux support LInux Loader (LILO), and GRand Unified Boot loader (GRUB). The most latest Linux distributions these days support GRUB2, which can load all distributions of Linux, BSD UNIX, Mac OS X, and DOS. Micorsoft Windows support New Technology Loader (NTLDR).

One must not confuse these boot loader programs with operating system installer programs like anaconda for RHEL and ubiquity for Debian Linux.

Now there are three stages of GRUB2 booting process:

a. Stage-1: We have already seen the 446 bytes `boot.img` is stored in the MBR. It is configured to load the Stage-1.5 boot loader.
b. Stage 1.5: The code for Stage-1.5, called the `core.img` is of 30KB, which resides immediately after the MBR and before the first partition. This space is used to store file system drivers and loadable kernel modules which enable Stage-1.5 to load Stage-2 boot loader.
c. Stage 2: To describe the stage2 boot loader first see the contents of a Linux `/boot` directory

```
$ ls /boot/
initrd.img-3.13.0-43-generic
System.map-3.13.0-43-generic
config-3.13.0-43-generic
vmlinuz-3.13.0-43-generic
```

There is quite a useful stuff lying in the `/boot` directory. The four files of our interest are:

- **vmlinuz:** The compressed, bootable kernel image. On Linux systems, vmlinuz is a statically linked executable file that contains the Linux kernel in one of the object file formats supported by Linux**.**The letter z at the end denotes that it is compressed.
- **initrd.img:** The initial RAM disk; an early root filesystem that allows your kernel to bootsrap and get essential device drivers to get the final, official root filesystem
- **config:** A file that contains the configuration parameters for the kernel
- **System.map:** Symbol table used by kernel

If you happen to have multiple kernel versions available you may find these four files for all versions. Now this is the time that GRUB displays a list of all operating systems installed and wait for some time until it boots the default kernel with default command line arguments. Do view the contents of `/boot/grub/grub.cfg`, however, never make changes in this file. If you want to change the time out or the default kernel to be loaded make changes in the file `/etc/default/grub` instead.

Before we move on to the next step, let us talk about some command line arguments that can be passed to linux kernel. The Linux kernel accepts certain command-line options or boot time parameters at the moment it is started. In general, this is used to supply the kernel with information about hardware parameters that the kernel would not be able to determine on its own, or to avoid/override the values that the kernel would otherwise detect. Some of the important boot time parameters are:

- **`'init=...'`** This sets the initial command to be executed by the kernel. If this is not set, or cannot be found, the kernel will try `/sbin/init`, then `/etc/init`, then `/bin/init`, then `/bin/sh` and panic if all of this fails.
- **`'root=...'`** This argument tells the kernel what device is to be used as the root filesystem while booting, default is the `/dev/sda1`

- **'ro' and 'rw'** ro tells the kernel to mount the root file system as read onlly. This is done so that fsck(1) program can check and repair Linux file system. rw is the default.
- **panic=N'** By default a kernel do not reboot after a panic, but this option causes a kernel to reboot after N seconds
- **'reboot=[warm|cold][,[bios|hard]]'** By default a reboot is cold and hard. A cold reboot reset certain hardware, but might destroy not yet written data in a disk cache. A warm reboot may be faster. A hard reboot asks the keyboard controller to pulse the reset line low, but there is at least one type of motherboard where that doesn't work. The option 'reboot=bios' will instead jump through the BIOS.
- **debug'** This argument will cause the kernel to also print messages logged at level KERN_DEBUG.

  Note: For more details, students are advised to go through the man page of bootparam and try passing some of these arguments to kernel while it reboots.

4. Kernel Initialization

The steps performed in this phase are as follows:

- Kernel is loaded into memory
- It initializes/accesses the keyboard, monitor, disk controllers, timers … and sets up interrupt handling mechanisms.
- Now the kernel needs to mount the root file system, which may be is on a partition having capabilities like Logical Volume Management, RAID, Network File System. Unfortunately, these features are not compiled into the Linux kernel, rather are present as LKM in the /lib/modules/ directory which is present on the root file system itself. So, a 100$ question is “How the Linux kernel access the LKMs for mounting the root file system which are present on the root file system itself?” The answer is Initial RAM Disk, present in the /boot/initrd.img.

- So now the Linux kernel uncompress `/boot/initrd.img` and load/mount its contents in memory. Students can execute following command on the terminal to view the contents of this file:

  `$ lsinitramfs /boot/initrd.img--4.19.0-kali1-amd64`

  You can see almost all the directory hierarchy like `bin`, `dev`, `lib`, etc, which mirror the real/permanent file system. Moreover, there are lot of `.ko` files containing loadable kernel modules for various required tasks. So contents of `initrd` file serves as a temporary root file system and helps kernel to boot properly and load all the LKMs required to mount the actual root file system (be it on LVM or a RAID partition)
- Kernel then executes the command `pivot_root`, which alters the root partition from `initrd` to `/`. This removes the `initram` file system from memory and establishes the permanent file system.
- Kernel then initializes the scheduler with PID of zero.
- Kernel then forks and executes `/sbin/init` or `/bin/systemd`, which both are a soft link to /`lib/systemd/systemd` program. Now this `systemd`, which is said to be the grand dady of all processes is responsible for initializing and establishing the entire user space. The kernel itself moves in the background waiting to be called by other processes.

5. The Systemd Process

Once the Kernel initialization process is completed, the control is there with system daemon, previously known as the `init` process. The `systemd` run various scripts in the directories `/etc/rcx.d/`, where x is the runlevel. There are startup scripts with names starting with an S and some kill scripts having names starting with a K and then followed by numbers. If the system is to enter in runlevel5, then all the S-scripts are executed in sequence to start different userspace processes and to carry out system initialization. For further details, students are advised to go through the man pages of `runlevel`, `systemd`, `systemctl`. More on it later… ☺