Linux Booting Process

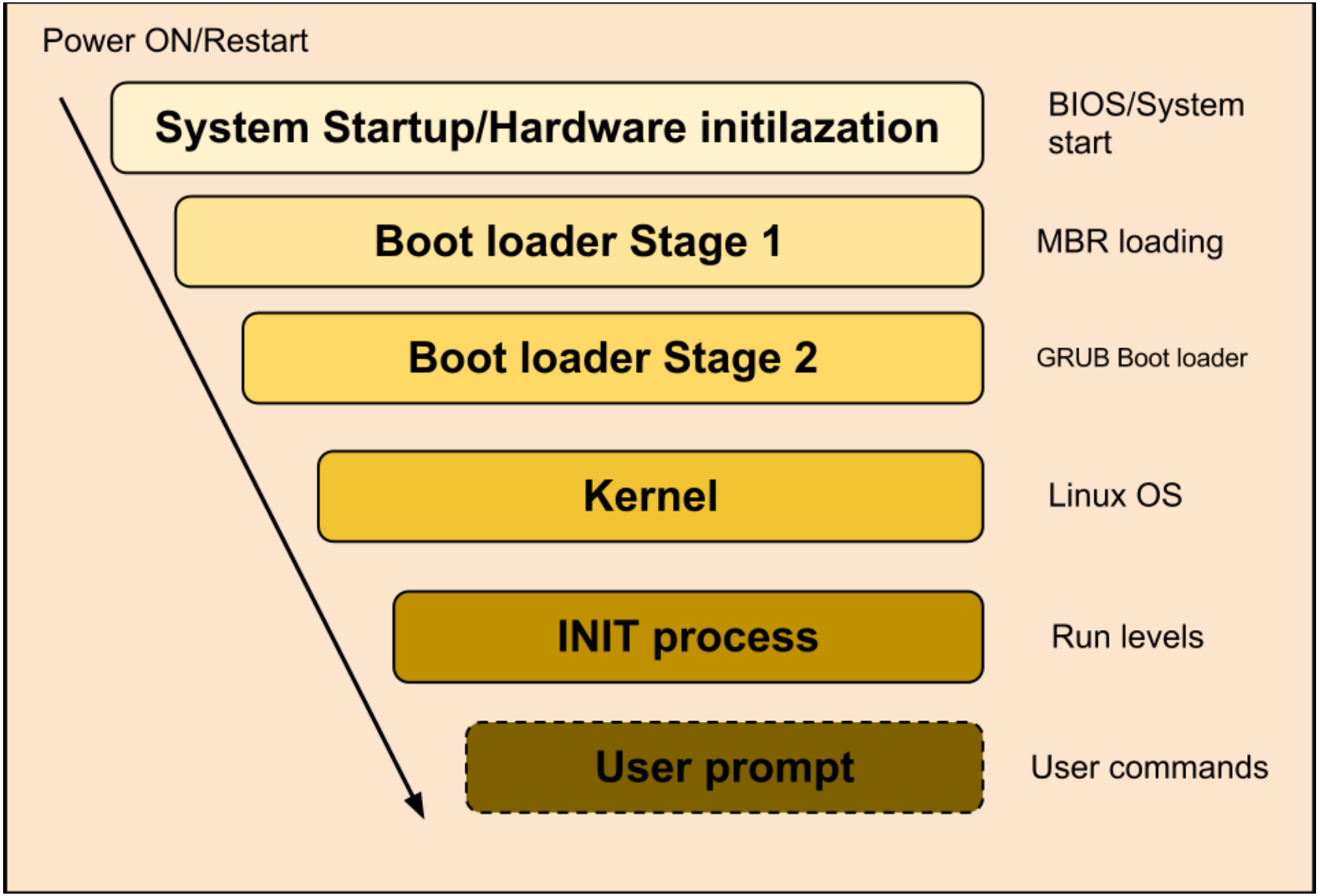
Roadmap

- **Linux Booting Process-** overview
- **System startup-** booting BIOS POST,
- **Bootloader Stage 1-** Bootsector and setup,
- **Bootloader Stage 2-** Using LILO/GRUB as bootloader,
- **Kernel**- High level initialization, SMP bootup on x86,
- **INIT process-** freeing initialization data and code,

What is Booting

- Booting is a process or set of operations that loads and hence starts the operating system, starting from the point when user switches on the power button.

Linux Booting Process



Linux Booting(in Detail)

1. System Startup- BIOS POST

This is the first stage of booting process. When you power on/Restart your machine the power is supplied to all the devices connected to that machine such as Motherboard HDD's, CD/DVD-ROM, Mouse, keyboard etc. The most intelligent device in the computer is Processor(CPU), when supplied with power will start running its sequence operations stored in its memory. The first instruction it will run is to pass control to **BIOS(Basic Input/Output System)** to do **POST(Power On Self Test)**. Once the control goes to BIOS it will take care of two things

- **Run POST operation.**
- **Selecting first Boot device.**

Linux Booting(in Detail)

2. **MBR Loading**

- MBR(Master Boot recorder) is a location on disk (512bytes)which have details about
  - **Primary boot loader code(This is of 446 Bytes)**
  - **Partition table information(64 Bytes)**
  - **Magic number(2 Bytes)**

Which will be equal to 512B (446+64+2)B.

- It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
- This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
- It contains information about GRUB (or LILO in old systems).
- So, in simple terms MBR loads and executes the GRUB boot loader.

Linux Booting(in Detail)

3. **Boot Loader stage 2(Grub Loader)**

- In this stage GRUB(Grand Unified Bootloader) which is located in the first 30 kilobytes of hard disk immediately following the MBR is loaded into RAM for reading its configuration and displays the GRUB boot menu (where the user can manually specify the boot parameters) to the user.
- GRUB loads the user-selected (or default) kernel into memory and passes control on to the kernel. If user do not select the OS, after a defined timeout GRUB will load the default kernel in the memory for starting it.
- GRUB has the knowledge of the filesystem (the older Linux loader LILO didn't understand filesystem).
- GRUB configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this). The following is sample grub.conf of CentOS.
- #boot=/dev/sda default=0 timeout=5 splashimage=(hd0,0)/boot/grub/splash.xpm.gz hiddenmenu title CentOS (2.6.18-194.el5PAE) root (hd0,0) kernel /boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/ initrd /boot/initrd-2.6.18-194.el5PAE.img

As you notice from the above info, it contains kernel and initrd image.

So, in simple terms GRUB just loads and executes Kernel and initrd images.

Linux Booting(in Detail)

4. **Kernel**

- Once the control is given to kernel which is the central part of all your OS and act as a mediator of hardware and software components.
- Kernel once loaded into to RAM it always resides on RAM until the machine is shutdown.
- Once the Kernel starts its operations the first thing it do is executing INIT process. The process is as follows:
  - Mounts the root file system as specified in the “root=” in grub.conf
  - Kernel executes the /sbin/init program
  - Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a ‘ps -ef | grep init’ and check the pid.
  - initrd stands for Initial RAM Disk.
  - initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

Linux Booting(in Detail)

5. INIT

- This is the main stage of Booting Process
- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
  - 0 – halt
  - 1 – Single user mode
  - 2 – Multiuser, without NFS
  - 3 – Full multiuser mode
  - 4 – unused
  - 5 – X11
  - 6 – reboot
- Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.
- Execute 'grep initdefault /etc/inittab' on your system to identify the default run level
- If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
- Typically you would set the default run level to either 3 or 5.

Linux Booting(in Detail)

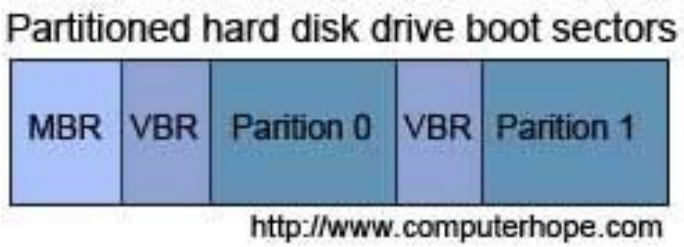
6. **User Prompt**

This is actually not part of booting process but thought of including it here for better understating. Once the Kernel get the control it start multiple instances of "getty" which waits for console logins which spawn one's user shell process and gives you user prompt to login.

Booting:Bootsector and setup

What is bootsector?

- A sector on the computer hard drive or other bootable disk drive that contains instructions on how to boot from the drive. IBM PC disk drives have different types of **boot sectors**; the master boot record (MBR), which is the first sector of a portioned hard drive that and the volume boot record (VBR), which contains partition information at the beginning of each partition.



The bootsector used to boot Linux kernel could be either:

- Linux bootsector (arch/i386/boot/bootsect.S),
- LILO (or other bootloader's) bootsector, or
- no bootsector (loadlin etc)

Booting: Bootsector and setup

We consider here the Linux bootsector in detail. The first few lines initialise the convenience macros to be used for segment values:

- 29 SETUPSECS = 4 /* default nr of setup-sectors */
- 30 BOOTSEG = 0x07C0 /* original address of boot-sector */
- 31 INITSEG = DEF_INITSEG /* we move boot here - out of the way */
- 32 SETUPSEG = DEF_SETUPSEG /* setup starts here */
- 33 SYSSEG = DEF_SYSSEG /* system loaded at 0x10000 (65536) */
- 34 SYSSIZE = DEF_SYSSIZE /* system size: # of 16-byte clicks */

(the numbers on the left are the line numbers of bootsect.S file) The values of DEF_INITSEG, DEF_SETUPSEG, DEF_SYSSEG and DEF_SYSSIZE are taken from include/asm/boot.h:)

LILO(Linux Loader) as Bootloader

- Not depend on a specific file system
- Can boot from harddisk
- Up to 16 different images
- Must change LILO when kernel image file or config file is changed

LILO(Linux Loader) as Bootloader

There are several advantages in using a specialised bootloader (LILO) over a bare bones Linux bootsector:

- Ability to choose between multiple Linux kernels or even multiple OSes.
- Ability to pass kernel command line parameters (there is a patch called BCP that adds this ability to bare-bones bootsector+setup).
- Ability to load much larger bzImage kernels - up to 2.5M vs 1M.
- Old versions of LILO (v17 and earlier) could not load bzImage kernels. The newer versions (as of a couple of years ago or earlier) use the same technique as bootsect+setup of moving data from low into high memory by means of BIOS services. Some people (Peter Anvin notably) argue that zImage support should be removed. The main reason (according to Alan Cox) it stays is that there are apparently some broken BIOSes that make it impossible to boot bzImage kernels while loading zImage ones fine.The last thing LILO does is to jump to setup.S and things proceed as normal.

LILO v\s GRUB

LILO

- LILO has a simpler interface and is easier.
- The LILO bootstrap process involves locating the kernel by in essence pointing to the first logical-sector of the Kernel file.
- LILO has no interactive command interface.
- LILO does not support booting from a network.
- LILO stores information regarding the location of the operating systems it can to load physically on the MBR. If you change your LILO config file, you have to rewrite the LILO stage one boot loader to the MBR

GRUB

- GRUB is more featured and handles odd configurations better.
- The GRUB bootstrap process is more filesystem aware and can locate a kernel file in a filesystem without having to specify a logical-sector.
- GRUB has interactive command line.
- whereas GRUB does.
- Compared with GRUB, this is a much more risky option since a misconfigured MBR could leave the system unbootable. With GRUB, if the configuration file is configured incorrectly, it will simply default to the GRUB command-line interface.

High Level Intialization

- By "high-level initialisation" we consider anything which is not directly related to bootstrap, even though parts of the code to perform this are written in asm, namely arch/i386/kernel/head.S which is the head of the uncompressed kernel. The following steps are performed:
- Initialise segment values (%ds = %es = %fs = %gs = __KERNEL_DS = 0x18).
- Initialise page tables.
- Enable paging by setting PG bit in %cr0.
- Zero-clean BSS (on SMP, only first CPU does this).
- Copy the first 2k of bootup parameters (kernel commandline).
- Check CPU type using EFLAGS and, if possible, cpuid, able to detect 386 and higher.
- The first CPU calls start_kernel(), all others call arch/i386/kernel/smpboot.c:initialize_secondary() if ready=1, which just reloads esp/eip and doesn't return.

SMP Bootup on x86

- On SMP, the BP goes through the normal sequence of bootsector, setup etc until it reaches the start_kernel().
- and then on to smp_init() and especially src/i386/kernel/smpboot.c:smp_boot_cpus(). Thesmp_boot_cpus() goes in a loop for each apicid (until NR_CPUS) and calls do_boot_cpu() on it.
- What do_boot_cpu() does is create (i.e. fork_by_hand) an idle task for the target cpu and write in well-known locations defined by the Intel MP spec (0x467/0x469)
- the EIP of trampoline code found in trampoline.S. Then it generates STARTUP IPI to the target cpu which makes this AP execute the code intrampoline.S.

Freeing initialisation data and code

When the operating system initialises itself, most of the code and data structures are never needed again. Most operating systems (BSD, FreeBSD etc.) cannot dispose of this unneeded information, thus wasting precious physical kernel memory. The excuse they use (see McKusick's 4.4BSD book) is that "the relevant code is spread around various subsystems and so it is not feasible to free it". Linux, of course, cannot use such excuses because under Linux "if something is possible in principle, then it is already implemented or somebody is working on it".

- Linux kernel can only be compiled as an ELF binary, and now we find out the reason (or one of the reasons) for that. The reason related to throwing away initialisation code/data is that Linux provides two macros to be used:
- __init - for initialisation code
- __initdata - for data