The GNU/Linux Boot Process

Author: Omkaram Venkatesh

2024, January 15th

Agenda: As I am on my way to build LFS (Linux From Scratch)/(Own Custom Linux Operating System), having this knowledge is important. This helps me to not get lost in the process of mindlessly compiling the Kernel sources and other sources for hours during the LFS project which I might soon take up. This is just a write up to self just to strengthen my current knowledge. Beware! I could be wrong about few points and so this doc cannot be fully trusted for professional use.

(Just talking to my future self here) Hey! If you are wrong about anything here, then correct it.

Terminologies

Firmware: Its a software which resides on all digital electronic chips which more or less work the hardware directly.

BIOS: BIOS is a firmware which resides on Motherboard EEPROM chip (a non volatile chip) which is the first program that executes after a system power's up.

UEFI: This is also a firmware similar to BIOS. Not every Motherboard supports it. Most modern systems have BIOS and UEFI on chip as modes, and leave it to the user to decide which one to use to start the boot process and this process is called as "choosing the mode". The user can choose to boot either in BIOS mode or UEFI mode.

Kernel: A kernel is a modular piece of software running with the highest privileges on machine which is the core of an Operating System. The Kernel's job is to manage, allocate, monitor processes, interacting with drivers/modules like loading and unloading, interacting directly with hardware and also talking to apps on the user space with the highest privleges. You can find it in the /boot directory of your file system.

Drivers: These are individual special programs which operates directly inside the Kernel and helps the kernel interact with the Hardware's firmware created for a specific device/hardware in mind. List of common drivers are, File system drivers (to support ext2, ntfs, ext4, brtfs etc), Display drivers, Audio drivers, Bluetooth drivers, Keyboard drivers, Network drivers, Printer drivers etc

Bootloader: It's the software which boots the operating system and it is responsible for loading the Kernel into the Memory. Bootloaders are typically are two staged or multi staged. The Stage-2 Bootloaders resides in /boot directory of the root of the fs. The Stage-1 Bootloaders are installed on to the boot sector such as MBR or GPT. Stage-2 Bootloader aka the ‘real Bootloader’ is called by the Stage-1 Bootloader. This process is called "Chaining". Popular bootloaders are Grub, Grub2, Syslinux, Brug, Lilo etc.

MBR: Aka the Master Boot Record is first few hundred bytes (512 bytes) of a configured Hard drive. It's a reserved space at the beginning of the drive which contains info on how the partitions on the drive are organised. It also contains the Stage-1 Bootloader. It is also referred as a "Partition Table/Partition Scheme" as it holds the start and end sectors of each partition and other critical metadata of the said partitions.

GPT: GUID Partition table is a modern replacement of the MBR partition scheme. Unlike MBR which holds all the partition metadata in the beginning bytes/sector of the drive which is considered bad because in situations where this first sector gets corrupted, the user could potentially lose either none, some or all the data even though the data is virtually still there in the drive. It happens because the when the MBR gets corrupted, the BIOS/UEFI would not be able to tell from which starting and ending sectors the boot partition and the root partition exist, to boot or mount, and it gets hard to retrieve the data without this metadata, but in the case of the GPT, the GPT holds the partition metadata by distributing them in the first or the end sectors of all the existing partition. It also does data integrity checks periodically. If the first Partition which is typically the boot partition somehow gets corrupted, there is still scope to boot and save the data because of the redundancy/mirroring mechanism GPT follows. GPT also has a version of the MBR called a "Protective MBR" which is useful when the user is booting legacy operating system which only recognises the MBR partion scheme and consider GPT as alien. Typically, the BIOS recognises MBR, and the UEFI recognises both MBR and GPT, but the user is

recommended to go with the compatible partition scheme for the right firmware. UEFI chooses to work with GPT for compatibility reasons. Awesome yeh!

***InitRamfs/Initrd*:** InitRamfs short form for "Initial RAM file system" is a simple and minimalist file system which is the evolved version of the old Initrd aka the "Initial RAM disk". It's purpose is to simplify complicated tasks during the kernel boot process which the kernel directly cannot do. The InitRamfs mounts the real root file system aka "/" through "Init" file. It contains device drivers, systemd, init scripts, preload scripts, ld_linux.so files etc. The mini fs it contains has a Init file, /bin, /usr, /sbin, /etc, /lib64, /sbin, /scripts, /var, /run, /conf directories.

CPIO: Copy Input and Output is a archive file format. The InitRamfs is deployed using this file format.

Operating System (OS): It's a complex suite of applications, desktop environment (which comes with a suite of apps and window manger(s)(like wayland, hyprland, Xorg), display manager(s)(like Lightdm, sddm), drivers, and the kernel all combinedly working together and being referred as a single unit.

File System (fs): It can be many things depending on the context and who is naming it. In one context it is referring to the /usr, /root, /sys, /bin, /proc etc locations on the disk. In another context it is talking about the Partition type/format such as ext4, btrfs, zfs etc. Note: GPT and MBR are not filesystems, partitions or partition types. They are called "Partition schemes".

The boot process simplified

1. When the user powers the machine, let it be whatever the underlying architecture is (x86, arm), the EEPROM chip containing the BIOS or the UEFI firmware loads it into the memory (RAM) directly and tells the CPU to set instruction pointer to the beginning of the firmware program and start executing.

2. Now this firmware contains the instructions telling the CPU how to talk to the Harddrive (HD), but before doing that it also has instructions to do some Initial checks such as detecting devices and see whether all the devices are active, properly setup without any issues. If the firmware sees any problem which any devices, such as no power to HD etc, it simply exits. The firmware gathers all the information about the devices which are attached to the motherboard. It is programmed to look for the "bootable devices" in a sequence. In BIOS menu the user can choose the order of these devices. For example, sometimes the user would like to boot from the bootable USB stick first and if the stick is absent then it looks for CD, DVD, and finally the HD. This is how I manage to run Linux Mint or Debian off a USB stick which is the first bootable device preference on my personal machine.

   Once the bootable device is detected and set, the firmware always looks into the first sector/boot sector which is also known as MBR. The MBR is not a file system but is a tiny file represented as a sequence of bytes, 512 to be exact. It does not hold the complete bootloader within itself but it contains part of the bootloader and is something what the BIOS/UEFI refers to as a "boot signature". This signature is a special format (DOS in case of BIOS) which contains the initial boot code (Stage-1) which points to a Stage-2 boot code aka the ‘main bootloader’ which resides in the root / (aka root partition), whereever it is on the Harddrive, and it also containts the partition table data.

   So MBR contains the stage1 boot loader (440 bytes) and the partition table data (72 bytes) in the exact sequence. Interestingly, the user can also choose to put the root partition not at the first sectors of the HD, but at the last sector as well. This proves that the main bootloader doesn't necessarily have to be placed in the first parition of the bootable device (HD). The firmware instructs the cpu to copy the MBR which is on the first sector of the HD onto the RAM, because firmware always decide to a;ways look at this first sector no matter what (kinda hard coded) and sets the IP to start the cpu executing from there onwards.

3. As said before, the MBR contains the Stage-1 Bootloader code (which helps to locate the stage 2 boot loader on the disk) and partition information, and when the cpu executes the Stage-1, the IP jumps to the Stage-2 main bootloader which is residing in the /boot directory. Interestingly the Kernel code img file (vmlinuz) is also located in the same /boot location. So, the control goes to the main bootloader say Grub, and Grub starts executing it's code. The Grub has what is known as configuration file /boot/grub2/grub.cfg and also default configuration files /etc/default/grub. This is a file which contains what are the kernels/operating systems installated on the disk. The cfg file is mostly auto generated, because Grub is intelligent enough to probe for all the list of bootable partitions, Kernels and InitRamfs installed on a single disk. In case Grub is unable to locate the list of (Oses) bootable partitions on the disk, we have to intervene manually, and that is where os-prober comes to rescue. You might have (I have) used os-prober during your first dual boot installation. After successfully detecting and rebooting the next time, the Grub cfg will contain the new entries. The user is presented options to choose which kernel/Os to load into memory and start with amound several, and Grub handover’s the execution control to the Kernel.

4. Say you have Windows and Linux dual booted on your machine and you choose to go with the Linux Mint OS on Grub menu. Then the Linux kernel which is located in /boot is loaded into the memory and the control is handed over to it and Grub's work is complete. The bootloader also loads something called the InitiRamfs or InitRd alongside the kernel and more on that later. The Linux kernel now starts its execution which is also confusingly called as the "Kernel boot process" or the "boot process".

   Now comes the strange and the important part. It was a headache for me to figure it out the first time. So don't be upset if you forget this and had to come back to take a look at this doc.

5. The Linux kernel is modular by nature and it build by its developers in such a way that it is lightweight, robust and generic. This means that Linus Torvald the creator of the Linux Kernel doesn't intend to shop all the possible drivers specific to all possible devices in the world and ship them along with the kernel. Only the bare essentials default drivers are provided within the kernel package/image. All the essential drivers let it be hardware or software drivers are to be provided by the maintainers of a linux distribution aka (Distros).

   (Let's see if I can make one of my own in the future using LFS and BLFS).

   If a certain linux distribution does not provide the necessary drivers, then certain programs cannot work and certain hardware cannot be accessed. And this is why if you remember, back in the golden days of Windows 7, people had to download the drivers from the homepage of the Laptop/Desktop manufacturers like Dell or HP. So hardware drivers are mostly provided by the manufacturer for specific operating system and chipset architecture in mind, whereas software drivers are provided by the Os/distributions like Windows, Ubuntu, Linux Mint, Fedora, Arch etc. Now a days even the hardware drivers are provided by these Open source Distributions because they are "Open source". This is where 'Richard Stallman the Great' disagrees.

   Slighty going off topic here a bit to set some context. He complained that the Ubuntu, the most widely used Linux Distro by Canonical is "Open Source" but not "Free Software" and hence is not endorsed as FOSS complaint. By "Free" as in "Freedom". Because, Ubuntu like most Linux distros, packages proprietary NVIDIA drivers, which God only knows what they are actual doing in code. And Linus once publicly said the NVIDIA the company is a trouble maker and their worst vendor. And I trust him.

   So, basically in the case of GNU/Linux, the Distros provide majority of the device drivers, either hardware or software. These drives are to packaged into what we call the InitRamfs CPIO archive file. The job of the InitRamfs or InitRd is to do those things which the Linux kernel cannot by itself. This is where things get a bit messy to understand. But first we should ask "***what exactly cannot the Linux kernel do by itself*?**" And, "***Is this InitRamfs mandatory*?**" for the system fully bootup?

   But first, to answer the first question we must ask another question, "***what does the Linux kernel do?***". It's simple. The kernel mounts the root file system aka / for the userspace to exist and also loads and unloads drivers dynamically as modules during the runtime of the OS. The Userspace is the imaginary space where all the apps or programs which the users runs. Now, we can ask another question, "***Why can't the kernel load all the modules statically during the boot process itself instead of doing it during the userspace runtime?***" The answer to that is, "Bloat". If we do that, the kernel gets bloated heavily and moreover the kernel should be dedicated programmed in that way to load it's drivers directly from within. Luckily, that's not how the Linux kernel operates. Secondly, it is totally a waste of memory to load all the modules into kernel and keep it hanging like that all the time simply because not all programs the user is currently running require all the driver modules at all the times. For example, I may choose to not use the audio, and network drivers today, so why to load them during boot? We might simply think we should unload them if not in use after boot. But there lies the **egg and chicken problem.** The kernel having modules built-in cannot unload the modules itself because the modules are part of itself. You cannot instruct to unload something which is a part of you and still expect to be working. The kernel once loaded into the memory having the modules also within in the same memory cannot ask the cpu to unload the modules memory without risking to unload and crash itself. Just think. Say you are the Kernel. How can you ask the cpu to delete some memory while you are the one who is holding on to that memory. It's simply not possible. Infact, the same logic applies for loading the modules too. If the kernel has built-in modules, how can it extract and load parts of itself. This is like asking a compiled exe program to extract or load json or text resource files contained within itself dynamically and start working on it. It's called "being reflexive" and it's highly complicated to achieve, even in dumb and verbose languages like Java. If you still insist having built-in modules and risk them having in memory throughout the lifetime of the OS execution, then we have to ask the bootloader to do that job (which is not it's job) (which also means the Bootloader has to be awake and running infinitely), or create a custom linux kernel code to statically link the modules and so there won't be the concept of loading and unloading at all. So you got the answer to the first question, "***what exactly cannot the Linux kernel do by itself*?**".

   Now, let's ask the second question again. ***Is the InitRamfs mandatory?*** No. It's optional. Part of the job of the InitRamfs is to mount the filesystem and handover the control to the Init process called the PID1. All userspace

processes spawn off from PID1 as their parent or grandfather. Like my grandfather Manu aka FUC :) without which we all don't exist.

But the Linux kernel can also directly mount and unmount the rootfs, but the challenge it has to overcome is to call onto the file system command such as mount, umount. So, the kernel must also come with such app binaries as well if we want to do it that way. Interestingly, there are people who do such things and achieved it. These are the people who has really good knowledge on the kernel working and customising it. But still, even with good knowledge there are stuff even more hard to achieve. Such a mounting an encrypted rootfs etc. Interestingly, just for this reason, people created a saviour.

6. Here comes the **InitRamfs** to save the day. The name says it all. It is the initial file system that is loaded onto the memory since the boot. Its fs contains some essential app binaries, something similar to what you can find in /bin, /usr/bin, /etc, /lib64, /sbin, /scripts, /var, /run, /conf etc or busybox. Its not really a real fs, because for a real fs, there is a load of things needed to be check boxed such as having a file system partition type etc. Its a pseudo fs. It is also used a helping shell if something fails between the Kernel loading and the Init process begin. The InitRamfs typically created by specific Distro companies. It is served as a CPIO file and it is loaded into the memory simultaneously by the Grub Bootloader while Grub is also loading the kernel into memory. The Kernel starts using the driver modules from this mini fs throughout the execution of the userspace by staying active and uses the "mount" app binary to mount the rootfs. The kernel then executes the "init" script in the InitRamfs as PID1 using exec() and fork(), therefore marking the beginning of the userspace and it later hands over a pseudo control to the systemd (which is also referred as the "Real init"). Note: I noticed that the InitRamfs is much greater in size when compared to the vmlinuz kernel image. 10X greater.

   Just to see the contents inside the InitRamfs, you can use commands like `**lsinitrd --unpack initrd.img**` or `**cpio -ivd < initrd.img**` if the file format is an cpio archive, or directly use `**unmkinitramfs initrd.img target-export/**`.

   Just forget these stupid commands. Remembering commands is not the Linux way. Well, That's It.

Lets summarise the boot working flow again:

- The EEPROM chip loads the BIOS or the UEFI firmware to the memory.
- The BIOS or the UEFI probes for **bootable devices** and calls to load the MBR or GPT into memory. The MBR or GPT has a Stage-1 Bootloader installed like Grub. The Stage 1 Bootloader finds the main Stage 2 Bootloader on disk using the boot signature and the partition table information and executes it.
- Grub's stage-2 Bootloader looks at its cfg file, probes for **bootable partitions** and asks the user to choose the OS in case the machine is dual booted or it defaults to one OS provided in the cfg file.
- The Grub loads the kernel and InitRamfs and hands over the control to the kernel, and completes its execution. The kernel unloads the Grub from memory.
- The kernel using the Init script of the InitRamfs fs starts the Init process as PID 1 and won't die until the user power off the system.
- Note: In the above please notice the highlighted text **bootable devices** vs **bootable partitions**. There is a huge difference between the two. One is the literally devices (like HD or USB sticks etc) and the other is bootable locations on a single drive divided as several partitions (like Windows boot partition, Linux boot partition etc).