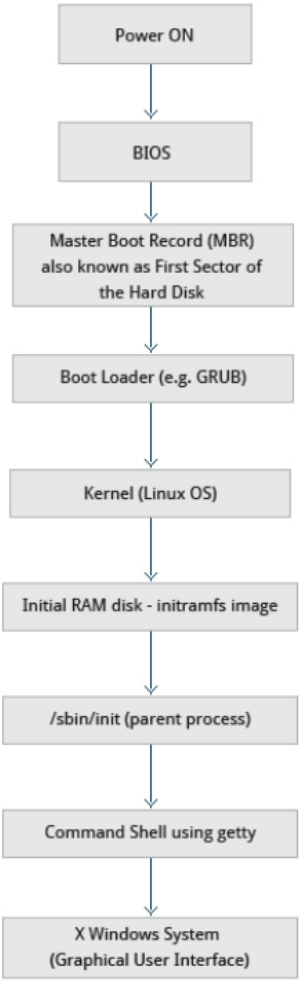
Chapter 3

The boot process
The Linux boot process is the procedure for initializing the system. It consists of everything that happens from when the computer power is first switched on until the user interface is fully operational.

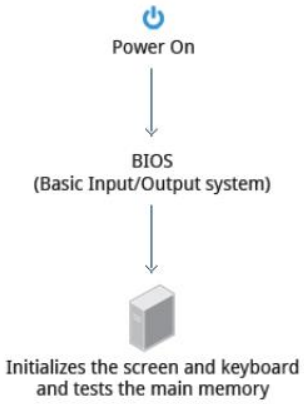


The Boot Process

BIOS

Starting an x86-based Linux system involves a number of steps. When the computer is powered on, the **B**asic **I**nput/**O**utput **S**ystem (**BIOS**) initializes the hardware, including the screen and keyboard, and tests the main memory. This process is also called **POST** (**P**ower **O**n **S**elf **T**est).

The BIOS software is stored on a ROM chip on the motherboard. After this, the remainder of the boot process is controlled by the operating system (OS).

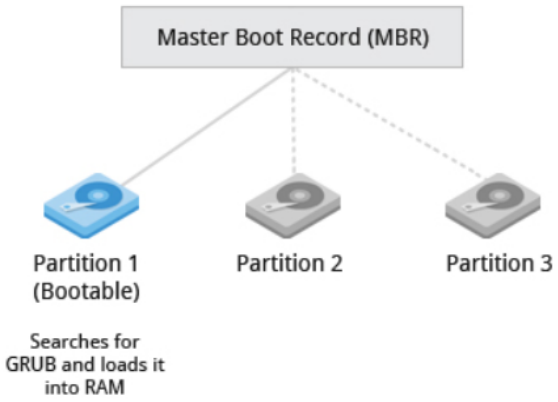


BIOS

Master Boot Record (MBR) and Boot Loader

Once the POST is completed, the system control passes from the BIOS to the **boot loader**. The boot loader is usually stored on one of the hard disks in the system, either in the boot sector (for traditional BIOS/MBR systems) or the **EFI** partition (for more recent (Unified) **E**xtensible **F**irmware **I**nterface or **EFI/UEFI** systems). Up to this stage, the machine does not access any mass storage media. Thereafter, information on date, time, and the most important peripherals are loaded from the CMOS values (after a technology used for the battery-powered memory store which allows the system to keep track of the date and time even when it is powered off).

A number of boot loaders exist for Linux; the most common ones are **GRUB** (for **GR**and **U**nified **B**oot loader), **ISOLINUX** (for booting from removable media), and **DAS U-Boot** (for booting on embedded devices/appliances). Most Linux boot loaders can present a user interface for choosing alternative options for booting Linux, and even other operating systems that might be installed. When booting Linux, the boot loader is responsible for loading the kernel image and the initial RAM disk or filesystem (which contains some critical files and device drivers needed to start the system) into memory.

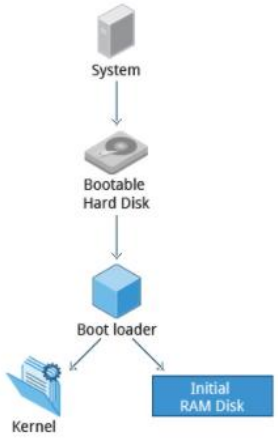


Boot Loader in action

The boot loader has two distinct stages:

For systems using the BIOS/MBR method, the boot loader resides at the first sector of the hard disk, also known as the **M**aster **B**oot **R**ecord (**MBR**). The size of the MBR is just 512 bytes. In this stage, the boot loader examines the **partition table** and finds a bootable partition. Once it finds a bootable partition, it then searches for the second stage boot loader, for example GRUB, and loads it into RAM (Random Access Memory). For systems using the EFI/UEFI method, UEFI firmware reads its Boot Manager data to determine which UEFI application is to be launched and from where (i.e. from which disk and partition the EFI partition can be found). The firmware then launches the UEFI application, for example GRUB, as defined in the boot entry in the firmware's boot manager. This procedure is more complicated, but more versatile than the older MBR methods.

The second stage boot loader resides under **/boot**. A splash screen is displayed, which allows us to choose which operating system (OS) to boot. After choosing the OS, the boot loader loads the kernel of the selected operating system into RAM and passes control to it. Kernels are almost always compressed, so its first job is to uncompress itself. After this, it will check and analyze the system hardware and initialize any hardware device drivers built into the kernel.

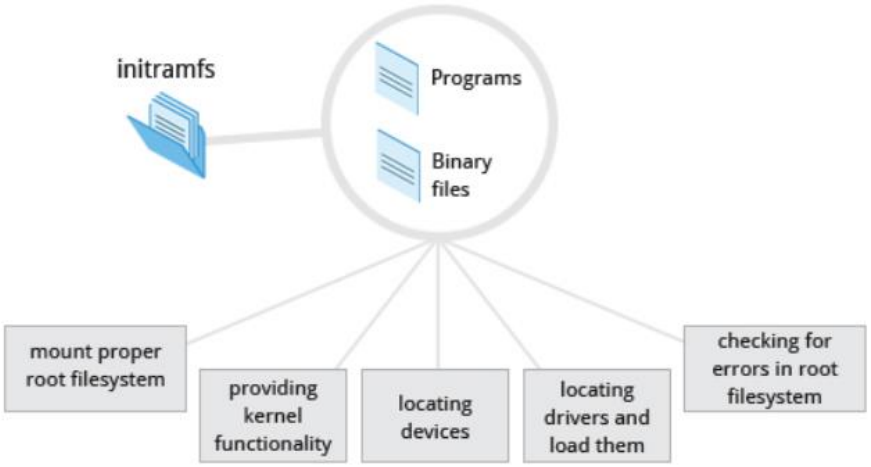


Initial RAM disk

The **initramfs** filesystem image contains programs and binary files that perform all actions needed to mount the proper root filesystem, like providing kernel functionality for the needed filesystem and device drivers for mass storage controllers with a facility called **udev** (for **u**ser **dev**ice), which is responsible for figuring out which devices are present, locating the device drivers they need to operate properly, and loading them. After the root filesystem has been found, it is checked for errors and mounted.

The **mount** program instructs the operating system that a filesystem is ready for use, and associates it with a particular point in the overall hierarchy of the filesystem (the **mount point**). If this is successful, the initramfs is cleared from RAM and the init program on the root filesystem (**/sbin/init**) is executed.

init handles the mounting and pivoting over to the final real root filesystem. If special hardware drivers are needed before the mass storage can be accessed, they must be in the initramfs image.



Text-mode login prompts

Near the end of the boot process, **init** starts a number of text-mode login prompts. These enable you to type your username, followed by your password, and to eventually get a command shell. However, if you are running a system with a graphical login interface, you will not see these at first.

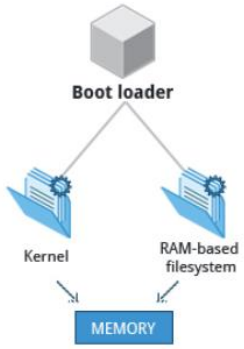
Usually, the default command shell is **bash** (the **GNU B**ourne **A**gain **Sh**ell), but there are a number of other advanced command shells available. The shell prints a text prompt, indicating it is ready to accept commands; after the user types the command and presses **Enter**, the command is executed, and another prompt is displayed after the command is done.

ARUNDHATI BANDOPADHYAYA (arundhati.bandopadhyaya@gmail.com)

Linux Kernel

The boot loader loads both the **kernel** and an initial RAM–based file system (initramfs) into memory, so it can be used directly by the kernel.

When the kernel is loaded in RAM, it immediately initializes and configures the computer's memory and also configures all the hardware attached to the system. This includes all processors, I/O subsystems, storage devices, etc. The kernel also loads some necessary user space applications.



/sbin/init and services

Once the kernel has set up all its hardware and mounted the root filesystem, the kernel runs **/sbin/init**. This then becomes the initial process, which then starts other processes to get the system running. Most other processes on the system trace their origin ultimately to **init**; exceptions include the so-called kernel processes. These are started by the kernel directly, and their job is to manage internal operating system details.

Besides starting the system, **init** is responsible for keeping the system running and for shutting it down cleanly. One of its responsibilities is to act when necessary as a manager for all non-kernel processes; it cleans up after them upon completion, and restarts user login services as needed when users log in and out, and does the same for other background system services.

Traditionally, this process startup was done using conventions that date back to the 1980s and the System V variety of UNIX. This serial process has the system passing through a sequence of **runlevels** containing collections of scripts that start and stop services. Each runlevel supports a different mode of running the system. Within each runlevel, individual services can be set to run, or to be shut down if running.

However, all major recent distributions have moved away from this sequential runlevel method of system initialization, although they usually support the System V conventions for compatibility purposes. Next, we discuss the newer methods, **systemd** and **Upstart**.

ARUNDHATI BANDOPADHYAYA (arundhati.bandopadhyaya@gmail.com)

Startup Alternatives

SysVinit viewed things as a serial process, divided into a series of sequential stages. Each stage required completion before the next could proceed. Thus, startup did not easily take advantage of the ***parallel processing*** that could be done on multiple processors or cores.

Furthermore, shutdown and reboot was seen as a relatively rare event; exactly how long it took was not considered important. This is no longer true, especially with mobile devices and embedded Linux systems. Some modern methods, such as the use of **containers**, can require almost instantaneous startup times. Thus, systems now require methods with faster and enhanced capabilities. Finally, the older methods required rather complicated startup scripts, which were difficult to keep universal across distribution versions, kernel versions, architectures, and types of systems. The two main alternatives developed were:

Upstart

- Developed by Ubuntu and first included in 2006
- Adopted in Fedora 9 (in 2008) and in RHEL 6 and its clones.

systemd

- Adopted by Fedora first (in 2011)
- Adopted by RHEL 7 and SUSE
- Replaced Upstart in Ubuntu 16.04

While the migration to **systemd** was rather controversial, it has been adopted by the major distributions, and so we will not discuss the older System V method or Upstart, which has become a dead end. Regardless of how one feels about the controversies or the technical methods of **systemd**, almost universal adoption has made learning how to work on Linux systems simpler, as there are fewer differences among distributions. We enumerate **systemd** features next.

systemd features

Systems with **systemd** start up faster than those with earlier **init** methods. This is largely because it replaces a serialized set of steps with aggressive parallelization techniques, which permits multiple services to be initiated simultaneously.

Complicated startup shell scripts are replaced with simpler configuration files, which enumerate what has to be done before a service is started, how to execute service startup, and what

conditions the service should indicate have been accomplished when startup is finished. One thing to note is that **/sbin/init** now just points to **/lib/systemd/systemd**; i.e. **systemd** takes over the **init** process.

One **systemd** command (**systemctl**) is used for most basic tasks. While we have not yet talked about working at the command line, here is a brief listing of its use:

- Starting, stopping, restarting a service (using **nfs** as an example) on a currently running system:
  $ sudo systemctl start|stop|restart nfs.service
- Enabling or disabling a system service from starting up at system boot:
  $ sudo systemctl enable|disable nfs.service

Linux filesystems

Different types of filesystems supported by Linux:

- Conventional disk filesystems: **ext2**, **ext3**, **ext4**, **XFS**, **Btrfs**, **JFS**, **NTFS**, etc.
- Flash storage filesystems: **ubifs**, **JFFS2**, **YAFFS**, etc.
- Database filesystems
- Special purpose filesystems: **procfs, sysfs, tmpfs, squashfs, debugfs**, etc.

Partitions and Filesystems

A **partition** is a physically contiguous section of a disk, or what appears to be so in some advanced setups.

A **filesystem** is a method of storing/finding files on a hard disk (usually in a partition).

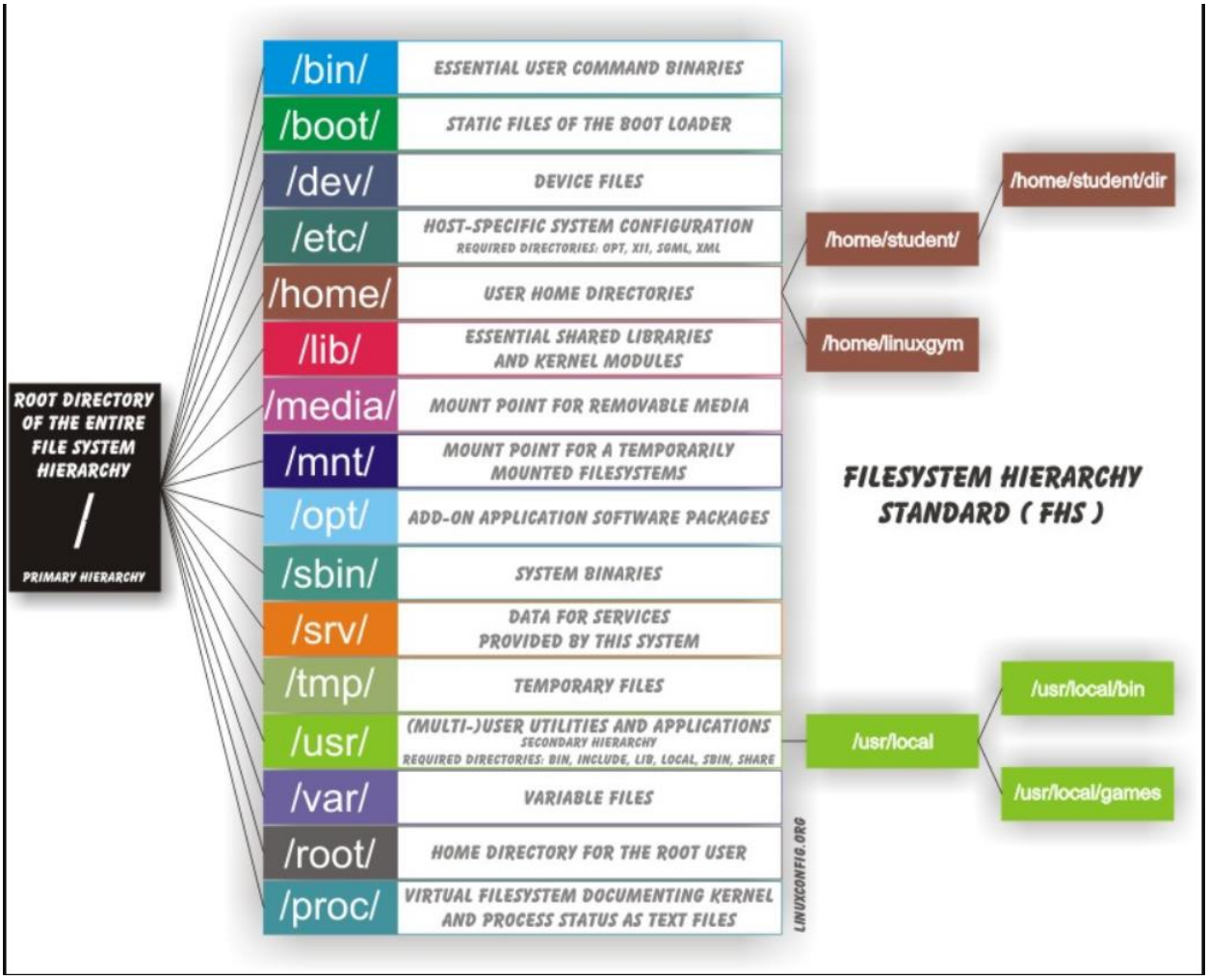
One can think of a partition as a container in which a filesystem resides, although in some circumstances, a filesystem can span more than one partition if one uses symbolic links.

	Windows	Linux
Partition	Disk1	`/dev/sda1`
Filesystem Type	NTFS/VFAT	EXT3/EXT4/XFS/BTRFS...
Mounting Parameters	DriveLetter	MountPoint
Base Folder (where OS is stored)	C:\	/

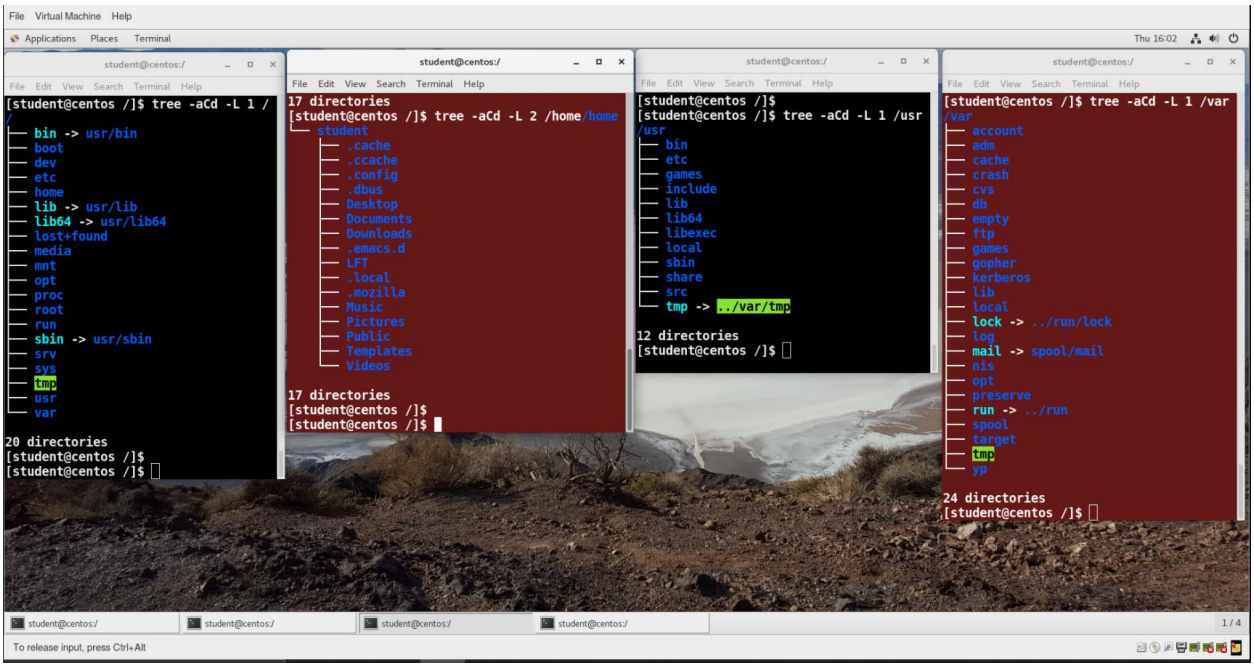
File system hierarchy standard

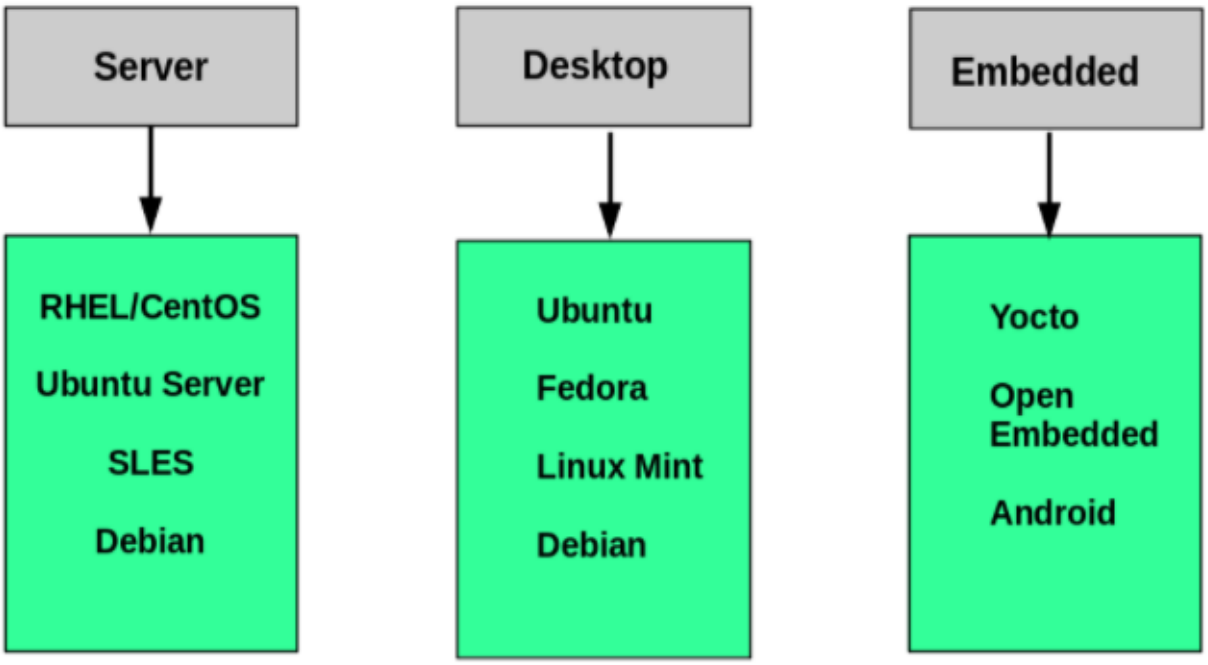
Linux systems store their important files according to a standard layout called the **F**ilesystem **H**ierarchy **S**tandard (**FHS**), which has long been maintained by the Linux Foundation. For more information, take a look at the following document: "*Filesystem Hierarchy Standard*" created by LSB Workgroup. Having a standard is designed to ensure that users, administrators, and developers can move between distributions without having to re-learn how the system is organized.

Linux uses the '**/**' character to separate paths (unlike Windows, which uses '****'), and does not have drive letters. Multiple drives and/or partitions are mounted as directories in the single filesystem. Removable media such as USB drives and CDs and DVDs will show up as mounted at **/run/media/yourusername/disklabel** for recent Linux systems, or under **/media** for older distributions. For example, if your username is **student** a USB pen drive labeled FEDORA might end up being found at **/run/media/student/FEDORA**, and a file **README.txt** on that disc would be at **/run/media/student/FEDORA/README.txt**.



All Linux filesystem names are case-sensitive, so **/boot**, **/Boot**, and **/BOOT** represent three different directories (or folders). Many distributions distinguish between core utilities needed for proper system operation and other programs, and place the latter in directories under **/usr** (think *user*). To get a sense for how the other programs are organized, find the **/usr** directory in the diagram from the previous page and compare the subdirectories with those that exist directly under the system root directory (*/*).





Some questions worth thinking about before deciding on a distribution include:

- What is the main function of the system (server or desktop)?
- What types of packages are important to the organization? For example, web server, word processing, etc.
- How much hard disk space is required and how much is available? For example, when installing Linux on an embedded device, space is usually constrained.

ARUNDHATI BANDOPADHYAYA (arundhati.bandopadhyaya@gmail.com)

- How often are packages updated?
- How long is the support cycle for each release? For example, **LTS** releases have long-term support.
- Do you need kernel customization from the vendor or a third party?
- What hardware are you running on? For example, it might be **X86**, **ARM**, **PPC**, etc.
- Do you need long-term stability? Can you accept (or need) a more volatile cutting edge system running the latest software?