16.1 Introduction

Booting is the process of bringing the system from an "off" status to a running operating system. Understanding this process is very important to fixing boot problems when they occur and modifying system states called *runlevels*.

The boot process takes places in four main stages, some of which are modified by administrators, while for the others it is sufficient just to be aware of the actions taking place:

1. Firmware Stage
2. Bootloader
3. Kernel Stage
4. Init Stage

16.2 Firmware Stage

The firmware stage is the first stage to take place after the computer is powered on. At this point the computer has power but needs to start executing some software that will eventually get a full kernel booted.

As previously mentioned, most firmware is referred to as the Basic Input Output System (BIOS) and is stored on the motherboard in non-volatile memory such as Read Only Memory (ROM) or flash memory. The BIOS provides a simplified view of the computer hardware so that the job of the next phase is much easier. A computer's hard drive may have gigabytes of space in which to store programs, but at this phase of the computer's life it is dealing with programs that must fit into kilobytes of space.

Even on systems that have replaced the traditional BIOS with the Unified Extensible Firmware Interface (UEFI), the system firmware is still often referred to as BIOS. For more detailed information of firmware outside of its role in the boot process, refer back to the previous chapter.

The BIOS has a number of jobs to perform as part of the first stage of the boot process. One of these jobs is the execution of a Power-On Self Test (POST) in order to ensure the hardware of the system is functioning properly. The POST runs some basic sanity checks on the CPU, memory, and peripherals so that obvious errors, such as missing memory chips, are found early in the boot cycle.

Modern servers have advanced peripherals that may have their own BIOS. The motherboard BIOS initializes peripheral BIOS as the system prepares to boot.

The final job of the BIOS is to find the proper boot drive from the available storage devices, and load the Master Boot Record (MBR) from that device. The Master Boot Record is the first sector (or 512 bytes) of the disk. It contains a partition table and a very small amount of executable code called the *first stage bootloader* whose purpose is to load the more feature-rich *second stage bootloader*.

16.3 Bootloader Stage

The bootloader will perform several operations, but the primary task is to load the Linux kernel into memory and execute it. Once that has occurred, the kernel takes over booting the system.

The most common bootloaders used on machines are the Linux Loader (LILO) and the Grand Unified Bootloader (GRUB). Both LILO and GRUB are bootloaders that are able to boot Linux from a system that is using a BIOS ROM. The latest version of GRUB supports booting Linux from a system using UEFI, while the Efi Linux Loader (ELILO) can be used in place of LILO on systems that use UEFI instead of traditional BIOS. UEFI systems give the firmware stage a lot more memory and capabilities so that it can handle much larger and more complicated hardware.

Outside of the IBM PC compatible architectures, there are additional bootloaders that are used. For Linux systems to boot on Sparc hardware, there is Sparc Improved bootLOader (SILO) and for PowerPC hardware there is Yet Another BOOTloader (YABOOT).

It is also possible to boot off the network through the Preboot Execution Environment (PXE). In the PXE system a compatible motherboard and network card contain enough intelligence to acquire an address from the network and use the Trivial File Transfer Protocol (TFTP) to download a special bootloader from a server. This is most often used to get many identical machines running quickly. It is powerful but complicated, and outside the scope of LPIC1.

As the bootloader is just some software that gets a kernel to run, it is possible to boot multiple operating systems at different times off of one computer in a process known as dual booting. The kernel that the bootloader is trying to run could be a Linux kernel, it could be a Microsoft Windows image, or it could be a bootable CD.

The bootloader can also pass parameters to the kernel, such as to boot into a maintenance mode or to enable or disable certain hardware. This is done by manipulating the bootloader configuration. GRUB provides a reasonably powerful command line interface that lets an administrator make changes to the kernel before it boots without requiring that the configuration be written to disk.

The bootloader then loads the kernel from disk into memory and transfers control over. The system is now running Linux, and may finish booting.

16.4 Kernel Stage

Now that the bootloader has loaded the kernel into memory, there is much work to be done before programs can be loaded. The kernel must initialize any hardware drivers and get the root filesystem mounted for the next stage. These two tasks are actually quite complicated because the facilities provided by the BIOS to access the hardware are quite limited. The kernel must boot the system in several phases.

The kernel itself is laid out much like a regular executable except that it it must be self contained -- shared libraries are not available at this point in the boot process so the kernel itself is statically linked. This kernel typically lives in the `/boot` partition which, on most hardware, is in a separate partition that's kept at the beginning of the hard drive. This location is important for some BIOS and bootloader combinations that can only access the first 1024 cylinders of the disk.

As the size of the kernel increased over time developers found it better to compress the kernel to make it fit within the limitations of the BIOS. Therefore the executable comprising the kernel will decompress itself as it is loaded, leading to the name of `zImage` for the kernel (the letter `z` being associated with the Unix compression library called `zlib`.)

The kernel grew even more and it became a problem to load the whole kernel into a consecutive block of memory. The big `zImage`, `bzimage`, kernel format came into being that allowed the kernel to be loaded into multiple memory blocks.

The Linux kernel must mount the root file system `/` in order to get to the next step and to make the system useful. However it is possible that the root filesystem exists on a device that the kernel does not know how to support. The solution to this is the initial RAM disk `initrd`. The kernel drivers necessary to continue the boot are bundled into a filesystem that is stored beside the kernel itself in `/boot`. The kernel boots, mounts the `initrd`, loads the drivers inside, and then remounts the real root filesystem using the new drivers. This allows drivers to be added to the kernel easily and for the kernel to mount the root filesystem over almost any storage hardware even if it's over a network.

As the kernel is booting it is able to initialize hardware and make detected devices available to the rest of the operating system.

The kernel's final job is to start the first process on the system. As it is the first process, it will normally have a process id (PID) of 1; the name of this process is `init`.

```
sysadmin@localhost:~$ ps -ejH
  PID  PGID   SID TTY          TIME CMD
    1     1     1 ?        00:00:00 init
   32    32    32 ?        00:00:00   rsyslogd
   37    37    37 ?        00:00:00   cron
   39    39    39 ?        00:00:00   sshd
   56    56    56 ?        00:00:00   named
   69      1      1 ?        00:00:00    login
   79     79      1 ?        00:00:00      bash
  676    676      1 ?        00:00:00        ps
```

The first process is responsible for much of the operation of the system from here on in. Usually this process will be `/sbin/init` though there are other options like `systemd`. Alternatively on embedded hardware the `init` process could be a shell or a specialized daemon. In any case, this first process will start the daemons that the rest of the system will use. This process will be the parent process for any process that otherwise is missing a parent. This process persists for the life of the system.

16.5 init Stage

The `init` stage finishes booting the system; the first process of the operating system is started and this process is responsible for starting all other system processes.

The first process to be booted has two important responsibilities: The first is to continue the booting process to get services running, login screens displaying, and consoles listening. The second is some basic process management. Any process that loses a parent gets adopted by `init`.

Until recently, this process followed a design that was established with the release of System V of Unix, which is sometimes referred to as SysVinit. The actual process that is executed is the init process. Recently, other programs have emerged to compete with and replace the traditional `init` process: Upstart and Systemd.

If the system uses the traditional `init` program, then the `/etc/inittab` file is used to determine what scripts will be executed to start the services that will be available on the system. The `inittab` file points to other scripts that do the work, usually stored in `/etc/init.d`. There will be more discussion of these scripts later, but in general each service the system will run has a script that can start, stop, and restart a service, and `init` will cause each of these to be run in turn as needed to get the system to the desired state.

If the traditional `init` has been replaced with Upstart, the scripts in the `/etc/init` directory are used to complete system initialization.

If the traditional `init` has been replaced with Systemd, then the files in the `/etc/systemd` directory are used for starting up and running the system.

Even if your system is using Systemd or Upstart as a replacement for the traditional `init` process, both replacements use an executable named `init` with the pathname `/sbin/init`. This is to maintain compatibility with many legacy processes. So, while some of the behavior of Systemd and Upstart will be different, they have some features that are similar to the traditional `init` process.

While your system will only use one of these booting technologies, it is important to understand all three. For example, while your current system uses the SysV init technique, you might find yourself working on a different distribution in the future that makes use of Upstart. In any case, all three processes are testable topics on the LPIC exams.

16.6 dmesg Command

The `dmesg` command can be executed after booting the system to see the messages generated by the kernel during boot time. This is useful when the system doesn't appear to boot correctly; the messages displayed can help an administrator troubleshoot the boot process.

Kernel messages are stored in a ring buffer of limited size, therefore the messages that are generated at boot time may be overwritten later as the buffer fills up. It is possible that some of the kernel messages generated at boot time may be stored in the `/var/log/dmesg` file. Each time the system boots, the `/var/log/dmesg` file is overwritten with the messages that were generated during the boot process.

It is common to execute the `dmesg` command upon connecting a new device to the system. This allows the administrator to see how the kernel dealt with the new device and usually to see what pathname the new device has been assigned.

A less common use of the `dmesg` command is to alter the level of messages that the kernel will print in the system console. Typing in a terminal and having kernel messages printing at the same time can become annoying, the command `dmesg -n 1` can be used to disable the printing of all but the most critical of kernel messages.

16.7 /var/log/messages File

Kernel messages and other system-related messages are typically stored in the `/var/log/messages` file. This file, which is considered the main log file, is alternatively named `/var/log/syslog` in some distributions.

Traditionally, the primary log file is updated with new log entries by the combination of the `syslogd` and `klogd` daemons. Replacements for these daemons include `rsyslogd` and `syslog-ng daemons`.

The main system log file can be helpful for analyzing why some services may not start correctly. It can also be helpful to determine why some devices may not be functioning; the kernel will place log entries in this log file if it can detect a device and as it loads the appropriate kernel modules or drivers to support new devices.

Although the `/var/log/messages` file is considered the main log file, there are other log files in the `/var/log` folder, which may need to be examined when trying to troubleshoot system service issues. For instance, the apache web server daemon `httpd`, manages its own log files in another location, like the `/var/log/httpd/error_log file`. Traditionally, all log files are stored in the `/var/log` directory.

Chapter 16: The Boot Process

This chapter will cover the following exam objectives:

101.2: Boot the System

Weight: 3

Description: Candidates should be able to guide the system through the booting process

Key Knowledge Areas:

- Demonstrate knowledge of the boot sequence from BIOS to boot completion
  Section 16.2 | Section 16.3 | Section 16.4 | Section 16.5
- Understanding of SysVinit and systemd
  Section 16.5
- Awareness of Upstart
  Section 16.5
- Check boot events in log files
  Section 16.7

Chapter 16: The Boot Process

MBR

The Master Boot Record (MBR) is the first 512 bytes of a storage device. It contains an operating system bootloader and the storage device's partition table.
Section 16.2

SysVinit

The traditional service management package for Linux, containing the init program (the first process that is run when the kernel has finished initializing) as well as some infrastructure to start and stop services and configure them.
Section 16.5

bootloader

The first piece of software started by the BIOS or UEFI. It is responsible for loading the kernel with the wanted kernel parameters, and initial RAM disk before initiating the boot process.
Section 16.3

dmesg

Displays the contents of the system message buffer
Section 16.6

init

The parent of all processes on the system, it is executed by the kernel and is responsible for starting all other processes.
Section 16.5

initramfs

Iinitial RAM file system, used by Linux systems to prepare the system during boot before the operating systems' init process starts.
Section 16.5

kernel

The central module of an operating system. It is the part of the operating system that loads first, and it remains in main memory.
| Section 16.4

systemd

A full replacement for init with parallel starting of services and other features, used by many distributions.
Section 16.5