LINUX

This sheet includes in-depth content about LINUX

Pre-requisite:

- Basic knowledge of Amazon Web Services
- Basic Shell Commands

Table of Content:

INTRODUCTION

Components of Linux

LINUX system comprises of 4 main parts:

1. The Linux kernel
2. The GNU utilities
3. A graphical desktop environment
4. Application software

Linux Kernel:

Kernel is the core of the Linux system, which controls all the hardware and software on the computer system. It allocates hardware when necessary and executes software when required.

To check the kernel version:

```
[me@linuxbox ~]$ uname -r
```

The `uname` command includes additional options that you can use to get more information about your kernel. Simply add an option after the command:

```
[me@linuxbox ~]$ uname --help
```

For displaying different parameters:

Parameter	Description
a	Display all information
o	Display the operating system (usually GNU/Linux)
r	Display kernel release
v	Display kernel version

Linux Distribution/Flavours

A complete Linux system package is called a distribution. Many different Linux distributions are available to meet just about any computing requirement you could have.

The different Linux distributions are often divided into three categories:

- Full core Linux distributions
- Specialized distributions
- LiveCD test distributions

Core Linux Distributions

A core Linux distribution contains a kernel, one or more graphical desktop environments, and just about every Linux application that is available, precompiled for the kernel. It provides one-stop shopping for a complete Linux installation. Table shows some of the more popular core Linux distributions.

Distribution	Description
Slackware	One of the original Linux distribution sets, popular with Linux geeks

Distribution	Description
Red Hat	A commercial business distribution used mainly for Internet servers
Fedora	A spin-off from Red Hat but designed for home use
Gentoo	A distribution designed for advanced Linux users, containing only Linux source code
openSUSE	Different distributions for business and home use
Debian	Popular with Linux experts and commercial Linux products

Specialized Linux Distributions

A new subgroup of Linux distributions has started to appear. These are typically based on one of the main distributions but contain only a subset of applications that would make sense for a specific area of use. Table shows some of the specialized Linux distributions available and what they specialize in.

Distribution	Description
CentOS	A free distribution built from the Red Hat Enterprise Linux source code
Ubuntu	A free distribution for school and home use
PCLinuxOS	A free distribution for home and office use
Mint	A free distribution for home entertainment use
dyne:bolic	A free distribution designed for audio and MIDI applications
Puppy Linux	A free small distribution that runs well on older PCs

The Linux LiveCD

A relatively new phenomenon in the Linux world is the bootable Linux CD distribution. This lets you see what a Linux system is like without actually installing it. Most modern PCs can boot from a CD instead of the standard hard drive. To take advantage of this, some Linux distributions create a bootable CD that contains a sample Linux system (called a Linux LiveCD). Because of the limitations of the single CD size, the sample can't contain a complete Linux system, but you'd be surprised at all the software they can cram in there. The result is that you can boot your PC from the CD and run a Linux distribution without having to install anything on your hard drive! Table shows some popular Linux LiveCDs that are available.

Distribution	Description
Knoppix	A German Linux, the first Linux LiveCD developed
PCLinuxOS	Full-blown Linux distribution on a LiveCD
Ubuntu	A worldwide Linux project, designed for many languages
Slax	A live Linux CD based on Slackware Linux
Puppy Linux	A full-featured Linux designed for older PCs

• Levels and Layers of Abstraction in a Linux System

Using abstraction to split computing systems into components makes things easier to understand, but it doesn't work without organization. We arrange components into layers or levels. A layer or level is a classification (or grouping) of a component according to where that component sits between the user and the hardware. Web browsers, games, and such sit at the top layer; at the bottom layer we have the memory in the computer hardware—the 0s and 1s. The operating system occupies most of the layers in between.

A Linux system has three main levels:

The `hardware` is at the base. Hardware includes the memory as well as one or more central processing units (CPUs) to perform computation and to read from and write to memory. Devices such as disks and network interfaces are also part of the

hardware. The next level up is the `kernel`, which is the core of the operating system. The kernel is software residing in memory that tells the CPU what to do. The kernel manages the hardware and acts primarily as an interface between the hardware and any running program.
`Processes`—the running programs that the kernel manages—collectively make up the system's upper level, called user space. (A more specific term for process is user process, regardless of whether a user directly interacts with the process.For example, all web servers run as user processes.)

User Processes	`Graphical User Interface Servers Shell`
Linux Kernel	`System Calls Process Management Memory Management Device Drivers`
Hardware	`Processor (CPU) Main Memory (RAM) Disks Network Ports`

. The kernel is in charge of managing tasks in four general system areas:

- ## Processes

  The kernel is responsible for determining which processes are allowed to use the CPU.

- ## Memory

  The kernel needs to keep track of all memory—what is currently allocated to a particular process, what might be shared between processes, and what is free.

- ## Device drivers

  The kernel acts as an interface between hardware (such as a disk) and processes. It's usually the kernel's job to operate the hardware.

- ## System calls and support

  Processes normally use system calls to communicate with the kernel.

Two system calls, `fork()` and `exec()`, are important to understanding how processes start up:
`fork()` : When a process calls fork(), the kernel creates a nearly identical copy of the process.
`exec()` : When a process calls exec(program), the kernel starts program, replacing the current process.

Basic Commands and Directory Hierarchy

• Bourne Shell/Shell

The shell is one of the most important parts of a Unix system. A shell is a program that runs commands, like the ones that users enter. The shell also serves as a small programming environment. Unix programmers often break common tasks into little components and use the shell to manage tasks and piece things together.

There are many different Unix shells, but all derive several of their features from the Bourne shell (/bin/sh), a standard shell developed at Bell Labs for early versions of Unix. Every Unix system needs the Bourne shell in order to function correctly, as you will see throughout this book. Linux uses an enhanced version of the Bourne shell called bash or the "Bourne-again" shell. The bash shell is the default shell on most Linux distributions, and /bin/sh is normally a link to bash on a Linux system. You should use the bash shell when running the examples in this book.

• Basic Commands

Now let's look at some more Unix commands. Most of the following programs take multiple arguments, and some have so many options and formats that an unabridged listing would be pointless. This is a simplified list of the basic commands; you don't need all of the details just yet.

- **`ls`**

The ls command lists the contents of a directory. The default is the current directory. Use `ls -l` for a detailed (long) listing and `ls -F` to display file type information. (For more on the file types and permissions displayed in the left column below, see Section 2.17.) Here is a sample long listing; it includes the owner of the file (column 3), the group (column 4), the file size (column 5), and the modification date/time (between column 5 and the filename):

```
$ ls -l
  total 3616
  -rw-r--r-- 1 juser users 3804 Apr 30 2011 abusive.c
```

```
-rw-r--r-- 1 juser users 4165 May 26 2010 battery.zip
Basic Commands and Directory Hierarchy 15
-rw-r--r-- 1 juser users 131219 Oct 26 2012 beav_1.40-13.tar.gz
-rw-r--r-- 1 juser users 6255 May 30 2010 country.c
drwxr-xr-x 2 juser users 4096 Jul 17 20:00 cs335
-rwxr-xr-x 1 juser users 7108 Feb 2 2011 dhry
-rw-r--r-- 1 juser users 11309 Oct 20 2010 dhry.c
-rw-r--r-- 1 juser users 56 Oct 6 2012 doit
drwxr-xr-x 6 juser users 4096 Feb 20 13:51 dw
drwxr-xr-x 3 juser users 4096 May 2 2011 hough-stuff
```

- ## cp

In its simplest form, cp copies files. For example, to copy file1 to file2, enter this:

```
$ cp file1 file2
```

To copy a number of files to a directory (folder) named dir, try this instead:

```
$ cp file1 ... fileN dir
```

- ## mv

The mv (move) command is like cp. In its simplest form, it renames a file. For example, to rename file1 to file2, enter this:

```
$ mv file1 file2
```

You can also use mv to move a number of files to a different directory:

```
$ mv file1 ... fileN dir
```

- ## touch

The touch command creates a file. If the file already exists, touch does not change it, but it does update the file's modification time stamp printed with the ls -l command. For example, to create an empty file, enter this:

```
$ touch file
```

Then run ls -l on that file. You should see output like the following, where the date and time u indicate when you ran touch:

```
$ ls -l file
-rw-r--r-- 1 juser users 0 May 21 18:32u file
```

- ## rm

To delete (remove) a file, use rm. After you remove a file, it's gone from your system and generally cannot be undeleted.

```
$ rm file
```

. echo

The echo command prints its arguments to the standard output:

```
$ echo Hello again.
Hello again.
```

The echo command is very useful for finding expansions of shell globs (“wildcards” such as *) and variables (such as $HOME), which you will encounter later

. File Modes and Permissions

Every Unix file has a set of permissions that determine whether you can read, write, or run the file. Running ls -l displays the permissions. Here’s an example of such a display:

```
-rw-r--r--u 1 juser somegroup 7041 Mar 26 19:34 endnotes.html
```

The file’s mode u represents the file’s permissions and some extra information. There are four parts to the mode, as illustrated in Figure 2-1. The first character of the mode is the file type. A dash (-) in this position, as in the example, denotes a regular file, meaning that there’s nothing special about the file. This is by far the most common kind of file. Directories are also common and are indicated by a d in the file type slot.

The rest of a file’s mode contains the permissions, which break down into three sets: user, group, and other, in that order. For example, the `rw-` characters in the example are the user permissions, the `r--` characters that follow are the group permissions, and the final `r--` characters are the other permissions. Each permission set can contain four basic representations: `r` Means that the file is readable. `w` Means that the file is writable. `x` Means that the file is executable (you can run it as a program).

- Means nothing. The user permissions (the first set) pertain to the user who owns the file. In the preceding example, that’s juser. The second set, group permissions, are for the file’s group (somegroup in the example). Any user in that group can take advantage of these permissions.

Everyone else on the system has access according to the third set, the other permissions, which are sometimes called world permissions.

Mode	Meaning	Used For
644 user:	read/write; group, other:	read files
600 user:	read/write; group, other:	none files
755 user:	read/write/execute; group, other:	read/execute directories, programs
700 user:	read/write/execute; group, other:	none directories, programs
711 user:	read/write/execute; group, other:	execute directories

- **Symbolic Links**
- ```
  lrwxrwxrwx 1 ruser users 11 Feb 27 13:52 somedir -> /home/origdir
  ```

Creating Symbolic Links

To create a symbolic link from target to linkname, use ln -s:

```
$ ln -s target linkname
```

Navigating Directories

Unix has a directory hierarchy that starts at /, sometimes called the root directory. The directory separator is the slash (/), not the backslash (). There are several standard subdirectories in the root directory, such as /usr.

When you refer to a file or directory, you specify a path or pathname. When a path starts with / (such as /usr/lib), it's a full or absolute path. A path component identified by two dots (..) specifies the parent of a directory. For example, if you're working in /usr/lib, the path .. would refer to /usr. Similarly, ../bin would refer to /usr/bin.

One dot (.) refers to the current directory; for example, if you're in /usr/lib, the path . is still /usr/lib, and ./X11 is /usr/lib/X11. You won't have to use . very often because
```

most commands default to the current directory if a path doesn't start with / (you could just use X11 instead of ./X11 in the preceding example).

A path not beginning with / is called a relative path. Most of the time, you'll work with relative pathnames, because you'll already be in the directory you need to be in or somewhere close by.

Now that you have a sense of the basic directory mechanics, here are some essential directory commands.

- cd

The current working directory is the directory that a process (such as the shell) is currently in. The cd command changes the shell's current working directory:

```
$ cd dir
```

If you omit dir, the shell returns to your home directory, the directory you started in when you first logged in.

- mkdir

The mkdir command creates a new directory dir:

```
$ mkdir dir
```

- rmdir

The rmdir command removes the directory dir:

```
$ rmdir dir
```

If dir isn't empty, this command fails. However, if you're impatient, you probably don't want to laboriously delete all the files and subdirectories inside dir first. You can use rm -rf dir to delete a directory and its contents, but be careful! This is one of the few commands that can do serious damage, especially if you run it as the superuser. The -r option specifies recursive delete to repeatedly delete everything inside dir, and -f forces the delete operation. Don't use the -rf flags with globs such as a star (*). And above all, always double-check your command before you run it.

Archiving and Compressing Files

Now that you've learned about files, permissions, and possible errors, you need to master gzip and tar.

- gzip The program gzip (GNU Zip) is one of the current standard Unix compression programs. A file that ends with .gz is a GNU Zip archive. Use

gunzip file.gz to uncompress .gz and remove the suffix; to compress it again, use gzip file.

- tar

Unlike the zip programs for other operating systems, gzip does not create archives of files; that is, it doesn't pack multiple files and directories into one file. To create an archive, use tar instead:

```
$ tar cvf archive.tar file1 file2 ...
```

- Unpacking tar files

To unpack a .tar file with tar use the x flag:

```
$ tar xvf archive.tar
```

Compressed Archives (.tar.gz)

Many beginners find it confusing that archives are normally found compressed, with filenames ending in .tar.gz. To unpack a compressed archive, work from the right side to the left; get rid of the .gz first and then worry about the .tar. For example, these two commands decompress and unpack .tar.gz:

```
$ gunzip file.tar.gz
$ tar xvf file.tar
```

- zcat

The method shown above isn't the fastest or most efficient way to invoke tar on a compressed archive, and it wastes disk space and kernel I/O time. A better way is to combine archival and compression functions with a pipeline. For example, this command pipeline unpacks .tar.gz:

```
$ zcat file.tar.gz | tar xvf -
$ tar ztvf file.tar.gz
```

Dot Files

Change to your home directory, take a look around with ls, and then run ls -a. Do you see the difference in the output? When you run ls without the -a, you won't see the configuration files called dot files. These are files and directories whose names begin with a dot (.). Common dot files are .bashrc and .login, and there are dot directories, too, such as .ssh. There is nothing special about dot files or directories. Some programs don't show them by default so that you won't see a complete mess when listing the contents of your home directory. For example, ls doesn't list dot files

unless you use the -a option. In addition, shell globs don't match dot files unless you explicitly use a pattern such as .*.

NOTE

```
     You can run into problems with globs because .* matches . and .. (the current
and
     parent directories). You may wish to use a pattern such as .[^.]* or .??* to
get all
     dot files except the current and parent directories.
```

• Device Name Summary

It can sometimes be difficult to find the name of a device (for example, when partitioning a disk). Here are a few ways to find out what it is:

- Query udevd using udevadm
- Look for the device in the /sys directory.
- Guess the name from the output of the dmesg command (which prints the last few kernel messages) or the kernel system log file This output might contain a description of the devices on your system.
- For a disk device that is already visible to the system, you can check the output of the mount command.
- Run cat /proc/devices to see the block and character devices for which your system currently has drivers. Each line consists of a number and name. The number is the major number of the device as described in If you can guess the device from the name, look in /dev for the character or block devices with the corresponding major number, and you've found the device files.

Among these methods, only the first is reliable, but it does require udev. If you get into a situation where udev is not available, try the other methods but keep in mind that the kernel might not have a device file for your hardware.

• Filesystems

The last link between the kernel and user space for disks is typically the filesystem; this is what you're accustomed to interacting with when you run commands such as ls and cd. As previously mentioned, the filesystem is a form of database; it supplies the structure to transform a simple block device into the sophisticated hierarchy of files and subdirectories that users can understand.

At one time, filesystems resided on disks and other physical media used exclusively for data storage. However, the tree-like directory structure and I/O interface of

filesystems are quite versatile, so filesystems now perform a variety of tasks, such as the system interfaces that you see in /sys and /proc.

Creating a Filesystem

Once you're done with the partitioning process described in Section 4.1, you're ready to create filesystems. As with partitioning, you'll do this in user space because a user-space process can directly access and manipulate a block device. The mkfs utility can create many kinds of filesystems. For example, you can create an ext4 partition on /dev/sdf2 with this command:

```
# mkfs -t ext4 /dev/sdf2
```

And there's even more indirection. Inspect the mkfs.* files behind the commands and you'll see the following:

```
$ ls -l /sbin/mkfs.*
-rwxr-xr-x 1 root root 17896 Mar 29 21:49 /sbin/mkfs.bfs
-rwxr-xr-x 1 root root 30280 Mar 29 21:49 /sbin/mkfs.cramfs
lrwxrwxrwx 1 root root 6 Mar 30 13:25 /sbin/mkfs.ext2 -> mke2fs
lrwxrwxrwx 1 root root 6 Mar 30 13:25 /sbin/mkfs.ext3 -> mke2fs
lrwxrwxrwx 1 root root 6 Mar 30 13:25 /sbin/mkfs.ext4 -> mke2fs
lrwxrwxrwx 1 root root 6 Mar 30 13:25 /sbin/mkfs.ext4dev -> mke2fs
-rwxr-xr-x 1 root root 26200 Mar 29 21:49 /sbin/mkfs.minix
lrwxrwxrwx 1 root root 7 Dec 19 2011 /sbin/mkfs.msdos -> mkdosfs
lrwxrwxrwx 1 root root 6 Mar 5 2012 /sbin/mkfs.ntfs -> mkntfs
lrwxrwxrwx 1 root root 7 Dec 19 2011 /sbin/mkfs.vfat -> mkdosfs
```

To mount a filesystem, use the mount command as follows with the filesystem type, device, and desired mount point:

```
# mount -t type device mountpoint
```

For example, to mount the Fourth Extended filesystem /dev/sdf2 on /home/extra, use this command:

```
# mount -t ext4 /dev/sdf2 /home/extra
```

To unmount

(detach) a filesystem, use the umount command:

```
# umount mountpoint
```

To view a list of devices and the corresponding filesystems and UUIDs on your system, use the blkid (block ID) program:

```
# blkid
/dev/sdf2: UUID="a9011c2b-1c03-4288-b3fe-8ba961ab0898" TYPE="ext4"
/dev/sda1: UUID="70ccd6e7-6ae6-44f6-812c-51aab8036d29" TYPE="ext4"
/dev/sda5: UUID="592dcfd1-58da-4769-9ea8-5f412a896980" TYPE="swap"
/dev/sde1: SEC_TYPE="msdos" UUID="3762-6138" TYPE="vfat"
```

• Partitioning Disk Devices

There are many kinds of partition tables. The traditional table is the one found inside the Master Boot Record (MBR). A newer standard starting to gain traction is the Globally Unique Identifier Partition Table (GPT).

Here is an overview of the many Linux partitioning tools available:

parted

A text-based tool that supports both MBR and GPT.

gparted

A graphical version of parted.

fdisk

The traditional text-based Linux disk partitioning tool. fdisk does not support GPT.

gdisk

A version of fdisk that supports GPT but not MBR.

Because it supports both MBR and GPT, we'll use parted in this book. However, many people prefer the fdisk interface, and there's nothing wrong with that.

• Viewing a Partition Table

You can view your system's partition table with parted -l. Here is sample output from two disk devices with two different kinds of partition tables: # parted -l Model: ATA WDC WD3200AAJS-2 (scsi) Disk /dev/sda: 320GB Sector size (logical/physical): 512B/512B Partition Table: msdos Number Start End Size Type File system Flags 1 1049kB 316GB 316GB primary ext4 boot 2 316GB 320GB 4235MB extended 5 316GB 320GB 4235MB logical linux-swap(v1) Model: FLASH Drive UT_USB20 (scsi) Disk /dev/sdf: 4041MB Sector size (logical/physical): 512B/512B Partition Table: gpt Number Start End Size File system Name Flags 1 17.4kB 1000MB 1000MB myfirst 2 1000MB 4040MB 3040MB mysecond

• Swap Space

Not every partition on a disk contains a filesystem. It's also possible to augment the RAM on a machine with disk space. If you run out of real memory, the Linux virtual memory system can automatically move pieces of memory to and from a disk

storage. This is called swapping because pieces of idle programs are swapped to the disk in exchange for active pieces residing on the disk. The disk area used to store memory pages is called swap space (or just swap for short).

The free command's output includes the current swap usage in kilobytes as follows:

```
$ free
               total    used     free
--snip--
Swap:          514072  189804    324268
```

Using a Disk Partition as Swap Space

To use an entire disk partition as swap, follow these steps:

- 1.Make sure the partition is empty.
- 2.Run mkswap dev, where dev is the partition's device. This command puts a swap signature on the partition.
- 3.Execute swapon dev to register the space with the kernel.

Here is a sample entry that uses /dev/sda5 as a swap partition:

```
/dev/sda5 none swap sw 0 0
```

Using a File as Swap Space

You can use a regular file as swap space if you're in a situation where you would be forced to repartition a disk in order to create a swap partition. You shouldn't notice any problems when doing this. Use these commands to create an empty file, initialize it as swap, and add it to the swap pool:

```
# dd if=/dev/zero of=swap_file bs=1024k count=num_mb
# mkswap swap_file
# swapon swap_file
```

How the Linux Kernal Boot and User Space Start

Kernal Parameters

When running the Linux kernel, the boot loader passes in a set of text-based kernel parameters that tell the kernel how it should start. The parameters specify many different types of behavior, such as the amount of diagnostic output the kernel

should produce and device driver–specific options. You can view the kernel parameters from your system's boot by looking at the /proc/cmdline file:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-3.2.0-67-generic-pae root=UUID=70ccd6e7-6ae6-44f6-
812c-51aab8036d29 ro quiet splash vt.handoff=7
```

The parameters are either simple one-word flags, such as ro and quiet, or key=value pairs, such as vt.handoff=7. Many of the parameters are unimportant, such as the splash flag for displaying a splash screen, but one that is critical is the root parameter. This is the location of the root filesystem; without it, the kernel cannot find init and therefore cannot perform the user space start.

The root filesystem can be specified as a device file, such as in this example:

```
root=/dev/sda1
```

However, on most modern desktop systems, a UUID is more common:

```
root=UUID=70ccd6e7-6ae6-44f6-812c-51aab8036d29
```

The `ro` parameter is normal; it instructs the kernel to mount the root filesystem in read-only mode upon user space start. (Read-only mode ensures that `fsck` can check the root filesystem safely; after the check, the bootup process remounts the root filesystem in read-write mode.) Upon encountering a parameter that it does not understand, the Linux kernel saves the parameter.
The kernel later passes the parameter to init when performing the user space start.

For example, if you add `-s` to the kernel parameters, the kernel passes the `-s` to the init program to indicate that it should start in single-user mode.
Now let's look at the mechanics of how boot loaders start the kernel.

Boot Loaders

At the start of the boot process, before the kernel and init start, a boot loader starts the kernel. The task of a boot loader sounds simple: It loads the kernel into memory, and then starts the kernel with a set of kernel parameters. But consider the questions that the boot loader must answer:

- Where is the kernel?
- What kernel parameters should be passed to the kernel when it starts? The answers are (typically) that the kernel and its parameters are usually somewhere on the root filesystem. It sounds like the kernel parameters should be easy to find, except that the kernel is not yet running, so it can't traverse a filesystem to find the necessary files. Worse, the kernel device drivers normally used to access the disk are also unavailable. Think of this as a kind of "chicken or egg" problem.

The filesystem question is trickier. Most modern boot loaders can read partition tables and have built-in support for read-only access to filesystems. Thus, they can find and read files. This capability makes it far easier to dynamically configure and enhance the boot loader. Linux boot loaders have not always had this capability; without it, configuring the boot loader was more difficult.

Boot Loader Tasks

A Linux boot loader's core functionality includes the ability to do the following:

- Select among multiple kernels.
- Switch between sets of kernel parameters.
- Allow the user to manually override and edit kernel image names and parameters (for example, to enter single-user mode).
- Provide support for booting other operating systems.

Boot loaders have become considerably more advanced since the inception of the Linux kernel, with features such as history and menu systems, but the basic need has always been flexibility in kernel image and parameter selection. One interesting phenomenon is that certain needs have diminished.

For example, because you can now perform an emergency or recovery boot partially or entirely from a USB storage device, you probably won't have to worry about manually entering kernel parameters or going into single-user mode. But modern boot loaders offer more power than ever, which can be particularly handy if you're building custom kernels or just want to tweak parameters.

Here are the main boot loaders that you may encounter, in order of popularity:

GRUB

A near-universal standard on Linux systems

LILO

One of the first Linux boot loaders. ELILO is a UEFI version

SYSLINUX

Can be configured to run from many different kinds of filesystems

LOADLIN

Boots a kernel from MS-DOS

efilinux

A UEFI boot loader intended to serve as a model and reference for other UEFI boot loaders

coreboot (formerly LinuxBIOS)

A high-performance replacement for the PC BIOS that can include a kernel

Linux Kernel EFISTUB

A kernel plugin for loading the kernel directly from the EFI/UEFI System Partition (ESP) found on recent systems

GRUB Introduction

GRUB stands for Grand Unified Boot Loader. We'll cover GRUB 2; there is also an older version now called GRUB Legacy that is slowing falling out of use. One of GRUB's most important capabilities is filesystem navigation that allows for much easier kernel image and configuration selection. One of the best ways to see this in action and to learn about GRUB in general is to look at its menu. The interface is easy to navigate, but there's a good chance that you've never seen it. Linux distributions often do their best to hide the boot loader from you.

To access the GRUB menu, press and hold shift when your BIOS or firmware startup screen first appears. Otherwise, the boot loader configuration may not pause before loading the kernel. Figure 5-1 shows the GRUB menu. Press esc to temporarily disable the automatic boot timeout after the GRUB menu appears.

Try the following to explore the boot loader:

1. Reboot or power on your Linux system.
2. Hold down shift during the BIOS/Firmware self-test and/or splash screen to get the GRUB menu.
3. Press e to view the boot loader configuration commands for the default boot option.

Exploring Devices and Partitions with the GRUB Command Line

- **Listing Devices**

To get a feel for how GRUB refers to the devices on your system, access the GRUB command line by pressing C at the boot menu or configuration editor.

You should get the GRUB prompt:

```
grub>
```

You can enter any command here that you see in a configuration, but to get started, try a diagnostic command instead: ls. With no arguments, the output is a list of devices known to GRUB:

```
grub> ls
(hd0) (hd0,msdos1) (hd0,msdos5)
```

To get more detailed information, use ls -l. This command can be particularly useful because it displays any UUIDs of the partitions on the disk. For example:

```
grub> ls -l
Device hd0: Not a known filesystem - Total size 426743808 sectors
Partition hd0,msdos1: Filesystem type ext2 – Last modification time
2015-09-18 20:45:00 Friday, UUID 4898e145-b064-45bd-b7b4-7326b00273b7 -
Partition start at 2048 - Total size 424644608 sectors
Partition hd0,msdos5: Not a known filesystem - Partition start at
424648704 - Total size 2093056 sectors
```

File Navigation

Now let's look at GRUB's filesystem navigation capabilities. Determine the GRUB root with the echo command (recall that this is where GRUB expects to find the kernel):

```
grub> echo $root
hd0,msdos1
```

To use GRUB's ls command to list the files and directories in that root, you can append a forward slash to the end of the partition:

```
grub> ls (hd0,msdos1)/
```

But it's a pain to remember and type the actual root partition, so use the root variable to save yourself some time:

```
grub> ls ($root)/
```

The output is a short list of file and directory names on that partition's filesystem, such as etc/, bin/, and dev/. You should realize that this is now a completely different function of the GRUB ls: Before, you were listing devices, partition tables, and perhaps some filesystem header information. Now you're actually looking at the contents of filesystems.

You can take a deeper look into the files and directories on a partition in a similar manner. For example, to inspect the /boot directory, start with the following:

```
grub> ls ($root)/boot
```

You can also view all currently set GRUB variables with the set command:

```
    grub> set
    ?=0
    color_highlight=black/white
    color_normal=white/black
    --snip--
    prefix=(hd0,msdos1)/boot/grub
    root=hd0,msdos1
```

Reviewing Grub.cfg

First, take a quick look at grub.cfg to see how GRUB initializes its menu and kernel options. You'll see that the grub.cfg file consists of GRUB commands, which usually begin with a number of initialization steps followed by a series of menu entries for different kernel and boot configurations. The initialization isn't complicated; it's a bunch of function definitions and video setup commands like this:

```
    if loadfont /usr/share/grub/unicode.pf2 ; then
    set gfxmode=auto
    load_video
    insmod gfxterm
    --snip--
```

Later in this file you should see the available boot configurations, each beginning with the menuentry command. You should be able to read and understand this example based on what you learned in the preceding section:

```
    menuentry 'Ubuntu, with Linux 3.2.0-34-generic-pae' --class ubuntu --class
gnu-linux --class gnu
    --class os {
    recordfail
    gfxmode $linux_gfx_mode
    insmod gzio
    insmod part_msdos
    insmod ext2
    set root='(hd0,msdos1)'
    search --no-floppy --fs-uuid --set=root 70ccd6e7-6ae6-44f6-812c-51aab8036d29
    linux /boot/vmlinuz-3.2.0-34-generic-pae root=UUID=70ccd6e7-6ae6-44f6-812c-
51aab8036d29
    ro quiet splash $vt_handoff
    initrd /boot/initrd.img-3.2.0-34-generic-pae
    }
```

To see how the configuration generation works, look at the very beginning

of grub.cfg. There should be comment lines such as this:

```
    ### BEGIN /etc/grub.d/00_header ###
```

This command by itself simply prints the configuration to the standard output.)

```
    # grub-mkconfig
```

To write and install a newly generated GRUB configuration file, you can write the configuration to your GRUB directory with the -o option to grub-mkconfig, like this:

```
    # grub-mkconfig -o /boot/grub/grub.cfg
```

Installing GRUB on Your System

Installing the boot loader requires that you or an installer determine the following:

- The target GRUB directory as seen by your currently running system. That's usually /boot/grub, but it might be different if you're installing GRUB on another disk for use on another system.
- The current device of the GRUB target disk.
- For UEFI booting, the current mount point of the UEFI boot partition.

use this command to install GRUB on the MBR:

```
# grub-install /dev/sda
```

When running grub-install, tell it where those files should go as follows:

```
# grub-install --boot-directory=/mnt/boot /dev/sdc
```

The grub-install command runs this if it's available, so in theory all you need to do to install on an UEFI partition is the following: # grub-install --efi-directory=efi_dir –- bootloader-id=name

• UEFI Secure Boot Problems

One of the newest problems affecting Linux installations is the secure boot feature found on recent PCs. When active, this mechanism in UEFI requires boot loaders to be digitally signed by a trusted authority in order to run. Microsoft has required vendors shipping Windows 8 to use secure boot. The result is that if you try to install an unsigned boot loader (which is most current Linux distributions), it will not load.

The easiest way around this for anyone with no interest in Windows is to disable secure boot in the EFI settings. However, this won't work cleanly for dual-boot systems and may not be an option for all users. Therefore, Linux distributions are offering signed boot loaders. Some solutions are just frontends to GRUB, some offer a fully signed loading sequence (from the boot loader to the kernel), and others are entirely new boot loaders (some based on efilinux).

• Introduction to init

The init program is a user-space program like any other program on the Linux system, and you'll find it in /sbin along with many of the other system binaries. Its main purpose is to start and stop the essential service processes on the system, but newer versions have more responsibilities.

There are three major implementations of init in Linux distributions:

System V init

A traditional sequenced init (Sys V, usually pronounced “sys-five”). Red Hat Enterprise Linux and several other distributions use this version.

systemd

The emerging standard for init. Many distributions have moved to systemd, and most that have not yet done so are planning to move to it.

Upstart

The init on Ubuntu installations. However, as of this writing, Ubuntu has also planned to migrate to systemd.

systemd and Upstart attempt to remedy the performance issue by allowing many services to start in parallel thereby speeding up the boot process. Their implementations are quite different, though:

- systemd is goal oriented. You define a target that you want to achieve, along with its dependencies, and when you want to reach the target. systemd satisfies the dependencies and resolves the target. systemd can also defer the start of a service until it is absolutely needed.
- Upstart is reactionary. It receives events and, based on those events, runs jobs that can in turn produce more events, causing Upstart to run more jobs, and so on.

Finally, systemd and Upstart both offer some level of on-demand services. Rather than trying to start all the services that may be necessary at boot time (as the System V init would do), they start some services only when needed. This idea is not really new; this was done with the traditional inetd daemon, but the new implementations are more sophisticated.

Both systemd and Upstart offer some System V backward compatibility. For example, both support the concept of runlevels.

System V Runlevels

At any given time on a Linux system, a certain base set of processes (such as crond and udevd) is running. In System V init, this state of the machine is called its runlevel, which is denoted by a number from 0 through 6. A system spends most of its time in a single runlevel, but when you shut the machine down, init switches to a different

runlevel in order to terminate the system services in an orderly fashion and to tell the kernel to stop.

You can check your system's runlevel with the who -r command. A system running Upstart responds with something like this:

```
$ who -r
run-level 2 2015-09-06 08:37
```

This output tells us that the current runlevel is 2, as well as the date and time that the runlevel was established.

Runlevels serve various purposes, but the most common one is to distinguish between system startup, shutdown, single-user mode, and console mode states. For example, Fedora-based systems traditionally used runlevels 2 through 4 for the text console; a runlevel of 5 means that the system will start a GUI login.

But runlevels are becoming a thing of the past. Even though all three init versions in this book support them, systemd and Upstart consider runlevels obsolete as end states for the system. To systemd and Upstart, runlevels exist primarily to start services that support only the System V init scripts, and the implementations are so different that even if you're familiar with one type of init, you won't necessarily know what to do with another.

- **Processes and Resource Utilization**
- **Tracing Program Execution and System Calls**

The tools we've seen so far examine active processes. However, if you have no idea why a program dies almost immediately after starting up, even lsof won't help you. In fact, you'd have a difficult time even running lsof concurrently with a failed command.

The strace (system call trace) and ltrace (library trace) commands can help you discover what a program attempts to do. These tools produce extraordinarily large amounts of output, but once you know what to look for, you'll have more tools at your disposal for tracking down problems.

- strace Recall that a system call is a privileged operation that a user-space process asks the kernel to perform, such as opening and reading data from a file. The strace utility prints all the system calls that a process makes. To see it in action, run this command:
-
```
$ strace cat /dev/null
```

first lines of the output from this command should show execve() in action, followed by a memory initialization call, brk(), as follows:

```
    execve("/bin/cat", ["cat", "/dev/null"], [/* 58 vars */]) = 0
    brk(0) = 0x9b65000
    access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
    mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb77b5000
    access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
    open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
    --snip--
    open("/lib/libc.so.6", O_RDONLY) = 3
    read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\200^\1"..., 1024) =
1024
```

In addition, skip past the mmap output until you get to the lines that look like this:

```
    fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0
    open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
    fstat64(3, {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
    fadvise64_64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
    read(3, "", 32768) = 0
    close(3) = 0
    close(1) = 0
    close(2) = 0
    exit_group(0) = ?
```

- ltrace

The ltrace command tracks shared library calls. The output is similar to that of strace, which is why we're mentioning it here, but it doesn't track anything at the kernel level. Be warned that there are many more shared library calls than system calls. You'll definitely need to filter the output, and ltrace itself has many built-in options to assist you.

Tracking Processes

You learned how to use ps in Section 2.16 to list processes running on your system at a particular time. The ps command lists current processes, but it does little to tell you how processes change over time. Therefore, it won't really help you to determine which process is using too much CPU time or memory.

The top program is often more useful than ps because it displays the current system status as well as many of the fields in a ps listing, and it updates the display every second. Perhaps most important is that top shows the most active processes (that is, those currently taking up the most CPU time) at the top of its display.

You can send commands to top with keystrokes. These are some of the most important commands:

`Spacebar` Updates the display immediately.
`M` Sorts by current resident memory usage.
`T` Sorts by total (cumulative) CPU usage.

`P` Sorts by current CPU usage (the default).
`u` Displays only one user's processes.
`f` Selects different statistics to display.
`?` Displays a usage summary for all top commands.
Two other utilities for Linux, similar to top, offer an enhanced set of views and features: `atop` and `htop`. Most of the extra features are available from other utilities. For example, htop has many of abilities of the `lsof` command described in the next section.

Introduction to Resource Monitoring

Now we'll discuss some topics in resource monitoring, including processor (CPU) time, memory, and disk I/O. We'll examine utilization on a systemwide scale, as well as on a per-process basis.

Many people touch the inner workings of the Linux kernel in the interest of improving performance. However, most Linux systems perform well under a distribution's default settings, and you can spend days trying to tune your machine's performance without meaningful results, especially if you don't know what to look for. So rather than think about performance as you experiment with the tools in this, think about seeing the kernel in action as it divides resources among processes.

Measuring CPU Time

To monitor one or more specific processes over time, use the -p option to top, with this syntax:

```
$ top -p pid1 [-p pid2 ...]
```

To find out how much CPU time a command uses during its lifetime, use time. Most shells have a built-in time command that doesn't provide extensive statistics, so you'll probably need to run /usr/bin/time. For example, to measure the CPU time used by ls, run

```
$ /usr/bin/time ls
```

After ls terminates, time should print output like that below. The key fields are in boldface:

```
0.05user 0.09system 0:00.44elapsed 31%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (125major+51minor)pagefaults 0swaps
```

Monitoring CPU and Memory Performance with vmstat

Among the many tools available to monitor system performance, the `vmstat` command is one of the oldest, with minimal overhead. You'll find it handy for getting a high-level view of how often the kernel is swapping pages in and out, how busy the CPU is, and IO utilization.

The trick to unlocking the power of `vmstat` is to understand its output. For example, here's some output from `vmstat 2`, which reports statistics every 2 seconds:

```
$ vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
2 0 320416 3027696 198636 1072568 0 0 1 1 2 0 15 2 83 0
2 0 320416 3027288 198636 1072564 0 0 0 1182 407 636 1 0 99 0
1 0 320416 3026792 198640 1072572 0 0 0 58 281 537 1 0 99 0
0 0 320416 3024932 198648 1074924 0 0 0 308 318 541 0 0 99 1
0 0 320416 3024932 198648 1074968 0 0 0 0 208 416 0 0 99 0
0 0 320416 3026800 198648 1072616 0 0 0 0 207 389 0 0 100 0
```

Now, watch what happens when a big program starts up sometime later (the first two lines occur right before the program runs):

```
procs -----------memory---------- ---swap-- -----io---- -system-- ----cpu----
r b swpd free buff cache si so bi bo in cs us sy id wa
1 0 320412 2861252 198920 1106804 0 0 0 0 2477 4481 25 2 72 0u
1 0 320412 2861748 198924 1105624 0 0 0 40 2206 3966 26 2 72 0
1 0 320412 2860508 199320 1106504 0 0 210 18 2201 3904 26 2 71 1
1 1 320412 2817860 199332 1146052 0 0 19912 0 2446 4223 26 3 63 8
2 2 320284 2791608 200612 1157752 202 0 4960 854 3371 5714 27 3 51 18v
1 1 320252 2772076 201076 1166656 10 0 2142 1190 4188 7537 30 3 53 14
0 3 320244 2727632 202104 1175420 20 0 1890 216 4631 8706 36 4 46 14
```

I/O Monitoring

By default, vmstat shows you some general I/O statistics. Although you can get very detailed per-partition resource usage with vmstat -d, you'll get a lot of output from this option, which might be overwhelming. Instead, try starting out with a tool just for I/O called iostat.

Using iostat

Like vmstat, when run without any options, iostat shows the statistics for your machine's current uptime:

```
$ iostat
[kernel information]
avg-cpu: %user %nice %system %iowait %steal %idle
4.46 0.01 0.67 0.31 0.00 94.55
Device: tps kB_read/s kB_wrtn/s kB_read kB_wrtn
sda 4.67 7.28 49.86 9493727 65011716
sde 0.00 0.00 0.00 1230 0
```

The avg-cpu part at the top reports the same CPU utilization information as other utilities that you've seen in this chapter, so skip down to the bottom, which shows you the following for each device:

`tps` Average number of data transfers per second
`kB_read/s` Average number of kilobytes read per second
`kB_wrtn/s` Average number of kilobytes written per second
`kB_read` Total number of kilobytes read
`kB_wrtn` Total number of kilobytes written
By default, the iostat output omits partition information. To show all of the partition information, use the -p ALL option. Because there are many partitions on a typical system, you'll get a lot of output. Here's part of what you might see:

```
$ iostat -p ALL
--snip--
Device: tps kB_read/s kB_wrtn/s kB_read kB_wrtn
--snip--
sda 4.67 7.27 49.83 9496139 65051472
sda1 4.38 7.16 49.51 9352969 64635440
sda2 0.00 0.00 0.00 6 0
sda5 0.01 0.11 0.32 141884 416032
scd0 0.00 0.00 0.00 0 0
--snip--
sde 0.00 0.00 0.00 1230 0
```

There is a continuously updating display that shows the processes using the most I/O, with a general summary at the top:

```
# iotop
Total DISK READ: 4.76 K/s | Total DISK WRITE: 333.31 K/s
TID PRIO USER DISK READ DISK WRITE SWAPIN IO> COMMAND
260 be/3 root 0.00 B/s 38.09 K/s 0.00 % 6.98 % [jbd2/sda1-8]
2611 be/4 juser 4.76 K/s 10.32 K/s 0.00 % 0.21 % zeitgeist-daemon
2636 be/4 juser 0.00 B/s 84.12 K/s 0.00 % 0.20 % zeitgeist-fts
1329 be/4 juser 0.00 B/s 65.87 K/s 0.00 % 0.03 % soffice.b~ash-pipe=6
6845 be/4 juser 0.00 B/s 812.63 B/s 0.00 % 0.00 % chromium-browser
19069 be/4 juser 0.00 B/s 812.63 B/s 0.00 % 0.00 % rhythmbox
```

- # Networking and Its Configuration
- # Network Basics

This type of network is ubiquitous; most home and small office networks are configured this way. Each machine connected to the network is called a host. The hosts are connected to a router, which is a host that can move data from one network to another. These machines (here, Hosts A, B, and C) and the router form a local area network (LAN). The connections on the LAN can be wired or wireless.

The router is also connected to the Internet—the cloud in the figure. Because the router is connected to both the LAN and the Internet, all machines on the LAN also have access to the Internet through the router. One of the goals of this chapter is to see how the router provides this access.

Network Layers

A fully functioning network includes a full set of network layers called a network stack. Any functional network has a stack. The typical Internet stack, from the top to bottom layer, looks like this:

- **Application layer**

Contains the "language" that applications and servers use to communicate; usually a high-level protocol of some sort. Common application layer protocols include Hypertext Transfer Protocol (HTTP, used for the Web), Secure Socket Layer (SSL), and File Transfer Protocol (FTP). Application layer protocols can often be combined. For example, SSL is commonly used in conjunction with HTTP.

- **Transport layer**

Defines the data transmission characteristics of the application layer. This layer includes data integrity checking, source and destination ports, and specifications for breaking application data into packets (if the application layer has not already done so). Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are the most common transport layer protocols. The transport layer is also sometimes called the protocol layer.

- **Network or Internet layer Defines how to move packets from a source**

host to a destination host. The particular packet transit rule set for the Internet is known as Internet Protocol (IP). Because we'll only talk about Internet networks in this book, we'll really only be talking about the Internet layer. However, because network layers are meant to be hardware independent, you can simultaneously configure several independent network layers (such as IP, IPv6, IPX, and AppleTalk) on a single host.

- **Physical layer Defines how to send raw data across a physical medium,**

such as Ethernet or a modem. This is sometimes called the link layer or host-to-network layer.

- **Resolving Hostnames**

One of the final basic tasks in any network configuration is hostname resolution with DNS. You've already seen the host resolution tool that translates a name such as www.example.com to an IP address such as 10.23.2.132.

DNS differs from the network elements we've looked at so far because it's in the application layer, entirely in user space. Technically, it is slightly out of place in this chapter alongside the Internet and physical layer discussion, but without proper DNS configuration, your Internet connection is practically worthless. No one in their right mind advertises IP addresses for websites and email addresses because a host's IP address is subject to change and it's not easy to remember a bunch of numbers. Automatic network configuration services such as DHCP nearly always include DNS configuration.

Nearly all network applications on a Linux system perform DNS lookups. The resolution process typically unfolds like this:

- The application calls a function to look up the IP address behind a hostname. This function is in the system's shared library, so the application doesn't need to know the details of how it works or whether the implementation will change.
- When the function in the shared library runs, it acts according to a set of rules (found in /etc/nsswitch.conf) to determine a plan of action on lookups. For example, the rules usually say that even before going to DNS, check for a manual override in the /etc/hosts file.
- When the function decides to use DNS for the name lookup, it consults an additional configuration file to find a DNS name server. The name server is given as an IP address.
- The function sends a DNS lookup request (over the network) to the name server.
- The name server replies with the IP address for the hostname, and the function returns this IP address to the application. This is the simplified version. In a typical modern system, there are more actors attempting to speed up the transaction and/or add flexibility. Let's ignore that for now and take a closer look at the basic pieces.

This is the simplified version. In a typical modern system, there are more actors attempting to speed up the transaction and/or add flexibility. Let's ignore that for now and take a closer look at the basic pieces.

- ## /etc/hosts

On most systems, you can override hostname lookups with the /etc/hosts file. It usually looks like this:

```
127.0.0.1 localhost
10.23.2.3 atlantic.aem7.net atlantic
10.23.2.4 pacific.aem7.net pacific
```

- ## resolv.conf

The traditional configuration file for DNS servers is /etc/resolv.conf. When things were simpler, a typical example might have looked like this, where the ISP's name server addresses are 10.32.45.23 and 10.3.2.3: search mydomain.example.com example.com

```
nameserver 10.32.45.23
nameserver 10.3.2.3
```

- # Firewalls

Routers in particular should always include some kind of firewall to keep undesirable traffic out of your network. A firewall is a software and/or hardware configuration that usually sits on a router between the Internet and a smaller network, attempting to ensure that nothing "bad" from the Internet harms the smaller network. You can also set up firewall features for each machine where the machine screens all of its incoming and outgoing data at the packet level (as opposed to the application layer, where server programs usually try to perform some access control of their own). Firewalling on individual machines is sometimes called IP filtering.

A system can filter packets when it

- receives a packet,
- sends a packet, or
- forwards (routes) a packet to another host or gateway.

With no firewalling in place, a system just processes packets and sends them on their way. Firewalls put checkpoints for packets at the points of data transfer identified above. The checkpoints drop, reject, or accept packets, usually based on some of these criteria:

- The source or destination IP address or subnet
- The source or destination port (in the transport layer information)
- The firewall's network interface

Linux Firewall Basics

In Linux, you create firewall rules in a series known as a chain. A set of chains makes up a table. As a packet moves through the various parts of the Linux networking

subsystem, the kernel applies the rules in certain chains to the packets. For example, after receiving a new packet from the physical layer, the kernel activates rules in chains corresponding to input.

Let's look at how the IP tables system works in practice. Start by viewing the current configuration with this command:

```
# iptables -L
```

The output is usually an empty set of chains, as follows:

```
Chain INPUT (policy ACCEPT)
target prot opt source destination
Chain FORWARD (policy ACCEPT)
target prot opt source destination
Chain OUTPUT (policy ACCEPT)
target prot opt source destination
```

To set the policy on a chain, use iptables -P like this:

```
# iptables -P FORWARD DROP
```

Say that someone at 192.168.34.63 is annoying you. To prevent them from talking to your machine, run this command:

```
# iptables -A INPUT -s 192.168.34.63 -j DROP
```

To see the rule in place, run iptables -L:

```
Chain INPUT (policy ACCEPT)
target prot opt source destination
DROP all -- 192.168.34.63 anywhere
```

Unfortunately, your friend at 192.168.34.63 has told everyone on his subnet to open connections to your SMTP port (TCP port 25). To get rid of that traffic as well, run

The IP table list for INPUT now looks like this:

```
Chain INPUT (policy ACCEPT)
target prot opt source destination
DROP all -- 192.168.34.63 anywhere
DROP tcp -- 192.168.34.0/24 anywhere tcp dpt:smtp
```

However, it doesn't work. To see why, look at the new chain: Chain INPUT (policy ACCEPT) target prot opt source destination DROP all -- 192.168.34.63 anywhere DROP tcp -- 192.168.34.0/24 anywhere tcp dpt:smtp ACCEPT all -- 192.168.34.37 anywhere

The solution is to move the third rule to the top. First, delete the third rule with this command:

```
# iptables -D INPUT 3
```

Then insert that rule at the top of the chain with iptables -I:

```
# iptables -I INPUT -s 192.168.34.37 -j ACCEPT
```

Secure Shell (SS H)

Every server works a bit differently. Let's take a close look at one—the standalone SSH server. One of the most common network service applications is the secure shell (SSH), the de facto standard for remote access to a Unix machine. When configured, SSH allows secure shell logins, remote program execution, simple file sharing, and more—replacing the old, insecure telnet and rlogin remote-access systems with public-key cryptography for authentication and simpler ciphers for session data. Most ISPs and cloud providers require SSH for shell access to their services, and many Linuxbased network appliances (such as NAS devices) allow access via SSH as well. OpenSSH (http://www.openssh.com/) is a popular free SSH implementation for Unix, and nearly all Linux distributions come with it preinstalled. The OpenSSH client is ssh, and the server is sshd. There are two main SSH protocol versions: 1 and 2. OpenSSH supports both, but version 1 is rarely used.

Among its many useful capabilities and features, SSH does the following:

- Encrypts your password and all other session data, protecting you from snoopers.
- Tunnels other network connections, including those from X Window System clients. You'll learn more about X in Chapter 14.
- Offers clients for nearly any operating system.
- Uses keys for host authentication.

The SSHD Server

Running sshd requires a configuration file and host keys. Most distributions keep configurations in the /etc/ssh configuration directory and try to configure everything properly for you if you install their sshd package. (The configuration filename sshd_config is easy to confuse with the client's ssh_config setup file, so be careful.) You shouldn't need to change anything in sshd_config, but it never hurts to check. The file consists of keyword-value pairs, as shown in this fragment:

```
Port 22
#Protocol 2,1
#ListenAddress 0.0.0.0
#ListenAddress ::
HostKey /etc/ssh/ssh_host_key
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_dsa_key
```

Lines beginning with # are comments, and many comments in your sshd_config might indicate default values. The sshd_config(5) manual page contains descriptions of all possible values, but these are the most important ones:

`HostKey` file Uses file as a host key. (Host keys are described shortly.) `LogLevel level` Logs messages with syslog level level. `PermitRootLogin value` Permits the superuser to log in with SSH if value is set to yes. Set value to no to prevent this. `SyslogFacility name` Logs messages with syslog facility name. `X11Forwarding` value Enables X Window System client tunneling if value is set to yes. `XAuthLocation` path Provides a path for xauth. X11 tunneling will not work without this path. If xauth isn't in /usr/bin, set path to the full pathname for xauth.

OpenSSH Key Files

Filename	Key Type
ssh_host_rsa_key	Private RSA key (version 2)
ssh_host_rsa_key.pub	Public RSA key (version 2)
ssh_host_dsa_key	Private DSA key (version 2)
ssh_host_dsa_key.pub	Public DSA key (version 2)
ssh_host_key	Private RSA key (version 1)
ssh_host_key.pub	Public RSA key (version 1)

To create SSH protocol version 2 keys, use the ssh-keygen program that comes with OpenSSH: # ssh-keygen -t rsa -N '' -f /etc/ssh/ssh_host_rsa_key # ssh-keygen -t dsa -N '' -f /etc/ssh/ssh_host_dsa_key

For the version 1 keys, use

```
# ssh-keygen -t rsa1 -N '' -f /etc/ssh/ssh_host_key
```

. Diagnostic Tools

Let's look at a few diagnostic tools that are useful for poking around the application layer. Some dig into the transport and network layers, because everything in the application layer eventually maps down to something in those lower layers.

Table reviews a few useful options for viewing connections.

Option	Description
-t	Prints TCP port information
-u	Prints UDP port information
-l	Prints listening ports
-a	Prints every active port
-n	Disables name lookups (speeds things up; also useful if DNS isn't working)

- **lsof**

For a complete list of programs using or listening to ports, run

```
# lsof -i
```

When run as a regular user, this command only shows that user's processes. When run as root, the output should look something like this, displaying a variety of processes and users:

```
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
rpcbind 700 root 6u IPv4 10492 0t0 UDP *:sunrpc
rpcbind 700 root 8u IPv4 10508 0t0 TCP *:sunrpc (LISTEN)
avahi-dae 872 avahi 13u IPv4 21736375 0t0 UDP *:mdns
cupsd 1010 root 9u IPv6 42321174 0t0 TCP ip6-localhost:ipp (LISTEN)
ssh 14366 juser 3u IPv4 38995911 0t0 TCP thishost.local:55457->
somehost.example.com:ssh (ESTABLISHED)
chromium- 26534 juser 8r IPv4 42525253 0t0 TCP thishost.local:41551->
anotherhost.example.com:https (ESTABLISHED)
```

- **tcpdump**

If you need to see exactly what's crossing your network, tcpdump puts your network interface card into promiscuous mode and reports on every packet that crosses the wire. Entering tcpdump with no arguments produces output like the following, which includes an ARP request and web connection:

```
# tcpdump
tcpdump: listening on eth0
20:36:25.771304 arp who-has mikado.example.com tell duplex.example.com
20:36:25.774729 arp reply mikado.example.com is-at 0:2:2d:b:ee:4e
20:36:25.774796 duplex.example.com.48455 > mikado.example.com.www: S
3200063165:3200063165(0) win 5840 <mss 1460,sackOK,timestamp 38815804[|tcp]>
```

```
    (DF)
    20:36:25.779283 mikado.example.com.www > duplex.example.com.48455: S
    3494716463:3494716463(0) ack 3200063166 win 5792 <mss 1460,sackOK,timestamp
    4620[|tcp]> (DF)
    20:36:25.779409 duplex.example.com.48455 > mikado.example.com.www: . ack 1 win
    5840 <nop,nop,timestamp 38815805 4620> (DF)
    20:36:25.779787 duplex.example.com.48455 > mikado.example.com.www: P
1:427(426)
    ack 1 win 5840 <nop,nop,timestamp 38815805 4620> (DF)
    20:36:25.784012 mikado.example.com.www > duplex.example.com.48455: . ack 427
    win 6432 <nop,nop,timestamp 4620 38815805> (DF)
    20:36:25.845645 mikado.example.com.www > duplex.example.com.48455: P
1:773(772)
    ack 427 win 6432 <nop,nop,timestamp 4626 38815805> (DF)
    20:36:25.845732 duplex.example.com.48455 > mikado.example.com.www: . ack 773
    win 6948 <nop,nop,timestamp 38815812 4626> (DF)
    9 packets received by filter
    0 packets dropped by kernel
```

- ## netcat

If you need more flexibility in connecting to a remote host than a command like telnet host port allows, use netcat (or nc). netcat can connect to remote

TCP/UDP ports, specify a local port, listen on ports, scan ports, redirect standard I/O to and from network connections, and more. To open a TCP connection to a port with netcat, run

```
    $ netcat host port
```

netcat only terminates when the other side of the connection ends the connection, which can confuse things if you redirect standard input to netcat. You can end the connection at any time by pressing ctrl-C. (If you'd like the program and network connection to terminate based on the standard input stream, try the sock program instead.)

To listen on a particular port, run

```
    $ netcat -l -p port_number
```

- ## Port Scanning

Sometimes you don't even know what services the machines on your networks are offering or even which IP addresses are in use. The Network Mapper (Nmap) program scans all ports on a machine or network of machines looking for open ports, and it lists the ports it finds. Most distributions have an Nmap package, or you can get it at http://www.insecure.org/.

Run nmap host to run a generic scan on a host. For example:

```
    $ nmap 10.1.2.2
```

```
Starting Nmap 5.21 ( http://nmap.org ) at 2015-09-21 16:51 PST
Nmap scan report for 10.1.2.2
Host is up (0.00027s latency).
Not shown: 993 closed ports
PORT STATE SERVICE
22/tcp open ssh
25/tcp open smtp
80/tcp open http
111/tcp open rpcbind
8800/tcp open unknown
9000/tcp open cslistener
9090/tcp open zeus-admin
Nmap done: 1 IP address (1 host up) scanned in 0.12 seconds
```

As you can see, a number of services are open here, many of which are not enabled by default on most distributions. In fact, the only one here that's usually on by default is port 111, the rpcbind port.

. Shell Scripting

. Shell Script Basics

Bourne shell scripts generally start with the following line, which indicates that the /bin/sh program should execute the commands in the script file. (Make sure that no whitespace appears at the beginning of the script file.)

```
#!/bin/sh
```

The #! part is called a shebang; you'll see it in other scripts in this book. You can list any commands that you want the shell to execute following the #!/bin/sh line. For example:

```
#!/bin/sh
#
# Print something, then run ls
echo About to run the ls command.
ls
```

After creating a shell script and setting its permissions, you can run it by placing the script file in one of the directories in your command path and then running the script name on the command line. You can also run ./script if the script is located in your current working directory, or you can use the full pathname.

As with any program on Unix systems, you need to set the executable bit for a shell script file, but you must also set the read bit in order for the shell to read the file. The easiest way to do this is as follows:

```
$ chmod +rx script
```

This chmod command allows other users to read and execute script. If you don't want that, use the absolute mode 700 instead

Important Shell Script Utilities

Several programs are particularly useful in shell scripts. Certain utilities such as basename are really only practical when used with other programs, and therefore don't often find a place outside shell scripts. However, others such as awk can be quite useful on the command line, too.

basename

If you need to strip the extension from a filename or get rid of the directories in a full pathname, use the basename command. Try these examples on the command line to see how the command works:

```
$ basename example.html .html
$ basename /usr/local/bin/example
```

Shell Script Utilities

This example shows how you can use basename in a script to convert GIF image files to the PNG format:

```
#!/bin/sh
for file in *.gif; do
# exit if there are no files
if [ ! -f $file ]; then
exit
fi
b=$(basename $file .gif)
echo Converting $b.gif to $b.png...
giftopnm $b.gif | pnmtopng > $b.png
done
```

awk

The awk command is not a simple single-purpose command; it's actually a powerful programming language. Unfortunately, awk usage is now something of a lost art, having been replaced by larger languages such as Python.

This said, many, many people use awk to do one thing—to pick a single field out of an input stream like this:

```
$ ls -l | awk '{print $5}'
```

This command prints the fifth field of the ls output (the file size). The result is a list of file sizes.

sed

A very common task for sed is to substitute some text for a regular expression (see Section 2.5.1), like this:

```
$ sed 's/exp/text/'
```

So if you wanted to replace the first colon in /etc/passwd with a % and send the result to the standard output, you'd do it like this:

```
$ sed 's/:/%/' /etc/passwd
```

To substitute all colons in /etc/passwd, add a g modifier to the end of the operation, like this:

```
$ sed 's/:/%/g' /etc/passwd
```

Here's a command that operates on a per-line basis; it reads /etc/passwd and deletes lines three through six and sends the result to the standard output:

```
$ sed 3,6d /etc/passwd
```

You can also use a regular expression as the address. This command deletes any line that matches the regular expression exp:

```
$ sed '/exp/d'
```

- **xargs**

When you have to run one command on a huge number of files, the command or shell may respond that it can't fit all of the arguments in its buffer. Use xargs to get around this problem by running a command on each filename in its standard input stream.

```
$ find . -name '*.gif' -print | xargs file
```

When writing a script, use the following form instead, which changes the find output separator and the xargs argument delimiter from a newline to a NULL character:

```
$ find . -name '*.gif' -print0 | xargs -0 file
```

Here's how to perform the preceding task using only find:

```
$ find . -name '*.gif' -exec file {} \;
```

- **expr**

If you need to use arithmetic operations in your shell scripts, the expr command can help (and even do some string operations). For example, the command expr 1 + 2 prints 3. (Run expr --help for a full list of operations.)

The expr command is a clumsy, slow way of doing math. If you find yourself using it frequently, you should probably be using a language like Python instead of a shell script.

• exec

The exec command is a built-in shell feature that replaces the current shell process with the program you name after exec. It carries out the exec() system call that you learned about in Chapter 1. This feature is designed for saving system resources, but remember that there's no return; when you run exec in a shell script, the script and shell running the script are gone, replaced by the new command.

To test this in a shell window, try running exec cat. After you press ctrl-D or ctrl-C to terminate the cat program, your window should disappear because its child process no longer exists.

• Subshells

Say you need to alter the environment in a shell slightly but don't want a permanent change. You can change and restore a part of the environment (such as the path or working directory) using shell variables, but that's a clumsy way to go about things. The easy way around these kinds of problems is to use a subshell, an entirely new shell process that you can create just to run a command or two. The new shell has a copy of the original shell's environment, and when the new shell exits, any changes you made to its shell environment disappear, leaving the initial shell to run as normal.

To use a subshell, put the commands to be executed by the subshell in parentheses. For example, the following line executes the command uglyprogram in uglydir and leaves the original shell intact:

```
$ (cd uglydir; uglyprogram)
```

This example shows how to add a component to the path that might cause problems as a permanent change:

```
$ (PATH=/usr/confusing:$PATH; uglyprogram)
```

Using a subshell to make a single-use alteration to an environment variable is such a common task that there is even a built-in syntax that avoids the subshell:

```
$ PATH=/usr/confusing:$PATH uglyprogram
```

Pipes and background processes work with subshells, too. The following example uses tar to archive the entire directory tree within orig and then unpacks the archive into the new directory target, which effectively duplicates the files and folders in orig (this is useful because it preserves ownership and permissions, and it's generally faster than using a command such as cp -r):

```
$ tar cf - orig | (cd target; tar xvf -)
```

• Including Other Files in Scripts

If you need to include another file in your shell script, use the dot (.) operator. For example, this runs the commands in the file config.sh:

```
. config.share
```

This "include" file syntax does not start a subshell, and it can be useful for a group of scripts that need to use a single configuration file.

- ## Reading User Input

The read command reads a line of text from the standard input and stores the text in a variable. For example, the following command stores the input in $var:

```
$ read var
```

This is a built-in shell command that can be useful in conjunction with other shell features