Tuning and Customizing a Linux System

DANIEL L. MORRILL

Apress™

Tuning and Customizing a Linux System

ISBN (pbk): 1-893115-27-5

Printed and bound in the United States of America 12345678910

Technical Reviewer: Douglas Kilpatrick

Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Development Editor and Indexer: Valerie Perry

Copy Editors: Kim Goodfriend, Ami Knox, Nicole LeClerc

Production Editor: Kari Brooks

Compositor: Diana Van Winkle, Van Winkle Design Group

Artist: Cara Brunk, Blue Mud Productions

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.

Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

Part Two

Linux Distributions

Welcome to Part Two of *Tuning and Customizing a Linux System*! Part Two introduces the concept of a distribution of Linux software and then explains three such distributions in detail. After reading Part Two, you'll have a generalized understanding of what a Linux distribution actually is, which in turn will enable you to work with any distribution you encounter and figure it out. Additionally, you'll come away with a basic functional knowledge of the three distributions discussed.

The three distributions studied in detail are Red Hat Linux 7.3 (in Chapter 4), Slackware Linux 8.0 (in Chapter 5), and Debian GNU/Linux (in Chapter 6). The distributions are compared to one another, and both their similarities and their differences are highlighted. The ultimate goal is to provide you a basic mastery of three different Linux distributions and by extension *all* distributions. After reading Part Two, you'll be able to either use these distributions themselves or draw upon your knowledge to learn some other distribution not directly covered in this book.

Part Three builds on the material discussed in this part. That is, where Part Two teaches you everything you would want to know about the workings of a distribution, Part Three will show you how to customize a distribution by installing and configuring software on it. Finally, Part Four will put it all together and demonstrate some real-world case studies. For these reasons, Part Two is an absolutely crucial part of this book, since it's the foundation of so much else.

CHAPTER 3

The Nature of a Distribution

In Chapter 1 you learned what a Linux system is, what the various parts are, and how it all fits together. That should go a long way toward understanding Linux systems in general, right?

Well, it does, but there's more to it than that. For each of the components of a Linux system we've seen, there are several options. The kernel, system libraries, and other parts of the system usually have many different "live" versions; some are stable and well tested but no longer "current," while the cutting-edge versions may be buggy or unstable. Plus, sometimes there are competing options even for the same program, as in the case of desktop software where GNOME and KDE are both excellent choices.

The responsibility of a Linux vendor is to produce a system that works. This means selecting the best possible software products and versions for the task at hand. A particular configuration of such a collection of software is known as a *distribution*. This chapter will discuss the notion of a distribution and what makes up a distribution, and will describe how distributions are created.

Taking a Snapshot of the Linux Continuum

The phrase "Linux Continuum", is not from a science fiction book nor is it a bizarre cultist rite. Rather, it is really just a way of describing the overall state of the software that collectively becomes a Linux system. A lot of software packages from a lot of different sources go into a Linux system. All this software isn't just out there growing on a tree or falling like manna from heaven. It's actually out there growing on a million different trees, and the real manna is sometimes hidden under mud and grime. It can be hard work hunting down all this software and cobbling it together into a usable system. This is what Linux vendors do, and the end result is a distribution.

The vast majority of software packages aren't written explicitly to become part of a particular Linux distribution. Because of this, at any given time, each of these software packages has its own version, and even its own versioning scheme.

For example, at the time of this writing the current version of the emacs editor is 21.2, while the current stable version of XFree86 is 4.2.0.

Figure 3-1 depicts how a distribution is really just a sampling (or a "snapshot") of the larger overall continuum of Linux software. You can see from this figure how Red Hat Linux 7.3 is made up of a number of different packages, each with its own version. Obviously, Red Hat Linux consists of far more packages than just the few shown in Figure 3-1; however, it does show three important software packages:

- XFree86 implementation of the X Window System
- The Linux kernel
- KDE desktop

Each of these packages has its own version history, release cycle, etc. Red Hat Linux, meanwhile, constructs its distribution from particular versions of the software. The three packages represent an example continuum, while Red Hat Linux represents a snapshot. In this case, Red Hat Linux 7.3 shipped with XFree86 4.2.0, Linux 2.4.18, and KDE 3.0. Remember that this is a *snapshot* of the continuum and doesn't (usually) change over time; Red Hat may choose the latest version of a package available at the time it was released, but later versions may be released after Red Hat has released their own snapshot.

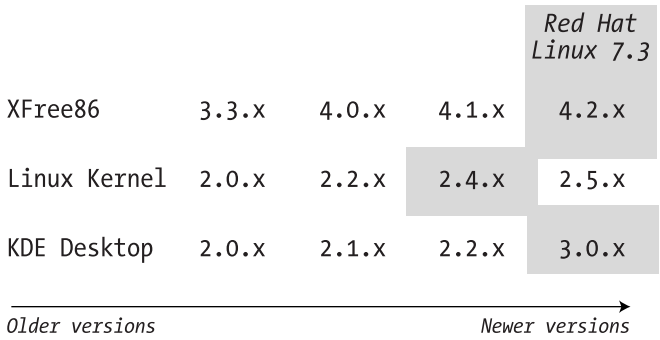

Figure 3-1. Example software continuum

NOTE *A Linux distribution is a collection of software that makes up a Linux-based operating system. Because each of the software packages in a distribution has its own independent version, the version number that a vendor assigns to its distribution does not bear any relationship to the versions of the software packages which comprise it.*

If all these various software packages and versions are starting to get blurred together in your mind, that's great, because it's the perfect way to view things. You really can view the state of open source software as one great big "space" of software and version numbers: a continuum. This continuum is constantly evolving and changing, as individual software packages develop and change versions. What a vendor calls a production-quality release is really just a snapshot taken of this continuum at one particular instant.

The responsibility of a vendor is to craft the snapshot so that the overall system is as useful, flexible, and bug-free as possible. Vendors do this by first identifying which software packages are going to go into the distribution at all, and then selecting the most recent stable version of that software that meets their goals.

Understanding the Goals of a Distribution

Linux systems are extremely flexible. Linux (the kernel) can be scaled up to large multi-processor systems, or scaled down into tiny memory-constrained PDAs. Obviously, the software requirements for a distribution running on the multi-processor system are going to be rather different that the software requirements for a PDA—unless, of course, someone's lucky enough to have a 32-processor PDA with 8GB of memory (that won't spontaneously burst into flames!).

So, the first decision that a distribution vendor has to make is what, exactly, are the goals of the distribution. Is the distribution to be for servers, or workstations? Is the goal a highly user-friendly environment suitable for desktop office use, or is the goal a high-performance traditional Unix environment for engineers? Is the distribution supposed to run on Intel, Alpha, PowerPC, SPARC, MIPS, ARM, or IBM S/390 mainframe?

This is, of course, a marketing decision (whether for a commercial entity or for a volunteer free effort), and different vendors target different markets. Red Hat Software, for example, produces a well-rounded distribution suitable for use as either a server or a desktop system. The Debian project produces a distribution similar in scope to Red Hat, but consisting entirely of free software. Mandrake Software produces a desktop-focused distribution, and TurboLinux produces a high-end distribution for clustering and parallel computing. The Slackware distribution aims to create a traditional Unix-like system. In other words, the market

each vendor targets defines the overall requirements of the distribution, and the particular selection of software that goes into the distribution.

Each user, meanwhile, also has specific needs. Some users need to set up a workstation for their own personal use, others need to set up a local file server, and still others need to set up extensive server farms to run a web site. Some Linux distributions, of course, are good at some tasks, and others are better at other tasks. Sometimes, the user has the luxury of picking the best distribution for the task. Frequently, however, the user prefers to use a specific distribution (perhaps by personal preference or corporate mandate) that might not have been optimized out of the box for the task. In these cases, it helps the user immensely to understand how the distribution was built and what its goals are, in order to customize the system for the desired task. This book will help the reader understand and customize the system, but it's up to the reader to know what her own goals and needs are.

This book covers three sample distributions:

- Red Hat Linux
- Slackware Linux
- Debian GNU/Linux

These distributions were chosen because of their huge influence on the evolution of Linux systems; almost every other distribution out there is derived from, inspired by, or related to one of these three. Despite their different focuses, however, each of these distributions consist of the same general set of software; they're snapshots of the same continuum, just from different angles. The rest of Part Two (in Chapters 4 through 6) is dedicated to discussing each of these distributions in detail, but first you need to understand exactly what a distribution is; that's the focus of the remainder of this chapter.

Dissecting a Linux Distribution

This section will describe the key components of a Linux distribution, which are:

- Linux kernel version
- Packaging format
- Filesystem layout
- System startup scripts

- System library versions
- X Window Desktop
- Userspace software

After reading this section, readers will understand the basics of what goes into a distribution; later chapters cover specific sample distributions in depth, but this section provides an introduction. Users who are already familiar with a Linux distribution or are experienced Unix users should be able to skim or skip this chapter, and proceed to the detailed discussions of the sample distributions, starting with Red Hat Linux in Chapter 4.

Linux Kernel

The Linux kernel is the core of the operating system (see Chapter 1). Generally, the Linux kernel has two "trees": the stable version and the development version. The *stable version* is a production-ready kernel, and is debugged and well tested by millions of users all over the world. The *development version*, in contrast, is where the key kernel developers do their tinkering and feature enhancements. The development tree is thus highly unstable, and is not for production systems, daily use, or people prone to apoplexy.

CROSS-REFERENCE *See Chapter 1 for a discussion of the Linux operating system.*

The Linux kernel has a numbering scheme of X.Y.Z, described as follows:

- X is the major version, representing major architectural changes
- Y is the minor version, indicating functional enhancements but few architectural changes
- Z is the build version, representing bug-fix or development patch levels

If Y is an even number, the kernel is stable; if it is odd, the kernel is in development. The current stable kernel series at the time of this writing is 2.4.*x*, and the current development series is 2.5.*x*. (A "series" is simply a series of similar versions.)

Generally speaking, distribution vendors are interested in producing stable, bug-free systems. Thus, any formal releases should obviously make use of only stable Linux kernels; otherwise the vendors risk exposing their users to excessive bugs in the kernel—the most sensitive and critical part of the system.

On the other hand, the Linux kernel, like many open source projects, has a long release cycle, when compared to many commercial efforts. This means that many of the distribution vendors often find that they need features that are new or experimental, and exist only in the current development kernels. Distribution vendors often must decide whether they will opt for a stable, less fully featured kernel, or a more fully equipped but probably less stable version.

Many distribution vendors adopt a compromise, by *backporting* features from development kernels to the previous release kernels. (The process of adding such advanced features to an older kernel is known as *backporting*.) Since a development kernel that is to become the next stable kernel was originally derived from the current stable version, it still shares many similarities with the older kernel. Thus, many of the features and bug fixes that are added to a typical development kernel are not strictly incompatible with the previous stable kernel.

The core kernel developers (led by Linus Torvalds) do not have enough time to try and manage these backports to older kernels; they are already spending all their available time on improving the new kernel. Many distribution vendors therefore do this work, and release customized kernels with their products. For example, universal serial bus (USB) support was added to the Linux kernel in the 2.3 development series, but Red Hat wanted USB support for their 2.2-based Red Hat Linux 7.0 distribution. They chose to do the work to backport the 2.3-version USB functionality, and shipped Red Hat 7.0 with a custom kernel. Moreover, distribution vendors sometimes even add functionality that isn't in the kernel at all, in either the development or stable trees; for example, Mandrake software has kernel patches used to make removable media easier to manage. This really isn't isolated behavior, and many distributions ship with customized kernels.

Backporting development kernel features isn't always possible, of course; sometimes new features in a development kernel depend on architectural changes that were made to the kernel that don't exist in the previous stable version, and so can't be backported. Still, in a lot of cases development features are considered unstable only because they are new and not yet extensively tested. Some vendors may disagree and regard these features as suitable for production or may have features of their own that they wish to add. In such cases, vendors may choose to ship a custom kernel in their distribution.

Some people disapprove of this behavior, citing the danger of fragmenting the kernel into many custom derivatives, and compatibility issues that might

arise when vendors customize a kernel so much that it differs from the "standard" kernels in subtle ways. However, you can view the "official" kernel as a sort of starting point for distribution vendors to begin with for developing their systems. After all, does it really matter whether the USB features you're using came from the 2.3 kernel or a 2.2 kernel, as long as your USB digital camera works?

Packaging Format

A distribution's *packaging format* is the mechanism used to install and manage the software that comes with a distribution. Since many of these software applications are placed side-by-side in the same directories, it is extremely useful to have a way to track which particular files were installed by which packages. Package managers also provide the ability to uninstall or upgrade software, and query the system for which software is installed.

Package management tools also generally provide versioning and dependency information. For example, a particular version of the KDE X Windows desktop requires a particular version of the Qt widget library. For KDE to run properly, the appropriate version of Qt must also be installed, but determining version information isn't always easy. Package managers can track this information, helping users and system administrators keep their software and versions in sync.

Every flavor of Unix—including the commercial ones—has its own packaging format or software maintenance mechanism. However, as with many other areas, the Linux package formats are far more functional and easier to use than most of the commercial Unix equivalents. Indeed, some commercial Unix vendors have even announced plans to include the RPM packaging format with their own systems.

Red Hat Package Manager

The first sophisticated packaging manager for Linux systems was the *Red Hat Package Manager* (RPM), developed by Red Hat Software for their distribution. RPM supports all of the standard packaging tasks, and has an extensive list of features. It installs, uninstalls, and upgrades packages, runs scripts during installations and upgrades that do extra work to configure the software, and supports package dependency tracking and version enforcement. RPM also supports extensive querying of package contents and documentation, has facilities for verifying the integrity of installed packages, and can even generate "source RPMs," for supporting users who wish or need to customize the RPMs. RPM's largest contribution to packaging software was probably its ability to reliably perform package upgrades and extensive dependency checks.

Deb Files

The Debian project has its own alternative packaging format. Since these files have the extension .deb, they are frequently known as "Deb files." The Debian packaging format is similar in functionality to RPM, but has some additional features, such as more extensive scripting capability and the ability to create "meta" packages for encapsulating other packages. Debian systems also have additional user-interface tools intended to make the packaging system easier to use.

RPM and the Debian format are the two most popular packaging formats around. Most other distributions use one of these two formats. A notable exception is Slackware, which uses its own format based on traditional tar files and searchable text databases. There are also supporting tools for these packaging systems that provide additional features; for example, both RPM and Debian's system have tools for searching servers for particular packages and automatically downloading and installing them.

Packaging Format vs. Distribution

It's important not to confuse the packaging format with the distribution itself. Two distributions can use the same packaging format, and yet have incompatible versions of the same software. For example, both Caldera and Red Hat use the RPM format for their distributions. However, an RPM created for a Red Hat system may not necessarily work on a Caldera OpenLinux system, which may have different versions of the libraries that the package requires. In other words, a packaging format only manages the installation and configuration of software, not the software itself. The packaging format doesn't change the fact that a piece of software requires another piece, it just lets you install it.

Functionally, there isn't that much difference between the various packaging formats. They all do the same things. Therefore, the choice of which format to use isn't made on technical or quality issues, but really on package availability issues. That is, whom do you want to be compatible with? Currently, the majority of software that is packaged for Linux is packaged in the RPM format. This doesn't mean that users of other formats (such as Debian) can't use the software; for a lot of software, .deb packages can be found in addition to RPM, and you always have the option of installing the software manually (or building it from source code) without using the packaging tool at all. However, if you're browsing the Web and find a random piece of software, you're more likely to find the Linux version packaged in RPM than in Debian's format.

Even this distinction is somewhat moot, since Debian's distribution also includes tools that can convert RPM files into Debian-compatible packages and

vice versa. As a user, you're far more likely to run into compatibility problems with required libraries than you are with the format used to package the software. Once either of these formats is installed, the same files are extracted anyway, after all.

That said, it currently appears that RPM has the greater momentum. More distributions use it, and it appears to be the de facto standard software-packaging format for Linux systems. However, the open source community is nothing if not dynamic, and this situation could change the week after this book is published.

Filesystem Layout

Even the greatest Linux distribution is worthless if you can't find the file you're looking for. From the very earliest days of Unix, the conventions for which files and programs go where on the disk have been a major consideration. Your average Unix-like system is a lot more complex than many other operating systems, and so having standards for where to find things is a big deal. The particular filesystem convention chosen by a vendor has a big impact on the rest of the system.

There is any number of ways to lay out the contents of a distribution on the disk, so there isn't really a set number of options to choose from. The Unix legacy has left behind a basic set of conventions for what programs go in what directories, but traditionally those have been rough guidelines at best, and the various vendors of both commercial Unix flavors and free systems alike have followed these loose conventions with varying degrees of rigor.

Directories

The /etc directory on Unix systems generally contains configuration files, scripts, and similar contents. However, Solaris systems place some administrative programs in this directory as well, whereas Linux systems and BSD systems typically place such programs in /sbin. Additionally, sometimes it's the software itself that determines where to place its files: on BSD systems, some third-party software such as Apache puts its configuration files in /usr/local/etc, where on most Linux systems even third-party software puts its configuration files in /etc.

Filesystem Hierarchy Standard

There is an ostensible convention for directory layout on Linux systems. The Filesystem Hierarchy Standard (FHS) addresses the confusion surrounding directory layouts on Unix systems, and has a set of standard guidelines for which types

of files should go in which directories. The FHS aims to be applicable to any Unix-like system, though it was originated and tailored for Linux systems. Most Linux distributions follow the FHS reasonably well, though there are a few exceptions. The open-source BSD systems also generally follow the FHS, but commercial vendors are proving to be somewhat slower to adopt the FHS, probably due to concerns for backward compatibility.

Table 3-1 summarizes the major aspects of the FHS, by showing the major directories identified by the FHS. This is a good place to start, but users who truly want to master the "Zen of Linux" should definitely read the entire document. The FHS can be found at `www.pathname.com/fhs/`. Once you've digested the FHS, you'll have made the first step to understanding a Linux system, which is simply knowing where to find things.

Table 3-1. Filesystem Hierarchy Standard Summary

DIRECTORY	NAME	TYPICAL CONTENTS	EXAMPLES
/etc	"etcetera"	Configuration files	/etc/passwd: user account information; /etc/resolv.conf: network lookup information
/boot	Boot files	Files required for the system to boot up	/boot/vmlinuz: the Linux kernel image
/dev	Device nodes	Files representing hardware devices on the system	/dev/sda: the first SCSI disk in the system; /dev/audio: the sound device
/lib	System libraries	Shared library files required by user programs	/lib/libc.so: the shared library (.so) for C programs; /lib/libpthread.so.0: the POSIX threads library
/bin	Program binaries	User application programs	/bin/ls: file listing utility; /bin/bash: the "Bourne Again SHell" command shell
/sbin	Administrative	Programs used primarily for administrative tasks programs	/sbin/ifconfig: network device configuration tool; /sbin/mkfs: filesystem formatting tool
/home	User home	Home directories for users	(Optional) /home/morrildl: the home directory for the user "morrildl"
/mnt	Mount points	Contains subdirectories where temporary filesystems can be mounted	/mnt/cdrom: mount point for the CD-ROM drive; /mnt/floppy: mount point for the floppy drive

Table 3-1. Filesystem Hierarchy Standard Summary (continued)

DIRECTORY	NAME	TYPICAL CONTENTS	EXAMPLES
/usr	User partition	Files useful to users once the system is running	/usr/X11R6: files related to the X Window system; /usr/bin: end-user program binaries
/var	Runtime programs	Files and directories for storing control information or data for programs and services	/var/spool/mail: email-related files; /var/log: system log files
/tmp	Temporary files	Files that are not persistent or important	

A bit of explanation of Table 3-1 is in order, especially with respect to the "/" directory and the "/usr" directory. Unix paths begin with a "/" character, and the top-level directory itself is "/". This is known as the *root directory* or *root partition*; if someone asks you to "cd into the root directory," they're actually asking you to type the command `cd /`. Similarly, the /usr directory is pronounced "slash user."

When you read the FHS, you'll probably notice that it defines two copies of most directories, for example, /bin and /usr/bin. The reason for this is related to the way the system boots up. During boot-up, Unix systems typically mount the root partition first, and only mount the user partitions later on in the process. Also, an administrator will occasionally boot the system into single-user or maintenance mode, to perform major upgrades, configuration maintenance, or security examinations. In this case, the system might stop at the root partition, and not mount any others at all. In both of these cases, a set of core programs (such as ls, mount, cd, vi, echo, etc.) is required for the system to function.

Since the system requires access to these core programs and files before mounting the rest of the volumes, they obviously have to be located on the root partition. Thus, the top-level directories such as /bin, /lib, and /sbin are there to contain a basic set of commands required for the system to complete the bootstrap process, while /usr/bin, /usr/lib, and /usr/sbin contain additional programs that are most commonly used by administrators on a running system.

For example, the `ls` command is required, so it is in /bin. However, the `find` command is not strictly required (though it is convenient), so it is located in /usr/bin. As a rule of thumb, if you're trying to find a program or are trying to decide where to place a new program, first decide whether it's critical for the system to function. If it is, use /bin, /lib, or /sbin; if it's not, use /usr/bin, /usr/lib, or /usr/sbin.

There is, in fact, even a third case here: the /usr/local directory. /usr/local is used for the same type of files as /usr (that is, files that aren't strictly required for the system to run or start up), but for files that are local to the particular system. For example, the X Windows software is fairly common and standard across all

installations, so it should be placed in /usr. However, an installation of the Apache web server that has been customized for a specific system is not standard and should be placed in /usr/local, rather than /usr.

There are a several rules of thumb to apply when trying to decide where to place a particular file. One approach is that anything that was included with the distribution as it was shipped should be placed in /usr, and everything else should be placed in /usr/local. Alternatively, if it is managed by the system packaging tools, it belongs in /usr; otherwise, it belongs in /usr/local. The point is that the FHS is a somewhat flexible standard, and you can really place a file anywhere you want, as long as you're consistent.

The way that the FHS breaks the filesystem up into these parallel, almost mirror-image directories may seem excessively complicated. However, there is good reason for the added complexity: it makes life easier for system administrators, and more recoverable in the case of disk failures. By breaking up the filesystem into core (root), non-core (/usr), and local (/usr/local) directories, the FHS establishes an easy way for administrators to place different types of files on different disks.

Partitioning with Upgrades in Mind

Recall that the placement of core files on the root directory allows the system to boot up and be managed without having to bother during the boot process with the other partitions on the system. However, this separation of the root directory and /usr means that either the core system or the user-level files and programs could be upgraded independently of each other.

Perhaps more importantly, this separation also means that the core system (the root and /usr volumes) can be upgraded independently of /usr/local and /opt. This allows administrators to upgrade a system without disturbing custom software installed on the system, and also means that the custom software can be preserved if the main system disk containing the root and /usr volumes goes bad. (It's easy to reinstall an operating system, but hard to recover months of work on the Apache web site you had running. It's even worse if the site was intact but you have to overwrite it to reinstall the operating system because they shared the same partition or disk.)

System Startup Scripts

What good is a computer if it won't start up? Following that thought, what good is a startup script if you can't figure out how to use it to start your own software? Linux systems, like all Unix systems, configure their startup and shutdown activities via a set of scripts. The system itself would be perfectly happy to just start up or shut down without doing anything for you, but that would be pretty annoying (and dangerous, if it meant disk partitions don't get properly managed). The startup and shutdown scripts—known as *runlevel command* or *rc* scripts—do the grunt work of starting and stopping your Linux system.

When a Unix system starts up, the kernel is the first thing to get loaded. It then kicks off a program known as *init*, which is responsible for starting up the rest of the system. This init program essentially manages runlevels. A *runlevel* is simply a configuration of programs and services that are to be run; usually, these configurations are geared toward a specific goal, such as normal multiuser operation, shutting down the system, starting it up, entering into maintenance mode, etc. Each runlevel corresponds to a different state; for example, runlevel 0 might mean "system halt" while runlevel 4 might correspond to "normal server mode." The init program is responsible for starting up any services (such as web servers, DNS servers, databases, and so on) and setting up the user environment (such as mounting all the partitions, allowing the user to log in, and starting up X Windows if necessary).

The kernel doesn't actually care how init does this; init could do the work itself, of course, but that's not very customizable, so init instead delegates the majority of its startup tasks to an external program. All init usually does is look for a configuration file in the /etc directory (named /etc/inittab) that tells it which runlevel it should enter into by default, and then invokes a program (almost always a shell script) for that runlevel. In fact, the init program itself is simply a configuration parameter for the kernel. The kernel can just as easily kick off a different program instead of init that has different behavior for setting up the system. For example, if the system has been damaged and needs to be recovered or maintained, the kernel can simply start an interactive command shell instead of init.

Normally, though, the kernel starts init, which runs the scripts for the current runlevel. There are actually seven of these system runlevels, summarized in Table 3-2. Whenever init changes to a new runlevel, it invokes the scripts for that runlevel, and the scripts start and stop programs as required by the new runlevel. Actually, init doesn't even distinguish between which runlevel is which; it just has up to six "modes" it can be in. By Unix convention, the runlevels usually have the meanings expressed in Table 3-2. Sometimes the runlevels will vary by Unix flavor or distribution; to find out exactly what runlevels your system supports, look at the configuration file for init, which is usually /etc/inittab.

Table 3-2. System Runlevels

RUNLEVEL	MEANING	BEHAVIOR
0	Halt	Shuts down all programs and services, unmounts filesystems, and halts the CPU
1	Single-User Mode	Shuts down most programs and services, permitting only root logins; frequently used for system maintenance
2	Networkless Multiuser	Starts all programs and services except network-related ones; allows all local users to log in
3	Normal Multiuser	Starts all programs and services, except X
4	Normal Multiuser	Frequently unused; otherwise just like runlevel 3
5	Graphical Multiuser	Just like runlevel 3, but also starts the X Window system for user logins
6	Reboot	Shuts down all programs and services and reboots the system

Each runlevel has a different meaning, which reflects what programs and services are started in that runlevel. An important concept is that any runlevel can be entered from any other runlevel. For example, runlevel 1 (single-user mode) can be entered either on system startup, or from another runlevel on a running system, such as runlevel 3. From init's perspective (and the perspective of the shell scripts that get run by init), the two cases are identical, and runlevel 1 has the same meaning and runs the same scripts. In other words, the terms "shuts down" and "starts up" are relative; if the system just started up, there's nothing to shut down, and vice versa. The ways in which you instruct init to switch runlevels vary by system; the most "portable" method is to use the `telinit` command, but most systems provide alternative commands such as `shutdown`.

NOTE *Remember, init doesn't do any work itself, but simply invokes the appropriate script for the current runlevel. What actually happens when that script is run is up to the script. This script is like any program, and can be as complex or as simple as the author chooses to make it.*

The format of the init scripts is one of the areas where you can trace the origins of a Unix flavor back to either the original AT&T SysV Unix or BSD operating systems. Each of these systems, though they share the same behavior and

usage of init, has different architectures for the actual scripts that get run by init. Though they fulfill the same purposes, they are fairly different in the details.

Most Unix-like systems, including Linux systems, choose one or the other of the SysV or BSD init-scripts models, and emulate it with varying degrees of accuracy. Red Hat Linux, for example, uses the SysV initscript model, whereas Slackware uses the BSD model. Since we will be covering both of these distributions in depth in Chapters 4 and 5, we'll cover the details of the SysV and BSD mechanisms there, rather than duplicate it here.

Like many scripting activities, whether an individual prefers the SysV or BSD initscript model is really a matter of personal taste. The SysV model can be described as more "modular," which some people prefer, whereas the BSD model is simpler, which other people prefer. It's hard to make a case that either is fundamentally better than another, though, so the initscripts model used by a particular Linux distribution is really immaterial, though it probably reflects which Unix flavor the distribution is trying to emulate.

System Library Versions

As mentioned in Chapter 1, it's really the system libraries that define the character of an operating system. If you chose to spend the time and effort, for example, you could write a set of system libraries that provide a clone of Windows, running directly on a Unix kernel (such as Linux). This would provide a completely different user interface and programming API, but would still have a POSIX-compliant kernel running underneath it all. In other words, by changing the system libraries you can completely alter the nature of an operating system.

You may be wondering what these system libraries are. The system libraries make application development and use easier by encapsulating a body of functionality into, well, a library. This allows end-user programs to reuse functionality by using the library, instead of having to interact with the kernel directly. There are system libraries for most aspects of the operating system, including network access, multithreading and process management, program linking and loading, and so on.

Actually, the term "system libraries" is a bit fuzzy. Libraries like those just mentioned are pretty much required for the system to work. However, there are additional libraries that aren't strictly required, but can still be considered system libraries. The most obvious example is shown in the libraries required for the X Window system to work; these libraries aren't required for the system to run, but they *are* required for the system to run a GUI. Other examples include the ldbm database libraries used by many programs, or the tcp_wrappers library used for security for some network programs. These libraries aren't required, but are still system libraries.

For the vast majority of cases, the only real option for system libraries for a Linux system is the GNU C Library package, known as *glibc*. The glibc package provides most of the libraries that are absolutely critical for the system to run. glibc doesn't include some of the optional system libraries; these other libraries are usually installed individually, as separate packages.

NOTE *Traditionally, Unix systems name their libraries libfoo.o, where "foo" is the name of the library. By this convention, libc refers to the C libraries. The glibc package provides many other libraries in addition to the basic libc.o, so it is somewhat poorly named.*

System libraries are probably one of the three most difficult pieces of software to write, along with a kernel and a compiler. So, once you have a functional set of system libraries (or kernel or compiler), think long and hard before going off and starting your own. For this reason, no one has written a serious competitor to the glibc for Linux systems, and every major distribution out there uses some version of the glibc for the core system libraries.

At least, no one has written a serious competitor for traditional desktop or server distributions. There are, in fact, several efforts to produce a set of system libraries for other hardware environments, such as PDAs, set-top "network appliances," and so forth. Many of these efforts are quite innovative. Transvirtual's PocketLinux, for example, consists of a Linux kernel that essentially uses a Java VM (name Kaffe) as the system libraries; the uCLinux project, meanwhile, has as a goal—a "Linux-on-chip" system—where a stripped-down but functional Linux kernel and rudimentary system library can be embedded into a piece of equipment.

Again, however, these are really still exceptions. As interesting as these efforts are, they only demonstrate the versatility of the Linux kernel, whereas we're interested in a more traditional Linux-based desktop, workstation, or server operating system. For such systems, the glibc is where it's at.

X Window Desktop

Ah, X. The X Window system may be one of the most beloved and reviled pieces of software in history. X has become the de facto (and some people would argue more than de facto) standard windowing system for Unix-like operating systems. X can seem bizarre to those new to it, can seem primitive to casual observers, and can seem incredibly powerful to experienced users.

The X Window system was first released in 1984 by the Massachusetts Institute of Technology's Laboratory for Computer Science, and is now governed by a

consortium of industry and other public and private groups. Since 1984, it has become the GUI of choice for Unix systems. Commonly referred to simply as "X," it has legions of zealous supporters, as well as staunch opponents. Indeed, the fortune cookie amusement program that ships with Red Hat Linux has many anti-X quotes in it; "X Windows: You'll wish we were kidding." and "X Windows: More than enough rope." Love it or hate it, however, everyone admits that X introduced several important architectural features to the GUI scene, including network transparency and a layered design for user interfaces.

Network Transparency

Network transparency means that you can view the output of an X program running on a different machine across the network. That machine, in fact, doesn't need to have X running, or even have a monitor at all. This network transparency was standardized in the X protocol, which allows any GUI to support the viewing of X apps simply by implementing the protocol; for example, you can purchase X servers for Windows that will let you view X applications on your Windows box. This network transparency is one of the primary reasons why X has survived as long as it has; without it, X would have been left in the pixilated dust of its competitors.

Layered GUI Model

X's other major claim to fame is the layered GUI model. The X Window system itself is essentially just an event framework for managing windows and window events. Managing windows involves keeping track of where on the screen windows are located, window dimensions, and so on. Managing window events means sending notifications when buttons are pushed, when windows are resized, when repainting or redrawing windows, and so on. These window management events, together with the network transparency functionality, comprise the *X- Window protocol*. This protocol itself knows nothing about video cards, framebuffers, 3D acceleration, or any other aspect of actually rendering graphics to a display. These lower-level tasks are the responsibility of the X server, which is a program that interacts with the video card, implementing the X protocol.

Neither the X protocol nor the X server has knowledge of what actually goes on in a window, however. That is, X has no facilities for drawing buttons, check-boxes, or any of the other "widgets" that users are familiar with. The X protocol only manages events for these widgets, and the X server only handles drawing the widgets to the video card. X also doesn't provide any user interface components, such as icons, desktops, start menus, and so forth. X itself is very stripped-down

(when compared to other GUIs such as the MacOS or Windows) and relies on two others layers to do the real work: the widget set and the window manager. In other words, the X Window system consists of three parts:

- An X server
- A widget set
- A window manager

These layers are discussed in more detail in the rest of this section.

Widget Set

The *widget set* is a library or a sort of palette of elements commonly used in GUIs. Examples of widgets include the buttons and checkboxes mentioned in the previous section, as well as selection lists, radio buttons, and so forth. Since X imposes no restrictions on what these widgets have to look like, developers and artists are free to make widget sets look however they choose to. There are a lot of widget sets out there, with a very wide variety of appearances. Motif (developed by the Open Software Foundation, with no relationship to the Free Software Foundation) was the first highly popular widget set, and became the *de facto* standard widget set for many years. There are many widget sets available to choose from today; so many, in fact, that it's hard to keep track of them all!

Recently, GNU's GTK+ and TrollTech's Qt toolkits are emerging as modern, full-featured, and open-source libraries. These toolkits are really more than simple widget sets, since they are also providing fairly extensive frameworks for actually building applications. GTK+ was popularized by the GNOME desktop, and Qt was popularized by the K Desktop Environment (KDE).

GNOME and KDE

GNOME and KDE, in turn, are examples of the last layer of an X GUI: the window manager. A *window manager* is responsible for drawing window decorations such as borders, title bars, and the ubiquitous minimize and maximize buttons, as well as the "softer" aspects of providing the icons, desktops, menus, etc. that really characterize a user interface. There are probably as many or more window managers out there as widget toolkits, and frequently there is a window manager for each widget set. For example, in addition to KDE/Qt and GNOME/GTK+, the mwm window manager is based on Motif, while the fvwm and twm window managers have their own internal widget sets.

Window Manager vs. Desktop Environment

While I described KDE and GNOME as window managers, it's actually a little white lie. In reality, they're both examples of software suites that make up a desktop environment. The suites include several different programs, of which a window manager is only one. Typically, KDE and GNOME include:

- Window manager
- Desktop program (for placing icons on the background)
- Button panel (for containing buttons to launch programs)
- Taskbar or window list (for switching between running applications)
- Desktop pager (for switching between virtual desktops)
- Utility programs (for controlling and configuring all the rest)

Different desktops and window managers (including not only KDE and GNOME but others as well) will have some subset or superset of this list. To really unlock your productivity when using your desktop, you should learn about what features are available to you with your environment. Throughout the rest of this book, these suites will be referred to as desktop environments.

In a nutshell, the window manager handles moving, resizing, and closing the windows, while the widget set manages the appearance and behavior of the windows. X, meanwhile, provides the framework to run all of this. This componentized model for the GUI gives developers and users a truly staggering, and even daunting, degree of choice in GUIs. Some of these GUIs are very simple and lightweight, such as twm, short for "Tom's window manager." Others, like KDE and GNOME, are more than window managers and are better described as desktop environments.

Choosing an X Server, Widget Set, and Window Manager

Since there are three "layers" to an X Window environment, there are essentially three choices you need to make regarding the following:

- Which X server to use
- Which widget set to use
- Which window manager to use

For Linux systems and most of the BSD systems, the XFree86 package is the de facto standard X server and library set, so that's an easy choice, leaving only the widget set and window manager to be determined. However, since the widget set and window manager are usually closely related, there's only one choice to make here. In other words, the only choice you really have to make is which window manager or desktop you want to use.

The choice of window manager depends largely on what you want to do with the system. (Now where have we heard that before?) If you're reading this book, you are probably most interested in using it as a desktop, server, or workstation. In these cases, your alternative is Windows or a Macintosh, so the competition is pretty stiff. If this is what you're after, your best choices are KDE or GNOME.

KDE and GNOME are both very complete environments. They provide not only a window manager, but also various applications such as icon panels (for launching applications), task bars, and various other features that will make most Windows users feel right at home. KDE and GNOME also have features that competitors can't touch—even Windows and the Macintosh—such as the ability to support custom "themes" or "skins" that let the user customize the desktop right down to the widget set level. Figures 3-2 and 3-3 show the same KDE desktop running two completely different themes. Figure 3-2 depicts Alessandro Rossini's "Acqua GRAPHITE" theme, while Figure 3-3 depicts Martin Doege's "LCARS ACCESS 411" theme. Both themes can be obtained from the site `kde.themes.org`.

KDE and GNOME generally lag Windows only in some of the infrastructure aspects. For example, Windows has a more extensive, pervasive, and robust component object model (COM) for embedding application functionality and communicating between applications, which KDE and GNOME are just now starting to address. This becomes especially evident in the realm of office productivity applications, where COM is used extensively. However, both KDE and GNOME have embeddable component framework efforts underway, and are gaining ground quickly. The 2.1 version of KDE, for example, offers some features that even Windows doesn't have. Additionally, industry consortiums have recently formed behind both KDE and GNOME, including support from all the major commercial Unix vendors; this support could accelerate the progress of these environments even further.

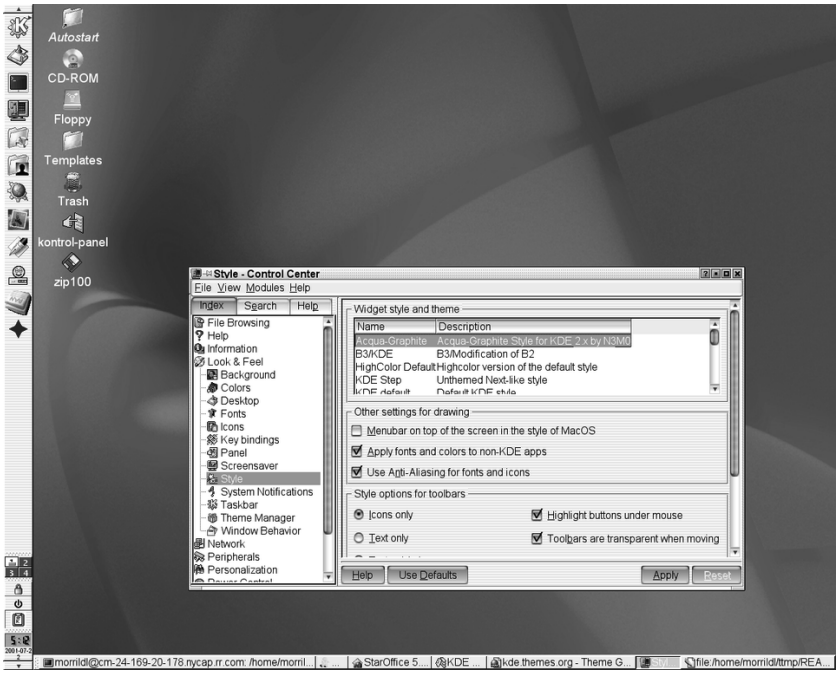

Figure 3-2. Example KDE theme: Acqua GRAPHITE

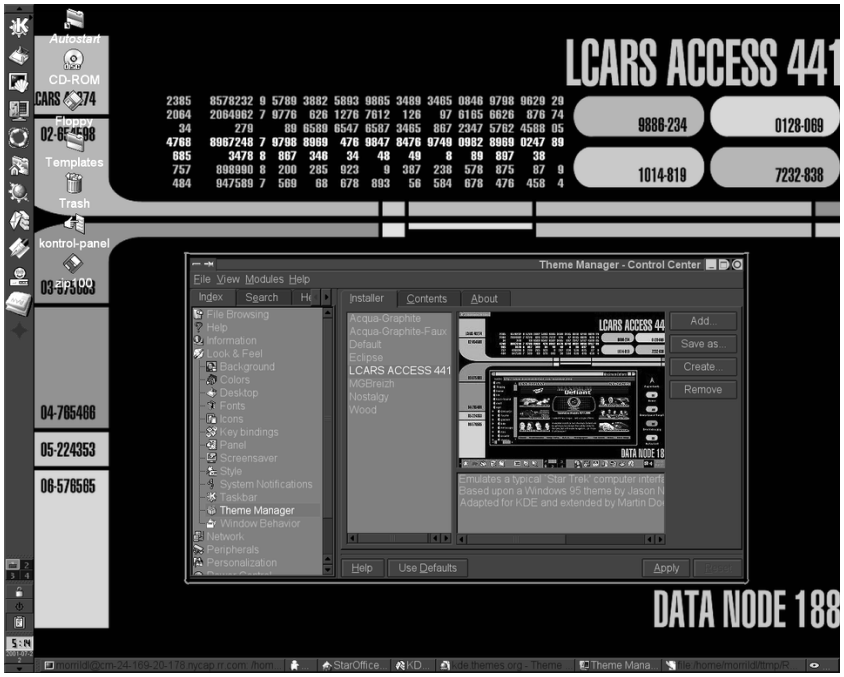

Figure 3-3. Example KDE theme: LCARS ACCESS

Application Software

So far, I've talked a lot about the system itself. That's fine if you're an enthusiast interested in technology for the sake of technology. However, the rest of us are interested in technology for our own sakes: what can it help us do? (Hopefully you've noticed this recurring theme by now.) On this note, a Linux system is only as useful as the productivity software installed on it. The real objective of a Linux distribution is to provide as much useful software as possible; everything up to this point is just enabling us to install and run what we really want to get at.

So, what kinds of software can you actually use with Linux? The answer is, all kinds. However, there's a lot of software out there, and not all of it is useful to everyone. Just as you don't want to leave out a software package users want, you don't want to add too many they don't need, either. So, to determine what kind of software to install, you first have to determine what kind of system you're installing.

Generally, there are three ways people use computers:

- Desktops (or personal computer)
- Servers
- Workstations

A *desktop* is generally used by very non-technical people who just want to get work done. *Servers* are run by individuals or organizations that need to provide services to each other or the public. A *workstation*, though, is a hybrid: it's part server and part desktop. A workstation is generally a desktop that frequently runs heftier applications (such as engineering design applications or scientific computing) and sometimes a few services (such as a low-volume web server). Microsoft Windows ME and the Macintosh OS are good desktop operating systems; Windows NT is a good workstation OS; Solaris and Linux are good workstation and server operating systems. (These examples are just generalities, of course; there's nothing that says the Mac OS can't be a workstation operating system.)

Why doesn't Linux make a good desktop operating system? Well, some people will argue that it does. However, there's a good deal more complexity to a Linux system than your average desktop OS, and users of desktop operating systems generally eschew complexity in all its myriad forms. The proof is in the pudding: Linux has exploded on the server and workstation, so if it's not on the desktop, there must be a reason. However, it's important to note the distinction between "has not" and "can not": Linux systems "have not" experienced vast penetration on the desktop yet; that doesn't mean they "can not." KDE and GNOME have extremely strong stories on the desktop, and only time will tell.

Why are KDE and GNOME so strong? Well, because contrary to popular belief, there is software for Linux systems. From office suites to digital camera software to image editors to web browsers and email clients, it's there on the desktop. From web servers to databases to LDAP directories to network file-sharing software, it's there on the server. From scientific computing and visualization to engineering applications to software development, it's there on the professional workstation. The main problem is that this software can be rather hard to locate and install; it sometimes takes a software developer just to get a program to run. Well, either a software developer or someone who's read this book, that is.

Enter the Linux Standards Base

It's probably becoming clear at this point that there are a lot of variables in creating a Linux distribution. Each distribution is independent, and the fact that two different distributions are based on the same Linux kernel does not by any means imply that they are compatible with one another. In fact, they frequently are not. Software compiled for one distribution may not run on another, and this increases the burden for software developers, who may end up having to support their software on multiple distributions.

To address this fact, the Linux Standards Base (LSB) project was created. This project is a collaboration between individuals and companies involved in Linux systems and other open-source or free software activities. The goal of the organization is to establish a basic set of core software that distribution vendors can include in their products—in essence, a sort of "mini-distribution" that vendors can use as the core of their own systems. By standardizing on the LSB, the various distributions can improve interoperability and compatibility between systems. At the time of this writing, the LSB had only recently released a formal 1.0.0 specification, and so no major distribution vendors support it yet. However, the LSB (and its sponsoring organization, the Free Standards Group) may become a major influence on Linux systems in the future. The LSB can be found at `www.linuxbase.org`.

Summary

In this chapter, you read about the basics of a distribution. You learned that as different as the various distributions are from each other, in the end they all must solve the same set of problems, such as which software to include and how to lay out the filesystem. Once you understand this, the task of mastering a new distribution is as simple (or as hard!) as figuring out how the distribution solves each of these problems.

In the next three chapters, you'll read about three actual distributions. These chapters discuss the distributions in detail and dissect them to illustrate how each solves these same basic problems. After reading these chapters, you'll have a broad set of real-world details to flesh out the general understanding you gained from this chapter.