INTEGRAL UNIVERSITY,LUCKNOW

DIRECTORATE OF DISTANCE EDUCATION

BCA-505 Paper Code: LOS/B

LINUX OPERATING SYSTEM

CONTENTS

UNIT: 1-INTRODUCTION TO LINUX

Contents

- Introduction to Linux
- Linux Distribution
- Operating Syatem and Linux
- Linux
- Unix
- Open Source Software
- Software Repositories
- Online Linux Information
- Souces
- Linux Documentation
- Review & Self Assessment Question
- Further Readings

Introduction to Linux

Linux is a fast and stable open source operating system for personal computers (PCs) and workstations that features professional-level Internet services, extensive development tools, fully functional graphical user interfaces (GUIs), and a massive number of applications ranging from office suites to multimedia applications. Linux was developed in the early 1990s by Linus Torvalds, along with other programmers around the world. As an operating system, Linux performs many of the same functions as Unix, Macintosh, Windows, and Windows NT. However, Linux is distinguished by its power and flexibility, along with being freely available. Most PC operating systems, such as Windows, began their development within the confines of small, restricted PCs, which have only recently become more versatile machines. Such operating systems are constantly being upgraded to keep up with the ever-changing capabilities of PC hardware. Linux, on the other hand, was developed in a different context. Linux is a PC version of the Unix operating system that has been used for decades on mainframes and minicomputers and is currently the system of choice for network servers and workstations. Linux brings the speed, efficiency, scalability, and flexibility of Unix to your PC, taking advantage of all the capabilities that PCs can now provide.

Technically, Linux consists of the operating system program, referred to as the kernel, which is the part originally developed by Linus Torvalds. But it has always been distributed with a massive number of

software applications, ranging from network servers and security programs to office applications and development tools. Linux has evolved as part of the open source software movement, in which independent programmers joined together to provide free, high-quality software to any user. Linux has become the premier platform for open source software, much of it developed by the Free Software Foundation's GNU project. Many of these applications are bundled as part of standard Linux distributions. Currently, thousands of open source applications are available for Linux from sites like SourceForge, Inc.'s **sourceforge.net**, K Desktop Environment's (KDE's) **kde-apps.org**, and GNU Network Object Model Environment's (GNOME's) **gnomefiles.org**. Most of these applications are also incorporated into the distribution repository, using packages that are distribution compliant.

Along with Linux's operating system capabilities come powerful networking features, including support for Internet, intranets, and Windows networking. As a norm, Linux distributions include fast, efficient, and stable Internet servers, such as the web, File Transfer Protocol (FTP), and DNS servers, along with proxy, news, and mail servers. In other words, Linux has everything you need to set up, support, and maintain a fully functional network. With both GNOME and KDE, Linux also provides GUIs with that same level of flexibility and power. Unlike Windows and the Mac, Linux enables you to choose the interface you want and then customize it further, adding panels, applets, virtual desktops, and menus, all with full drag-and-drop capabilities and Internet-aware tools.

Linux does all this at the right price. Linux is free, including the network servers and GUI desktops. Unlike the official Unix operating system, Linux is distributed freely under a GNU general public license as specified by the Free Software Foundation, making it available to anyone who wants to use it. GNU (the acronym stands for "GNUs Not Unix") is a project initiated and managed by the Free Software Foundation to provide free software to users, programmers, and developers. Linux is copyrighted, not public domain. However, a GNU public license has much the same effect as the software's being in the public domain. The GNU GPL is designed to ensure Linux remains free and, at the same time, standardized. Linux is technically the operating system kernel—the core operations—and only one official Linux kernel exists. People sometimes have the mistaken impression that Linux is somehow less than a professional operating system because it is free. Linux is, in fact, a PC, workstation, and server version of Unix. Many consider it far more stable and much more

powerful than Windows. This power and stability have made Linux an operating system of choice as a network server.

To appreciate Linux completely, you need to understand the special context in which the Unix operating system was developed. Unix, unlike most other operating systems, was developed in a research and academic environment. In universities, research laboratories, data centers, and enterprises, Unix is the system most often used. Its development has paralleled the entire computer and communications revolution over the past several decades. Computer professionals often developed new computer technologies on Unix, such as those developed for the Internet.Although a sophisticated system,Unix was designed from the beginning to be flexible. The Unix system itself can be easily modified to create different versions. In fact, many different vendors maintain different official versions of Unix. IBM, Sun, and Hewlett- Packard all sell and maintain their own versions of Unix. The unique demands of research programs often require that Unix be tailored to their own special needs. This inherent flexibility in the Unix design in no way detracts from its quality. In fact, this flexibility attests to the ruggedness of Unix, allowing it to adapt to practically any environment. This is the context in which Linux was developed. Linux is, in this sense, one other version of Unix — a version for the PC. The development of Linux by computer professionals working in a research like environment reflects the way Unix versions have usually been developed. Linux is publicly licensed and free—and reflects the deep roots Unix has in academic institutions, with their sense of public service and support. Linux is a top-rate operating system accessible to everyone, free of charge.

Linux Distributions

Although there is only one standard version of Linux, there are actually several different distributions. Different companies and groups have packaged Linux and Linux software in slightly different ways. Each company or group then releases the Linux package, usually on a CD-ROM. Later releases may include updated versions of programs or new software. Some of the more popular distributions are Red Hat, Ubuntu, Mepis, SUSE, Fedora, and Debian. The Linux kernel is centrally distributed through **ker** Linux has spawned a great variety of distributions. Many aim to provide a comprehensive solution providing support for any and all task. These include distributions like SUSE, Red Hat, and Ubuntu. Some are variations on other distributions, like Centos, which is based on Red Hat Enterprise Linux, and Ubuntu, which derives from Debian Linux. Others have been developed for more specialized tasks or to support certain

features. Distributions like Debian provide cutting edge developments. Some distributions provide more commercial versions, usually bundled with commercial applications such as databases or secure servers. Certain companies like Red Hat and Novell provide a commercial distribution that corresponds to a supported free distribution. The free distribution is used to develop new features, like the Fedora Project for Red Hat. Other distributions like Knoppix and Ubuntu specialize in Live-CDs, the entire Linux operating system on single CD.

Operating Systems and Linux

An operating system is a program that manages computer hardware and software for the user. Operating systems were originally designed to perform repetitive hardware tasks, which centered around managing files, running programs, and receiving commands from the user. You interact with an operating system through a user interface, which allows the operating system to receive and interpret instructions sent by the user. You need only send an instruction to the operating system to perform a task, such as reading a file or printing a document. An operating system's user interface can be as simple as entering commands on a line or as complex as selecting menus and icons on a desktop.

An operating system also manages software applications. To perform different tasks, such as editing documents or performing calculations, you need specific software applications. An editor is an example of a software application that enables you to edit a document, making changes and adding new text. The editor itself is a program consisting of instructions to be executed by the computer. For the program to be used, it must first be loaded into computer memory, and then its instructions are executed. The operating system controls the loading and execution of all programs, including any software applications. When you want to use an editor, simply instruct the operating system to load the editor application and execute it.

File management, program management, and user interaction are traditional features common to all operating systems. Linux, like all versions of Unix, adds two more features. Linux is a multiuser and multitasking system. As it is a multitasking system, you can ask the system to perform several tasks at the same time. While one task is being done, you can work on another. For example, you can edit a file while another file is being printed. You do not have to wait for the other file to finish printing before you edit. As it is a multiuser system, several users can log in to the system at the same time, each interacting with the system through his or her own terminal.

As a version of Unix, Linux shares that system's flexibility, a flexibility stemming from Unix's research origins. Developed by Ken Thompson at AT&T Bell Laboratories in the late 1960s and early 1970s, the Unix system incorporated many new developments in operating system design. Originally, Unix was designed as an operating system for researchers. One major goal was to create a system that could support the researchers' changing demands. To do this, Thompson had to design a system that could deal with many different kinds of tasks. Flexibility became more important than hardware efficiency. Like Unix, Linux has the advantage of being able to deal with the variety of tasks any user may face. The user is not confined to limited and rigid interactions with the operating system. Instead, the operating system is thought of as making a set of highly effective tools available to the user. This user oriented philosophy means you can configure and program the system to meet your specific needs. With Linux, the operating system becomes an operating environment.

History of Unix and Linux

As a version of Unix, the history of Linux naturally begins with Unix. The story begins in the late 1960s, when a concerted effort to develop new operating system techniques occurred. In 1968, a consortium of researchers from General Electric, AT&T Bell Laboratories, and the Massachusetts Institute of Technology carried out a special operating system research project called MULTICS (the Multiplexed Information and Computing Service). MULTICS incorporated many new concepts in multitasking, file management, and user interaction.

Unix

In 1969, Ken Thompson, Dennis Ritchie, and the researchers at AT&T Bell Laboratories developed the Unix operating system, incorporating many of the features of the MULTICS research project. They tailored the system for the needs of a research environment, designing it to run on minicomputers. From its inception,Unix was an affordable and efficient multiuser and multitasking operating system.

The Unix system became popular at Bell Labs as more and more researchers started using the system. In 1973, Dennis Ritchie collaborated with Ken Thompson to rewrite the programming code for the UNIX system in the C programming language.Unix gradually grew from one person's tailored design to a standard software product distributed by many different vendors, such as Novell and IBM. Initially, Unix was treated as a research product. The first versions of Unix were distributed free to the computer science departments of many noted universities. Throughout the

1970s, Bell Labs began issuing official versions of Unix and licensing the systems to different users. One of these users was the computer science department of the University of California, Berkeley. Berkeley added many new features to the system that later became standard. In 1975 Berkeley released its own version of Unix, known by its distribution arm, Berkeley Software Distribution (BSD). This BSD version of Unix became a major contender to the AT&T Bell Labs version. AT&T developed several research versions of Unix, and in 1983 it released the first commercial version, called System 3. This was later followed by System V, which became a supported commercial software product.

At the same time, the BSD version of Unix was developing through several releases. In the late 1970s, BSD Unix became the basis of a research project by the Department of Defense's Advanced Research Projects Agency (DARPA). As a result, in 1983, Berkeley released a powerful version of Unix called BSD release 4.2. This release included sophisticated file management as well as networking features based on Internet network protocols—the same protocols now used for the Internet. BSD release 4.2 was widely distributed and adopted by many vendors, such as Sun Microsystems.

In the mid-1980s, two competing standards emerged, one based on the AT&T version of Unix and the other based on the BSD version. AT&T's Unix System Laboratories developed System V release 4. Several other companies, such as IBM and Hewlett-Packard, established the Open Software Foundation (OSF) to create their own standard version of Unix. Two commercial standard versions of Unix existed then—the OSF version and System V release 4.

Linux

Originally designed specifically for Intel-based PCs, Linux started out at the University of Helsinki as a personal project of a computer science student named Linus Torvalds. At that time, students were making use of a program called Minix, which highlighted different Unix features. Minix was created by Professor Andrew Tanenbaum and widely distributed over the Internet to students around the world. Linus's intention was to create an effective PC version of Unix for Minix users. It was named Linux, and in 1991, Linus released version 0.11. Linux was widely distributed over the Internet, and in the following years, other programmers refined and added to it, incorporating most of the applications and features now found in standard Unix systems. All the major window managers have been ported to Linux. Linux has all the networking tools, such as FTP support, web browsers, and the whole range of network services such as email, the

domain name service, and dynamic host configuration, along with FTP, web, and print servers. It also has a full set of program development utilities, such as C++ compilers and debuggers. Given all its features, the Linux operating system remains small, stable, and fast. In its simplest format, Linux can run effectively on only 2MB of memory.

Although Linux has developed in the free and open environment of the Internet, it adheres to official Unix standards. Because of the proliferation of Unix versions in the previous decades, the Institute of Electrical and Electronics Engineers (IEEE) developed an independent Unix standard for the American National Standards Institute (ANSI). This new ANSI-standard Unix is called the Portable Operating System Interface for Computer Environments (POSIX). The standard defines how a Unix-like system needs to operate, specifying details such as system calls and interfaces. POSIX defines a universal standard to which all Unix versions must adhere. Most popular versions of Unix are now POSIX-compliant. Linux was developed from the beginning according to the POSIX standard. Linux also adheres to the Linux file system hierarchy standard (FHS), which specifies the location of files and directories in the Linux file structure.

Linux development is now overseen by The Linux Foundation (**linux-foundation.org**), which is a merger of The Free Standards Group and Open Source Development Labs (OSDL). This is the group that Linus Torvalds works with to develop new Linux versions.

Linux Overview

Like Unix, Linux can be generally divided into three major components: the kernel, the environment, and the file structure. The kernel is the core program that runs programs and manages hardware devices, such as disks and printers. The environment provides an interface for the user. It receives commands from the user and sends those commands to the kernel for execution. The file structure organizes the way files are stored on a storage device, such as a disk. Files are organized into directories. Each directory may contain any number of subdirectories, each holding files. Together, the kernel, the environment, and the file structure form the basic operating system structure. With these three, you can run programs, manage files, and interact with the system.

An environment provides an interface between the kernel and the user. It can be described as an interpreter. Such an interface interprets commands entered by the user and sends them to the kernel. Linux provides several kinds of environments: desktops, window managers, and command line shells. Each user on a Linux system has his or her own user

interface. Users can tailor their environments to their own special needs, whether they be shells, window managers, or desktops. In this sense, for the user, the operating system functions more as an operating environment, which the user can control.

In Linux, files are organized into directories, much as they are in Windows. The entire Linux file system is one large interconnected set of directories, each containing files. Some directories are standard directories reserved for system use. You can create your own directories for your own files, as well as easily move files from one directory to another. You can even move entire directories and share directories and files with other users on your system. With Linux, you can also set permissions on directories and files, allowing others to access them or restricting access to yourself alone. The directories of each user are, in fact, ultimately connected to the directories of other users. Directories are organized into a hierarchical tree structure, beginning with an initial root directory. All other directories are ultimately derived from this first root directory.

With KDE and GNOME, Linux now has a completely integrated GUI. You can perform all your Linux operations entirely from either interface. KDE and GNOME are fully operational desktops supporting drag-and-drop operations, enabling you to drag icons to your desktop and to set up your own menus on an Applications panel. Both rely on an underlying X Window System, which means as long as they are both installed on your system, applications from one can run on the other desktop. The GNOME and KDE sites are particularly helpful for documentation, news, and software you can download for those desktops. Both desktops can run any X Window System program, as well as any cursor-based program such as Emacs and Vi, which were designed to work in a shell environment. At the same time, a great many applications are written just for those desktops and included with your distributions. KDE and GNOME have complete sets of Internet tools, along with editors and graphics, multimedia, and system applications. Check their websites at **gnome.org** and **kde.org** for latest developments. As new versions are released, they include new software.

Open Source Software

Linux was developed as a cooperative open source effort over the Internet, so no company or institution controls Linux. Software developed for Linux reflects this background. Development often takes place when Linux users decide to work on a project together. The software is posted at an Internet site, and any Linux user can then access the site and download the software. Linux software development has always operated in an Internet

environment and is global in scope, enlisting programmers from around the world. The only thing you need to start a Linux-based software project is a website.

Most Linux software is developed as open source software. This means that the source code for an application is freely distributed along with the application. Programmers over the Internet can make their own contributions to a software package's development, modifying and correcting the source code. Linux is an open source operating system as well. Its source code is included in all its distributions and is freely available on the Internet. Many major software development efforts are also open source projects, as are the KDE and GNOME desktops, along with most of their applications. The Netscape Communicator web browser package has also become open source, with its source code freely available. The OpenOffice office suite supported by Sun is an open source project based on the StarOffice office suite (StarOffice is essentially Sun's commercial version of OpenOffice). Many of the open source applications that run on Linux have located their websites at SourceForge (**sourceforge.net**), which is a hosting site designed specifically to support open source projects. You can find more information about the open source movement at **opensource.org**.

Open source software is protected by public licenses. These prevent commercial companies from taking control of open source software by adding a few modifications of their own, copyrighting those changes, and selling the software as their own product. The most popular public license is the GNU GPL provided by the Free Software Foundation. This is the license that Linux is distributed under. The GNU GPL retains the copyright, freely licensing the software with the requirement that the software and any modifications made to it always be freely available. Other public licenses have also been created to support the demands of different kinds of open source projects. The GNU lesser general public license (LGPL) lets commercial applications use GNU licensed software libraries. The qt public license (QPL) lets open source developer's use the Qt libraries essential to the KDE desktop. You can find a complete listing at **opensource.org**

Linux is currently copyrighted under a GNU public license provided by the Free Software Foundation, and it is often referred to as GNU software. GNU software is distributed free, provided it is freely distributed to others. GNU software has proved both reliable and effective. Many of the popular Linux utilities, such as C compilers, shells, and editors, are GNU software applications. Installed with your Linux distribution are the GNU C++ and

Lisp compilers, Vi and Emacs editors, BASH and TCSH shells, as well as TeX and Ghostscript document formatters. In addition, there are many open source software projects that are licensed under the GNU GPL.

Under the terms of the GNU GPL, the original author retains the copyright, although anyone can modify the software and redistribute it, provided the source code is included, made public, and provided free. Also, no restriction exists on selling the software or giving it away free. One distributor could charge for the software, while another one could provide it free of charge. Major software companies are also providing Linux versions of their most popular applications. Oracle provides a Linux version of its Oracle database. (At present, no plans seem in the works for Microsoft applications.)

Linux Software

All Linux software is currently available from online repositories. You can download applications for desktops, Internet servers, office suites, and programming packages, among others. Software packages may be distributed through online repositories. Downloads and updates are handled automatically by your desktop software manager and updater.

In addition, you can download from third-party sources software that is in the form of compressed archives or software packages like RPM and DEB. RPM packages are those archived using the Red Hat Package Manager, which is used on several distributions. Compressed archives have an extension such as **.tar.gz** or **.tar.Z**, whereas RPM packages have an **.rpm** extension and DEB uses a **.deb** extension. Any RPM package that you download directly, from whatever site, can be installed easily with the click of a button using a distribution software manager on a desktop. You can also download the source version and compile it directly on your system. This has become a simple process, almost as simple as installing the compiled RPM versions.

Linux distributions also have a large number of mirror sites from which you can download their software packages for current releases. If you have trouble connecting to a main FTP site, try one of its mirrors.

Software Repositories

For many distributions, you can update to the latest software from the online repositories using a software updater. Linux distributions provide a comprehensive selection of software ranging from office and multimedia applications to Internet servers and administration services. Many popular applications are not included, though they may be provided on associated software sites. During installation, your software installer is configured to access your distribution repository.

Because of licensing restrictions, multimedia support for popular formats like MP3, DVD, and DivX is not included with distributions. A distribution-associated site, however, may provide support for these functions, and from there you can download support for MP3, DVD, and DivX software. You can download a free licensed MP3 gstreamer plug-in. Nvidia- or ATI-released Linux graphics drivers, but support for these can be found at associated distribution sites. Linux distributions do include the generic X.org Nvidia and ATI drivers, which will enable your graphics cards to work.

Third-Party Linux Software Repositories

Though almost all applications should be included in the distribution software repository, you could download and install software from third-party repositories. Always check first to see if the software you want is already in the distribution repository. If it is not available, then download from a third-party repository.

Several third-party repositories make it easy to locate an application and find information about it. Of particular note are **sourceforge.net**, **rpmfind.net**, **gnomefiles.org**, and **kde-apps .org**. The following tables list different sites for Linux software. Some third-party repositories and archives for Linux software are listed in Table 1-2, along with several specialized sites, such as those for commercial and game software. When downloading software packages, always check to see if versions are packaged for your particular distribution.

Linux Office and Database Software

Many professional-level databases and office suites are now available for Linux.These include Oracle and IBM databases, as well as the OpenOffice and KOffice suites. Table 1-3 lists sites for office suites and databases. Most of the office suites, as well as MySQL and PostgreSQL, are already included on the distribution repositories and may be part of your install disk. Many of the other sites provide free personal versions of their software for Linux, and others are entirely free. You can download from them directly and install the software on your Linux system. URL Site

Internet Servers

One of the most important features of Linux, as of all Unix systems, is its set of Internet clients and servers. The Internet was designed and developed on Unix systems, and Internet clients and servers, such as those for FTP and the Web, were first implemented on BSD versions of Unix. DARPANET, the precursor to the Internet, was set up to link Unix systems at different universities across the nation. Linux contains a full set of

Internet clients and servers, including mail, news, FTP, and web, as well as proxy clients and servers.

Development Resources'

Linux has always provided strong support for programming languages and tools. All distributions include the GNU C and C++ (gcc) compiler with supporting tools such as make. Linux distributions usually come with full development support for the KDE and GNOME desktops, letting you create your own GNOME and KDE applications. You can also download the Linux version of the Java Software Development Kit for creating Java programs. A version of Perl for Linux is also included with most distributions. You can download current versions from their websites.

Online Linux Information Sources

Extensive online resources are available on almost any Linux topic. The tables in this chapter list sites where you can obtain software, display documentation, and read articles on the latest developments. Many Linux websites provide news, articles, and information about Linux. Several, such as linuxjournal.com, are based on popular Linux magazines. Some specialize in particular areas such as linuxgames.com for the latest games ported for Linux. Currently, many Linux websites provide news, information, and articles on Linux developments, as well as documentation, software links, and other resources

Linux Documentation

Linux documentation has also been developed over the Internet. Much of the documentation currently available for Linux can be downloaded from Internet FTP sites. A special Linux project called the Linux Documentation Project (LDP), headed by Matt Welsh, has developed a complete set of Linux manuals. The documentation is available at the LDP home site, tldp.org. Linux documents provided by the LDP are listed in Table 1-7, along with their Internet sites. The Linux documentation for your installed software will be available at your **/usr/share/doc** directory.

An extensive number of mirrors are maintained for the LDP. You can link to any of them through a variety of sources, such as the LDP home site, tldp.org, and linuxjournal.org. The documentation includes a user's guide, an introduction, and administrative guides.

These are available in text, PostScript, or web page format. You can also find briefer explanations in what are referred to as HOW-TO documents. Distribution websites provide extensive Linux documentation and software. The gnome .org site holds documentation for the GNOME desktop, while kde.org holds documentation for the KDE desktop.

Review & Self Assessment Question

Q1- What is operating system?
Q2- what is Linux operating system?
Q3- What do you mean by Linux Documentation?
Q4-Define the term “K Desktop Environment”

Further Readings

Linux Operating System Richard Petersen
Linux Operating System Paul S. Wang
Linux Operating System by David Maxwell and Andrew Bedford
Linux Operating System by Richard Blum and Christine Bresnahan
Linux Operating System by Bhatt P.C.P

UNIT: 2- GETTING STARTED WITH LINUX

Contents

Introduction

Using Linux has become an intuitive process, with easy-to-use interfaces, including graphical logins and graphical user interfaces (GUIs) like GNOME and KDE. Even the standard Linux command line interface has become more user friendly with editable commands, history lists, and cursor-based tools. Distribution installation tools also use simple GUIs. Installation has become a very easy procedure, taking only a few minutes. The use of online repositories by many distributions allows for small initial installs that can be later enhanced with selected additional software.

To start using Linux, you have to know how to access your Linux system and, once you are on the system, how to execute commands and run applications. Access is supported through either the default graphical login or a command line login. For the graphical login, a simple window appears with menus for selecting login options and text boxes for entering your username and password. Once you access your system, you can then interact with it using either a command line interface or a GUI. With GUIs like GNOME and KDE, you can use windows, menus, and icons to interact with your system.

Linux is noted for providing easy access to extensive help documentation. It's easy to obtain information quickly about any Linux command and utility while logged in to the system. You can access an online manual that describes each command or obtain help that provides

more detailed explanations of different Linux features. A complete set of manuals provided by the Linux Documentation Project (LDP) is on your system and available for you to browse through or print. Both the GNOME and KDE desktops provide help systems that give you easy access to desktop, system, and application help files.

Install Issues

Each distribution has its own graphical install tool that lets you install Linux very easily. Installation is often a simple matter of clicking a few buttons. However, install CDs and DVDs provide only a core subset of what is available because the software available has grown so massive that most distributions provide online repositories for downloading. Installation is now more a matter of setting up an initial configuration that you can later expand using these online repositories. Many distributions also allow you to create your own install discs, customizing the collection of software you want on your install CD/DVD. Other installation considerations include the following:

• Most distributions provide Live-CDs that allow you to do minimal installs. This helps you avoid a lengthy download of install CDs or DVDs. You can then install just the packages you want from online repositories.

• The use of online repositories means that most installed software needs to be downloaded and updated from the repositories soon after installation. The software on install CDs and DVDs quickly becomes out of date.

• Some distributions provide updated versions of a release, including updated software since the original release. These are often provided by separate distribution projects. Check the distribution sites for availability.

• Much of your hardware is now automatically detected, including your graphics card and monitor.

• Most distributions use parted to set up your partitions. Parted is a very easy-to-use partition management tool.

• Installation can be performed from numerous sources, by using network methods like NFS, File Transfer Protocol (FTP), and Hypertext Transfer Protocol (HTTP).

• Dual-boot installation is supported with either the GRUB or Linux Loader (LILO) boot managers. Linux boot managers can be configured easily to boot Windows, Mac, and other Linux installations on the same system.

• Distributions distinguish between 32-bit and 64-bit releases. Most CPUs in newer computers support 64-bit, whereas older or weaker systems may not.

• Network configuration is normally automatic, using Dynamic Host Configuration Protocol (DHCP) or IPv6 to connect to a network router.
• During installation you may have the option to customize your partitions, letting you set up RAID and LVM file systems if you wish.
• If you are using LVM or RAID file systems, be sure you have a separate boot partition of a standard Linux file system type.
• Most distributions perform a post-install procedure that perform basic configuration tasks like setting the date and time, configuring your firewall, and creating a user account (a root [administrative] account is set up during installation).

Most distributions provide a means to access your Linux system in rescue mode. Should your system stop working, you can access your files by using your install disc to start up Linux with a command line interface and access your installed file system. This allows you to fix your problem by editing or replacing configuration files.

If you have problems with the GRUB boot loader you can reinstall it with the **grub-install** command. This can happen if you later install Windows on your system. Windows will overwrite your boot manager. Use **grub-install** with the device name of the hard disk to reinstall the Linux boot manager. Be sure to put in an entry for your Windows system. Keep in mind that some distribution use alternative boot loaders like LILO.

Accessing Your Linux System

To access and use your Linux system, you must carefully follow required startup and shutdown procedures. You do not simply turn off your computer. Linux does, however, implement journaling, which allows you to automatically recover your system after the computer suddenly loses power and shuts off.

If you have installed the boot loader GRUB, when you turn on or reset your computer, the boot loader first decides what operating system to load and run. GRUB will display a menu of operating systems from which to choose.

If, instead, you wait a moment or press the ENTER key, the boot loader loads the default operating system. If a Windows system is listed, you can choose to start that instead.

You can think of your Linux operating system as operating on two different levels, one running on top of the other. The first level is when you start your Linux system and where the system loads and runs. It has control of your computer and all its peripherals. You still are not able to interact with it, however. After Linux starts, it displays a login screen, waiting for a user to log in to the system and start using it. You cannot gain access to

Linux unless you log in first. You can think of logging in and using Linux as the next level. Now you can issue commands instructing Linux to perform tasks. You can use utilities and programs such as editors or compilers, or even games. Depending on a choice you made during installation, however, you may be interacting with the system using either a simple command line interface or the desktop directly. There are both command line login prompts and graphical login windows. Most distributions will use a graphical interface by default, presenting you with a graphical login window at which you enter your username and password. If you choose not to use the graphical interface, you are presented with a simple command line prompt to enter your username.

The Display Managers: GDM and KDM

With the graphical login, your GUI starts up immediately and displays a login window with boxes for a username and password. When you enter your username and password and then press ENTER, your default GUI starts up.

For most distributions, graphical logins are handled either by the GNOME Display Manager (GDM) or the KDE Display Manager (KDM). The GDM and KDM manage the login interface along with authenticating a user password and username and then starting up a selected desktop. If problems ever occur using the GUI, you can force an exit of the GUI with the CTRL-ALT-BACKSPACE keys, returning to the login screen (or the command line if you started your GUI from there). Also, from the display manager, you can shift to the command line interface with the CTRL-ALT-F1 keys and then shift back to the GUI with the CTRL-ALT-F7 keys.

When you log out from the desktop, you return to the display manager Login window. From the Options menu, you can select the desktop or window manager you want to start up. Here you can select KDE to start up the K Desktop, for example, instead of GNOME. The Language menu lists a variety of different languages that Linux supports. Choose one to change the language interface.

To shut down your Linux system, click the Shutdown button. To restart, select the Restart option from the Options menu. Alternatively, you can also shut down or restart from your desktop. From the System menu, select the Shutdown entry. GNOME will display a dialog screen with the buttons Suspend, Shutdown, and Reboot. Shutdown is the default and will occur automatically after a few seconds. Selecting Reboot will shut down and restart your system. KDE will prompt you to end a session, shutdown, or logout. (You can also open a Terminal window and enter the **shutdown**,

halt, or **reboot** command, as described later; **halt** will log out and shut down your system.)

Switching Users

Once you have logged in to your desktop, you can switch to different user without having to log out or end your current user session. On GNOME you use the User Switcher tool, a GNOME applet on the panel. For KDE you use the Switch User entry on the Main menu.

User Switcher: GNOME

On GNOME, the switcher will appear on the panel as the name of the currently logged-in user. If you left-click the name, a list of all other users will be displayed. Check boxes next to each show which users are logged in and running. To switch a user, select the user from this menu. If the user is not already logged in, the login manager (the GDM) will appear and you can enter that user's password. If the user is already logged in, then the Login window for the lock screen will appear (you can disable the lock screen). Just enter the user's password. The user's original session will continue with the same open windows and applications running as when the user switched off. You can easily switch back and forth between logged in users, with all users retaining their session from where they left off. When you switch off from a user, that user's running programs will continue in the background.

Right-clicking the switcher will list several user management items, such as configuring the login screen, managing users, or changing the user's password and personal information. The Preferences item lets you configure how the User Switcher is displayed on your panel. Instead of the user's name, you could use the term Users or a user icon. You can also choose whether to use a lock screen when the user switches. Disabling the lock screen option will let you switch seamlessly between logged-in users.

Switch User: KDE

On KDE, the Switch User entry on the Main menu will display a list of users you can change to. You can also elect to start a different session, hiding your current one. In effect this lets you start up your desktop again as the same user. You can also lock the current session before starting a new one. New sessions can be referenced starting with the F7 key for the first session. Use CTRL-ALT-F7 to access the first session and CTRL-ALT-F8 for the second session.

Accessing Linux from the Command Line Interface

For the command line interface, you are initially given a login prompt. The system is now running and waiting for a user to log in and use it. You can

enter your username and password to use the system. The login prompt is preceded by the hostname you gave your system. In this example, the hostname is **turtle**. When you finish using Linux, you first log out. Linux then displays exactly the same login prompt, waiting for you or another user to log in again. This is the equivalent of the Login window provided by the GDM. You can then log in to another account.

Logging In and Out with the Command Line

Once you log in to an account, you can enter and execute commands. Logging in to your Linux account involves two steps: entering your username and then entering your password. Type in the username for your user account. If you make a mistake, you can erase characters with the BACKSPACE key. In the next example, the user enters the username **richlp** and is then prompted to enter the password:

Password:

When you type in your password, it does not appear on the screen. This is to protect your password from being seen by others. If you enter either the username or the password incorrectly, the system will respond with the error message "Login incorrect" and will ask for your username again, starting the login process over. You can then reenter your username and password.

Once you enter your username and password correctly, you are logged in to the system. Your command line prompt is displayed, waiting for you to enter a command. Notice the command line prompt is a dollar sign (**$**), not a number sign (**#**). The **$** is the prompt for regular users, whereas the **#** is the prompt solely for the root user. In this version of Linux, your prompt is preceded by the hostname and the directory you are in. Both are bounded by a set of brackets.

```
[turtle /home/richlp]$
```

To end your session, issue the **logout** or **exit** command. This returns you to the login prompt, and Linux waits for another user to log in:

```
[turtle /home/richlp]$ logout
```

Shutting Down Linux from the Command Line

If you want to turn off your computer, you must first shut down Linux. Not shutting down Linux may require Linux to perform a lengthy systems check when it starts up again. You shut down your system in either of two ways. First log in to an account and then enter the **halt** command. This command will log you out and shut down the system.

```
$ halt
```

Alternatively, you can use the **shutdown** command with the **-h** option. Or, with the **-r** option, the system shuts down and then reboots. In the next

example, the system is shut down after five minutes. To shut down the system immediately, you can use **+0** or the word **now**.

```
# shutdown -h now
```

You can also force your system to reboot at the login prompt by holding down the CTRL and ALT keys and then pressing the DEL key (CTRL-ALT-DEL). Your system will go through the standard shutdown procedure and then reboot your computer.

The GNOME and KDE Desktops

Two alternative desktop GUIs can be installed on most Linux systems: GNOME and KDE. Each has its own style and appearance. GNOME uses the Clearlooks theme for its interface with the distribution screen background and menu icon as its default.

It is important to keep in mind that though the GNOME and KDE interfaces appear similar, they are really two very different desktop interfaces with separate tools for selecting preferences. The Preferences menus on GNOME and KDE display very different selections of desktop configuration tools.

Though GNOME and KDE are wholly integrated desktops, they in fact interact with the operating system through a window manager—Metacity in the case of GNOME and the KDE window manager for KDE. You can use a different GNOME- or KDE-compliant window manager if you wish, or simply use a window manager in place of either KDE or GNOME.

KDE

The K Desktop Environment (KDE) displays a panel at the bottom of the screen that looks very similar to one displayed on the top of the GNOME desktop. The file manager appears slightly different but operates much the same way as the GNOME file manager. There is a Control Center entry in the Main menu that opens the KDE control center, from which you can configure every aspect of KDE, such as themes, panels, peripherals like printers and keyboards, even the KDE file manager's web browsing capabilities.

XFce4

The XFce4 desktop is a new lightweight desktop designed to run fast without the kind of overhead seen in full-featured desktops like KDE and GNOME. It includes its own file manager and panel, but the emphasis is on modularity and simplicity. The desktop consists of a collection of modules, including the xffm file manager, the xfce4-panel panel, and the xfwm4 window manager. In keeping with its focus on simplicity, its small

scale makes it appropriate for laptops or dedicated systems that have no need for the complex overhead found in other desktops.

GNOME

The GNOME desktop display shows three menus: Applications, Places, and System. The Places menu lets you easily access commonly used locations like your home directory, the desktop folder for any files on your desktop, and the Computer window, through which you can access devices, shared file systems, and all the directories on your local system. The System menu includes Preferences and Administration menus. The Preferences menu is used for configuring your GNOME settings, such as the theme you want to use and the behavior of your mouse.

To move a window, left-click and drag its title bars. Each window supports Maximize, Minimize, and Close buttons. Double-clicking the title bar will maximize the window. Each window will have a corresponding button on the bottom panel. You can use this button to minimize and restore the window. The desktop supports full drag-and-drop capabilities. You can drag folders, icons, and applications to the desktop or other file manager windows open to other folders. The move operation is the default drag operation. To copy files, press the CTRL key and then click and drag before releasing the mouse button. To create a link, hold both the CTRL and SHIFT keys while dragging the icon to the location where you want the link, such as the desktop.

GNOME provides several tools for configuring your desktop. These are listed in the System | Preferences menu. Configuration preference tools are organized into several submenus: Personal, Look and Feel, Internet and Network, Hardware, and System. Those that do not fall into any category are listed directly. Several are discussed in different sections in this and other chapters. The Help button on each preference window will display detailed descriptions and examples. Some of the more important tools are discussed here.

The Keyboard Shortcuts configuration (Personal | Keyboard Shortcuts) lets you map keys to certain tasks, for example, mapping multimedia keys on a keyboard to media tasks such as play and pause. The File Management configuration (Personal | File Management) lets you determine the way files and directories are displayed, along with added information to show in icon captions or list views. The Windows configuration (Look and Feel | Windows) is where you can enable features like window roll-up, window movement key, and mouse window selection.

The Mouse and Keyboard preferences are the primary tools for configuring your mouse and keyboard (Hardware | Keyboard and

Hardware | Mouse). The Mouse preferences let you choose a mouse image and configure its motion and hand orientation. The Keyboard preferences window shows several panels for selecting your keyboard model (Layout), configuring keys (Layout Options) and repeat delay (Keyboard), and even enforcing breaks from power typing as a health precaution.

GNOME and KDE Applets

GNOME applets are small programs that operate off your panel. It is very easy to add applets. Right-click the panel and select the Add entry. This lists all available applets. Some helpful applets are dictionary lookup; the current weather; the system monitor, which shows your CPU usage; the CPU Frequency Scaling Monitor for Cool and Quiet processors; and Search, which searches your system for files, as well as Lock, Shutdown, and Logout buttons. Some of these, including Find, Lock, and Logout, are already on the Places menu. You can drag these directly from the menu to the panel to add the applet. Following the web browser and email icons, you have, from left to right: Search for files, dictionary lookup, Tomboy note taker, Network connection monitor, CPU scaling monitor, System Monitor, Weather report, Eyes that follow your mouse around, User Switcher, and the Logout, Shutdown, and Lock Screen buttons.

On KDE, right-click the panel and select Add Applet to Panel. From the KDE applets window, you can select similar applets such as System Monitor and Sound Mixer.

Starting a GUI from the Command Line

Once logged in to the system from the command line, you still have the option of starting an X Window System GUI, such as GNOME or KDE. In Linux, the command **startx** starts a desktop. The **startx** command starts the GNOME desktop by default. Once you shut down the desktop, you will return to your command line interface, still logged in.

```
$ startx
```

Desktop Operations

There are several desktop operations that you may want to take advantage of when first setting up your desktop. These include selecting themes, setting your font sizes larger for high resolution monitors, burning CD/DVD discs, searching your desktop for files, using removable media like USB drives, and accessing remote hosts.

Desktop Themes

On GNOME, you use the Themes Preferences tool to select or customize a theme. Themes control your desktop appearance. When you open the Theme tool, a list of currently installed themes is shown. The GNOME

theme is initially selected. You can move down the list to select a different theme if you wish.

The true power of Themes is in the ability it provides users to customize any given theme. Themes are organized into three components: controls, window border, and icons. Controls cover the appearance of window and dialog controls such as buttons and slider bars. Window border specifies how title bars, borders, and window buttons are displayed. Icons specify how all icons used on the desktop are displayed, whether on the file manager, desktop, or the panel. You can mix and match components from any installed theme to make your own theme. You can even download and install separate components like specific icon sets, which you can then use in a customized theme.

Clicking the Customize button will open a Themes Details window with panels of the different theme components. The ones used for the current theme will be already selected. In the control, window border, and icon panels you will see listings of the different installed themes. An additional Color panel lets you set the background and text colors for windows, input boxes, and selected items. You can then mix and match different components like icons, window styles, and controls, creating your own customized theme. Upon selecting a component, your desktop automatically changes, showing you how it looks.

One you have created a new customized theme, a Custom Theme entry will appear in the theme list. To save the customized theme, click the Save Theme button. This opens a dialog where you can enter a theme name, any notes, and specify whether you want to also keep the theme background. The saved theme then appears in the theme listing.

On KDE, open the Theme manager in the KDE Control Center under Appearances and Themes. Select the theme you want from the Theme panel. The selected theme will be displayed on the facing panel. Buttons in the Customize section let you build a customized theme, selecting background, icons, colors, styles, fonts, and even screensavers. To download new themes, click the Get new themes link in the upper right corner. This opens the Kde-look web page for KDE themes. You will have to download themes, extract them, and then click the Install theme button, locating and selecting the downloaded theme's **.kth** file. This method works only for themes in the Theme manager format, **kth**. Themes not in this format have to be installed manually.

GNOME themes and icons installed directly by a user are placed in the **.themes** and **.icons** directories in the user's home directory. Should you want these themes made available for all users, you can move them from

the **.themes** and **.icons** directories to the **/usr/share/ icons** and **/usr/share/themes** directories. Be sure to log in as the root user. You then need to change ownership of the moved themes and icons to the root user:

chown -R root:root /usr/share/themes/newtheme

User KDE themes are placed in the **.kde/share/apps/kthememanager** directory.

Fonts

Most distributions now use the fontconfig method for managing fonts. You can easily change font sizes, add new fonts, and configure features like anti-aliasing. Both GNOME and KDE provide tools for selecting, resizing, and adding fonts.

Resizing Desktop Fonts

With very large monitors and their high resolutions becoming more common, one feature users find helpful is the ability to increase the desktop font sizes. On a large widescreen monitor, resolutions less than the native one tend not to scale well. A monitor always looks best in its native resolution. However, with a large native resolution like 1900 × 1200, text sizes become so small they are hard to read. You can overcome this issue by increasing the font size. Use the font tools on your desktop to change these sizes (System | Preferences | Look And Feel | Fonts on GNOME; for KDE, select the Fonts entry in the Control Center's Appearance and Themes).

Adding Fonts

To add a new font (for both GNOME and KDE), just enter the **fonts:/** URL in a file manager window. This opens the font window. Drag and drop your font file to it. When you restart, your font will be available for use on your desktop. KDE will have Personal and System folders for fonts, initially showing icons for each. For user fonts, open the Personal Fonts window. Fonts that are Zip archived, should first be opened with the Archive manager and then can be dragged from the archive manager to the font viewer. To remove a font, right-click it in the font viewer and select Move to Trash or Delete.

User fonts will be installed to a user's **.fonts** directory. For fonts to be available to all users, they have to be installed in the **/usr/share/fonts** directory, making them system fonts. On KDE, you do this by opening the System folder, instead of the Personal folder, when you start up the fonts viewer. You can do this from any user login. Then drag any font packages to this **fonts:/System** window. On GNOME, you have to log in as the root user and manually copy fonts to the **/usr/share/fonts** directory. If your system has both GNOME and KDE installed, you can install system fonts

using KDE (Konqueror file manager), and they will be available on GNOME.

To provide speedy access to system fonts, you should create font information cache files for the **/usr/share/fonts** directory. To do this, run the **fc-cache** command as the root user.

Configuring Fonts

On GNOME, to better refine your font display, you can use the font rendering tool. Open the Font Preferences tool (System | Preferences | Look and Feel | Fonts). In the Font Rendering section are basic font rendering features like Monochrome, Best contrast, Best shapes, and Subpixel smoothing. Choose the one that works best. For LCDS, choose Subpixel smoothing. For detailed configuration, click the Details button. Here you can set smoothing, hinting (anti-aliasing), and subpixel color order features. The subpixel color order is hardware dependent. On KDE, in the KDE control center, select the Fonts entry under Appearance and Themes. Click the Use anti-aliasing for fonts check box, and then click the Configure button to open a window to let you select hinting and subpixel options.

On GNOME, clicking a font entry in the Fonts Preferences tool will open a Pick a Font dialog that will list all available fonts. On KDE, clicking any of the Choose buttons on the Control Center's Fonts panel will also open a window listing all available fonts. You can also generate a listing with the **fc-list** command. The list will be unsorted, so you should pipe it first to the sort command. You can use **fc-list** with any font name or name pattern to search for fonts, with options to search by language, family, or styles. See the **/etc/share/ fontconfig** documentation for more details.

fc-list | sort

Configuring Your Personal Information

On GNOME, the About Me preferences dialog lets you set up personal information to be used with your desktop applications, as well as change your password. Clicking the Image icon in the top left corner opens a browser window where you can select the image to use. The Faces directory is selected by default with images you can use. The selected image is displayed to the right in the browser window. For a personal photograph, you can use the Pictures folder. This is the Pictures folder in your home directory. Should you place a photograph or image there, you can then select if for your personal image. The image will be used in the Login screen when showing your user entry. Should you want to change your password, you can click the Change password button at the top right.

There are three panels: Contact, Address, and Personal Info. On the Contact panel you enter email (home and work), telephone, and instant messaging addresses. On the Address panel you enter your home and work addresses, and on the Personal Info panel you list your web addresses and work information.

On KDE, you can select the Password panel in the Security entry on the KDE Control Center. Here you can select a picture for your account. Contact information is handled by other applications, like Kontact for mail and user information.

Sessions

You can configure your desktop to restore your previously opened windows and applications, as well as specify startup programs. When you log out, you may want the windows you have open and the applications you have running to be automatically started when you log back in. In effect, you are saving your current session and having it restored it when you log back in. For example, if you are in the middle of working on a spreadsheet, you can save your work but not close the file. Then log out. When you log back in, your spreadsheet will be opened automatically to where you left off.

For GNOME, saving sessions is not turned on by default. You use the Sessions preferences dialog's Session Options panel (System | Preferences | Personal | Sessions) to save sessions. You can save your current session manually or opt to have all your sessions saved automatically when you log out, restoring them whenever you log in.

On KDE you can configure your session manager by selecting Sessions from the KDE Components entry in the Control Center. By default, the previous session is restored when you log in. You can also determine default shutdown behavior.

Using Removable Devices and Media

Linux desktops now support removable devices and media such as digital cameras, PDAs, card readers, and even USB printers. These devices are handled automatically with an appropriate device interface set up on the fly when needed. Such hotplugged devices are identified, and where appropriate, their icons will appear in the file manager window. For example, when you connect a USB drive to your system, it will be detected and displayed as storage device with its own file system.

Installing Multimedia Support: MP3, DVD, and DivX

Because of licensing and other restrictions, many Linux distributions do not include MP3, DVD, or DivX media support in their free versions. You have to purchase their commercial versions, which include the appropriate

licenses for this support. Alternatively, you can obtain this support from independent operations such as those at **fluendo.com**. DivX support can be obtained from **labs.divx.com/DivXLinuxCodec**. Check the multimedia information pages at your distribution website for more information.

Command Line Interface

When using the command line interface, you are given a simple prompt at which you type in your command. Even with a GUI, you sometimes need to execute commands on a command line. The Terminal window is no longer available on the GNOME desktop menu. You now have to access it from the Applications | System Tools menu. If you use Terminal windows frequently, you may want to just drag the menu entry to the desktop to create a desktop icon for the Terminal window. Just click to open.

Linux commands make extensive use of options and arguments. Be careful to place your arguments and options in their correct order on the command line. The format for a Linux command is the command name followed by options, and then by arguments, as shown here:

$ command-name options arguments

An option is a one-letter code preceded by one or two hyphens, which modifies the type of action the command takes. Options and arguments may or may not be optional, depending on the command. For example, the **ls** command can take an option, **-s**. The **ls** command displays a listing of files in your directory, and the **-s** option adds the size of each file in blocks. You enter the command and its option on the command line as follows:

$ ls -s

An argument is data the command may need to execute its task. In many cases, this is a filename. An argument is entered as a word on the command line after any options. For example, to display the contents of a file, you can use the **more** command with the file's name as its argument. The **less** or **more** command used with the filename **mydata** would be entered on the command line as follows:

$ less mydata

The command line is actually a buffer of text you can edit. Before you press ENTER, you can perform editing commands on the existing text. The editing capabilities provide a way to correct mistakes you may make when typing in a command and its options. The BACKSPACE and DEL keys let you erase the character you just typed in. With this character-erasing capability, you can BACKSPACE over the entire line if you want, erasing what you entered. CTRL-U erases the whole line and enables you to start over again at the prompt.

Help Resources

A great deal of support documentation is already installed on your system and is also accessible from online sources. Table 2-1 lists Help tools and resources accessible on most Linux systems. Both the GNOME and KDE desktops feature Help systems that use a browser-like interface to display help files. To start the GNOME or KDE Help browser, select the Help entry in the main menu. You can then choose from the respective desktop user guides, including the KDE manual, Linux Man pages, and GNU info pages. The GNOME Help Browser also accesses documents for GNOME applications such as the File Roller archive tool and Evolution mail client. The GNOME Help browser and the KDE Help Center also incorporate browser capabilities, including bookmarks and history lists for documents you view.

Resource	Description
KDE Help Center	KDE Help tool, GUI for documentation on KDE desktop and applications, Man pages, and Info documents
GNOME Help Browser	GNOME Help tool, GUI for accessing documentation for the GNOME desktop and applications, Man pages, and Info documents
/usr/share/doc	Location of application documentation
man *command*	Linux Man pages, detailed information on Linux commands, including syntax and options
info *application*	GNU Info pages, documentation on GNU applications

Context-Sensitive Help

Both GNOME and KDE, along with applications, provide context-sensitive help. Each KDE and GNOME application features detailed manuals that are displayed using their respective Help browsers. Also, system administrative tools feature detailed explanations for each task.

Application Documentation

On your system, the **/usr/share/doc** directory contains documentation files installed by each application. Within each directory, you can usually find HOW-TO, README, and INSTALL documents for that application.

The Man Pages

You can also access the Man pages, which are manuals for Linux commands available from the command line interface, using the **man** command. Enter **man** with the command for which you want information. The following example asks for information on the **ls** command:

```
$ man ls
```

Pressing the SPACEBAR key advances you to the next page. Pressing the B key moves you back a page. When you finish, press the Q key to quit the Man utility and return to the command line. You activate a search by pressing either the slash (/) or question mark (?). The / searches forward; the ? searches backward. When you press the /, a line opens at the

bottom of your screen, and you then enter a word to search for. Press ENTER to activate the search. You can repeat the same search by pressing the N key. You needn't reenter the pattern.

The Info Pages

Online documentation for GNU applications, such as the GNU C and C++ compiler (gcc) and the Emacs editor, also exist as info pages accessible from the GNOME and KDE Help Centers. You can also access this documentation by entering the command **info.** This brings up a special screen listing different GNU applications. The info interface has its own set of commands. You can learn more about it by entering **info info**. Typing **m** opens a line at the bottom of the screen where you can enter the first few letters of the application. Pressing ENTER brings up the info file on that application.

Software Repositories

For most Linux distributions, software has grown so large and undergoes such frequent updates that it no longer makes sense to use discs as the primary means of distribution. Instead, distribution is effected using an online software repository. This repository contains an extensive collection of distribution-compliant software.

This entire approach heralds a move from thinking of most Linux software as something included on a few discs to viewing the disc as a core from which you can expand your installed software as you like from online repositories. Most software is now located at the Internet-connected repositories. You can now think of that software as an easily installed extension of your current collection. Relying on disc media for your software has become, in a sense, obsolete.

Windows Access and Applications

In many cases, certain accommodations need to be made for Windows systems. Most Linux systems are part of networks that also run Windows systems. Using Linux Samba servers, your Linux and Windows systems can share directories and printers. In addition, you may also need to run a Windows application directly on your Linux system. Though there is an enormous amount of Linux software available, in some cases you may need or prefer to run a Windows application. The Wine compatibility layer allows you to do just that for many Windows applications (but not all).

Setting up Windows Network Access: Samba

Most local and home networks may include some systems working on Microsoft Windows and others on Linux. You may need to let a Windows computer access a Linux system or vice versa. Windows, because of its

massive market presence, tends to benefit from both drivers and applications support not found for Linux. Though there are equivalent applications on Linux, many of which are as good or better, some applications run best on Windows, if for no other reason than that the vendor only develops drivers for Windows.

One solution is to use the superior server and storage capabilities of Linux to manage and hold data, while using Windows systems with their unique applications and drivers to run applications. For example, you can use a Linux system to hold pictures and videos, while using Windows systems to show or run them. Video or pictures can be streamed through your router to the system that wants to run them. In fact, many commercial DVR systems use a version of Linux to manage video recording and storage. Another use would be to enable Windows systems to use devices like printers that may be connected to a Linux system, or vice versa.

To allow Windows to access a Linux system and Linux to access a Windows system, you use the Samba server. Samba has two methods of authentication, shares and users, though the shares method has been deprecated. User authentication requires that there be corresponding accounts in the Windows and Linux systems. You need to set up a Samba user with a Samba password. The Samba user should be the same name as an established account. The Windows user and Samba user can have the same name, though a Windows user can be mapped to a Samba user. A share can be made open to specific users and function as an extension of the user's storage space. On most current distributions, Samba user and password information are kept in tdb (trivial data base) Samba database files, which can be edited and added to using the **pdbedit** command.

To set up simple file sharing on a Linux system, you first need to configure your Samba server. You can do this by directly editing the **/etc/samba/samba.conf** file. If you just edit the **/etc/samba/samba.conf** file, you first need to specify the name of your Windows network. Samba provides a configuration tool called SWAT that you can use with any browser to configurae your Samba server, adding users and setting up shares. Some distributions, like Ubuntu, set up Samba automatically. KDE also provides Samba configuration.

Once set up, both GNOME and KDE allow you to browse and access Samba shares from your desktop, letting you also access shared Windows directories and printers on other systems. On GNOME click the Network and then the Windows Network icon on the My Computer window. You will see an icon for your Windows network. On either

GNOME or KDE you can enter the **smb:** URL in the a file manager window to access your Windows networks.

When a Windows user wants to access the share on the Linux system, they open their My Network Places (Network on Vista) and then select Add a network place to add a network place entry for the share, or View workgroup computers to see computers on your Windows network. Selecting the Linux Samba server will display your Samba shares. To access the share, the user will be required to enter the Samba username and the Samba password. You have the option of having the username and password remembered for automatic access.

Running Windows Software on Linux: Wine

Wine is a Windows compatibility layer that will allow you to run many Windows applications natively on Linux. Though you could run the Windows operating system on it, the actual Windows operating system is not required. Windows applications will run as if they were Linux applications, able to access the entire Linux file system and use Linux-connected devices. Applications that are heavily driver dependent, such as graphic-intensive games, most likely will not run. Others, such as newsreaders, which do not rely on any specialized drivers, may run very well. For some applications, you may also need to copy over specific Windows DLLs from a working Windows system to your Wine Windows **system32** or **system** directory.

To install Wine on your system, search for **wine** on you distributions repositories. For some distributions you may have to download wine directly from **winehq.org.** Binaries for several distributions are provided.

Once installed, a Wine menu will appear in the Applications menu. The Wine menu holds entries for Wine configuration, the Wine software uninstaller, and the Wine file browser, as well as a regedit registry editor, a notepad, and a Wine help tool. ‘

To set up Wine, a user starts the Wine Configuration tool. This opens a window with panels for Applications, Libraries (DLL selection), Audio (sound drivers), Drives, Desktop Integration, and Graphics. On the Applications panel you can select which version of Windows an application is designed for. The Drives panel will list your detected partitions, as well as your Windows-emulated drives, such as drive C:. The C: drive is really just a directory, **.wine/drive_c**, not a partition of a fixed size. Your actual Linux file system will be listed as the Z: drive.

Once configured, Wine will set up a **.wine** directory on the user’s home directory (the directory is hidden, so enable Show Hidden Files in the file

browser View menu to display it). Within that directory will be the **drive_c** directory, which functions as the C: drive, holding your Windows system files and program files in the **Window**s and **Program File** subdirectories. The **System and System32** directories are located in the **Windows** directory. Here is where you place any needed DLL files. The **Program Files** directory will hold your installed Windows programs, just as they would be installed on a Windows **Program Files** directory.

To install a Windows application with Wine, you can either use the Wine configuration tool or open a Terminal window and run the **wine** command with the Windows application as an argument. The following example installs the popular newsbin program:\

$ wine newsbin.exe

To install with the Windows Configuration tool, select the Applications panel and then click Add.

Some applications, such as newsbin, will also require that you use certain DLL files from a working Windows operating system. The DLL files are normally copied to the user's **.wine/drive_c/Windows/system32** directory.

Icons for installed Windows software will appear on your desktop. Just double-click an icon to start up the application. It will run normally within a Linux window, as would any Linux application.

Installing Windows fonts on Wine is a simple matter of copying fonts from a Windows font directory to your Wine **.wine/drive_c/Windows/fonts** directory. You can just copy any Windows **.ttf** file to this directory to install a font. You can also use the Microsoft common web fonts available from **fontconfig.org**.

Wine will use a stripped-down window style for features like buttons and the title bar. If you want to use the XP style, download and install the Royal theme from Microsoft. Keep in mind, however, that supporting this theme is very resource intensive and will likely slow down your system.

Review & Self Assessment Question

Q1- What do you mean by GDM /KDM?

Q2- What do you mean by GNOME?

Q3-What do you mean by session?

Q4-What is Comman Line Interface?

Further Readings

Linux Operating System Richard Petersen

Linux Operating System Paul S. Wang

Linux Operating System by David Maxwell and Andrew Bedford

UNIT: 3- THE SHELL

Contents

Shell

The shell is a command interpreter that provides a line-oriented interactive and non interactive interface between the user and the operating system. You enter commands on a command line; they are interpreted by the shell and then sent as instructions to the operating system (the command line interface is accessible from GNOME and KDE through a Terminal windows—Applications/Accessories menu). You can also place commands in a script file to be consecutively executed, much like a program. This interpretive capability of the shell provides for many sophisticated features. For example, the shell has a set of file matching characters that can generate filenames. The shell can redirect input and output, as well as run operations in the background, freeing you to perform other tasks.

Several different types of shells have been developed for Linux: the Bourne Again shell (BASH), the Korn shell, the TCSH shell, and the Z shell. TCSH is an enhanced version of the C shell used on many Unix systems, especially BSD versions. You need only one type of shell to do your work. Linux includes all the major shells, although it installs and uses the BASH shell as the default. If you use the command line shell, you will be using the BASH shell unless you specify another. This chapter primarily discusses the BASH shell, which shares many of the same

features as other shells. A brief discussion of the C shell, TCSH, and the Z shell follows at the end of the chapter, noting differences.

You can find out more about shells at their respective websites, as listed in Table 3-1. Also, a detailed online manual is available for each installed shell. Use the **man** command and the shell's keyword to access them, **bash** for the BASH shell, **ksh** for the Korn shell, **zsh** for the Z shell, and **tsch** for the TSCH shell. For the C shell you can use **csh**, which links to **tcsh**. For example, the command **man bash** will access the BASH shell online manual.

The Command Line

The Linux command line interface consists of a single line into which you enter commands with any of their options and arguments. From GNOME or KDE, you can access the command line interface by opening a terminal window. Should you start Linux with the command line interface, you will be presented with a BASH shell command line when you log in.

By default, the BASH shell has a dollar sign (**$**) prompt, but Linux has several other types of shells, each with its own prompt (**%** for the C shell, for example). The root user will have a different prompt, the #. A shell prompt, such as the one shown here, marks the beginning of the command line:

$

You can enter a command along with options and arguments at the prompt. For example, with an **-l** option, the **ls** command will display a line of information about each file, listing such data as its size and the date and time it was last modified. The dash before the **-l** option is required. Linux uses it to distinguish an option from an argument.

$ ls -l

If you want the information displayed only for a particular file, you can add that file's name as the argument, following the **-l** option:

$ **ls -l mydata**

-rw-r--r-- 1 chris weather 207 Feb 20 11:55 mydata

You can enter a command on several lines by typing a backslash just before you press ENTER. The backslash "escapes" the ENTER key, effectively continuing the same command line to the next line. In the next example, the **cp** command is entered on three lines:

$ **cp -1**

**mydata **

/home/george/myproject/newdata

You can also enter several commands on the same line by separating them with a semicolon (**;**). In effect the semicolon operates as an execute

operation. Commands will be executed in the sequence they are entered. The following command executes an **ls** command followed by a **date** command.

$ ls ; date

You can also conditionally run several commands on the same line with the **&&** operator. A command is executed only if the previous one is true. This feature is useful for running several dependent scripts on the same line. In the next example, the **ls** command runs only if the **date** command is successfully executed.

$ date && ls

Command Line Editing

The BASH shell, which is your default shell, has special command line editing capabilities that you may find helpful as you learn Linux . You can easily modify commands you have entered before executing them, moving anywhere on the command line and inserting or deleting characters. This is particularly helpful for complex commands. You can use the CTRL-F or RIGHT ARROW key to move forward a character or the CTRL-B or LEFT ARROW key to move back a character. CTRL-D or DEL deletes the character the cursor is on, and CTRL-H or BACKSPACE deletes the character before the cursor. To add text, you use the arrow keys to move the cursor to where you want to insert text and type the new characters. You can even cut words with the CTRL-W or ALT-D key and then use the CTRL-Y key to paste them back in at a different position, effectively moving the words. As a rule, the CTRL version of the command operates on characters, and the ALT version works on words, such as CTRL-T to transpose characters and ALT-T to transpose words. At any time, you can press ENTER to execute the command. The actual associations of keys and their tasks, along with global settings, are specified in the **/etc/inputrc** file.

Movement Commands	Operation
CTRL-F, RIGHT-ARROW	Move forward a character.
CTRL-B, LEFT-ARROW	Move backward a character.
CTRL-A or HOME	Move to beginning of line.
CTRL-E or END	Move to end of line.
ALT-F	Move forward a word.
ALT-B	Move backward a word.
CTRL-L	Clear screen and place line at top.

Editing Commands	Operation
CTRL-D or DEL	Delete character cursor is on.
CTRL-H or BACKSPACE	Delete character before the cursor.
CTRL-K	Cut remainder of line from cursor position.
CTRL-U	Cut from cursor position to beginning of line.
CTRL-W	Cut previous word.
CTRL-C	Cut entire line.
ALT-D	Cut the remainder of a word.
ALT-DEL	Cut from the cursor to the beginning of a word.
CTRL-Y	Paste previous cut text.
ALT-Y	Paste from set of previously cut text.
CTRL-Y	Paste previous cut text.
CTRL-V	Insert quoted text, used for inserting control or meta (ALT) keys as text, such as CTRL-B for backspace or CTRL-T for tabs.
ALT-T	Transpose current and previous word.
ALT-L	Lowercase current word.
ALT-U	Uppercase current word.
ALT-C	Capitalize current word.
CTRL-SHIFT-_	Undo previous change.

Command and Filename Completion

The BASH command line has a built-in feature that performs command line and filename completion. Automatic completions can be effected using the TAB key. If you enter an incomplete pattern as a command or filename argument, you can then press the TAB key to activate the command and filename completion feature, which completes the pattern.

Directories will have attached to their name .If more than one command or files has the same prefix, the shell simply beeps and waits for you to enter the TAB key again. It then displays a list of possible command completions and waits for you to add enough characters to select a unique command or filename. In situations where you know there are likely multiple possibilities, you can just press the ESC key instead of two TABs. In the next example, the user issues a **cat** command with an incomplete filename. When the user presses the TAB key, the system searches for a match and, when it finds one, fills in the filename. The user can then press ENTER to execute the command.

$ cat pre tab

$ cat preface

Automatic completion also works with the names of variables, users, and hosts. In this case, the partial text needs to be preceded by a special character indicating the type of name. Variables begin with a $ sign, so any text beginning with a $ sign is treated as a variable to be completed. Variables are selected from previously defined variables, like system shell variables (see Chapter 4). Usernames begin with a tilde (~). Host names

begin with an @ sign, with possible names taken from the **/etc/hosts** file. A listing of possible automatic completions follows:

• Filenames begin with any text or /.
• Shell variable text begins with a $ sign.
• Username text begins with a ~ sign.
• Host name text begins with a @.
• Commands, aliases, and text in files begin with normal text.

For example, to complete the variable HOME given just $HOM, simply enter a TAB character.

$ **echo $HOM** <tab>
$ echo $HOME

If you enter just an H, then you can enter two tabs to see all possible variables beginning with H. The command line will be redisplayed, letting you complete the name.

$ **echo $H** <tab> <tab>
$HISTCMD $HISTFILE $HOME $HOSTTYPE HISTFILE $HISTSIZE $HISTNAME
$ **echo $H**

You can also specifically select the kind of text to complete, using corresponding command keys. In this case, it does not matter what kind of sign a name begins with. For example, the ALT-~ will treat the current text as a username. ALT-@ will treat it as a host name and ALT-$, as a variable. ALT-! will treat it as a command. To display a list of possible completions, use the CTRL-X key with the appropriate completion key, as in CTRL –x-$ to list possible variable completion.

Command (CTRL-R for Listing Possible Completions)	Description
TAB	Automatic completion
TAB TAB or ESC	List possible completions
ALT-/, CTRL-R-/	Filename completion, normal text for automatic
ALT-$, CTRL-R-$	Shell variable completion, $ for automatic
ALT-~, CTRL-R-~	Username completion, ~ for automatic
ALT-@, CTRL-R-@	Host name completion, @ for automatic
ALT-!, CTRL-R-!	Command name completion, normal text for automatic

History

The BASH shell keeps a list, called a history list, of your previously entered commands. You can display each command, in turn, on your command line by pressing the UP ARROW key. The DOWN ARROW key moves you down the list. You can modify and execute any of these previous commands when you display them on your command line.

History Events

In the BASH shell, the history utility keeps a record of the most recent commands you have executed. The commands are numbered starting at 1, and a limit exists to the number of commands remembered—the default is 500. The history utility is a kind of short-term memory, keeping track of the most recent commands you have executed. To see the set of your most recent commands, type **history** on the command line and press ENTER. A list of your most recent commands is then displayed, preceded by a number.

$ **history**

1 cp mydata today
2 vi mydata
3 mv mydata reports
4 cd reports
5 ls

Each of these commands is technically referred to as an event. An event describes an action that has been taken—a command that has been executed. The events are numbered according to their sequence of execution. The most recent event has the largest number.

Each of these events can be identified by its number or beginning characters in the command.

The history utility enables you to reference a former event, placing it on your command line and enabling you to execute it. The easiest way to do this is to use the UP ARROW and DOWN ARROW keys to place history events on your command line, one at a time. You needn't display the list first with **history**. Pressing the UP ARROW key once places the last history event on your command line. Pressing it again places the next history event on your command. Pressing the DOWN ARROW key places the next event on the command line.

You can use certain control and meta keys to perform other history operations like searching the history list. A meta key is the ALT key, or the ESC key on keyboards that have no ALT key. The ALT key is used here. ALT-< will move you to the beginning of the history list; ALT-N will search it. CTRL-S and CTRL-R will perform incremental searches, displaying matching commands as you type in a search string. Table 3-4 lists the different commands for referencing the history list.

History Commands	Description
CTRL-N or DOWN ARROW	Move down to the next event in the history list.
CTRL-P or UP ARROW	Move up to the previous event in the history list.
ALT-<	Move to the beginning of the history event list.
ALT->	Move to the end of the history event list.
ALT-N	Forward search, next matching item.
ALT-P	Backward search, previous matching item.
CTRL-S	Forward search history, forward incremental search.
CTRL-R	Reverse search history, reverse incremental search.
`fc` *event-reference*	Edits an event with the standard editor and then executes it **Options** `-l` List recent history events; same as `history` command `-e` *editor event-reference*; invokes a specified editor to edit a specific event
History Event References	
!*event num*	References an event by its event number.
!!	References the previous command.
!*characters*	References an event beginning with the specified characters.
!?*pattern*?	References an event containing the specified pattern.
!-*event num*	References an event with an offset from the first event.
!*num*-*num*	References a range of events.

You can also reference and execute history events using the **!** history command. The! is followed by a reference that identifies the command. The reference can be either the number of the event or a beginning set of characters in the event. In the next example, the third command in the history list is referenced first by number and then by the beginning characters:

$ **!3**

mv mydata reports

$ **!mv my**

mv mydata reports

You can also reference an event using an offset from the end of the list. A negative number will offset from the end of the list to that event, thereby referencing it. In the next example, the fourth command, **cd mydata**, is referenced using a negative offset, and then executed. Remember that you are offsetting from the end of the list—in this case, event 5— up toward the beginning of the list, event 1. An offset of 4 beginning from event 5 places you at event 2.

$ **!-4**

vi mydata

To reference the last event, you use a following!, as in **!!**. In the next example, the command **!!** executes the last command the user executed—in this case, **ls**:

$ **!!**

Ls

mydata today reports

History Event Editing

You can also edit any event in the history list before you execute it. In the BASH shell, you can do this two ways. You can use the command line editor capability to reference and edit any event in the history list. You can also use a history **fc** command option to reference an event and edit it with the full Vi editor. Each approach involves two different editing capabilities. The first is limited to the commands in the command line editor, which edits only a single line with a subset of Emacs commands. At the same time, however, it enables you to reference events easily in the history list. The second approach invokes the standard Vi editor with all its features, but only for a specified history event.

With the command line editor, not only can you edit the current command, you can also move to a previous event in the history list to edit and execute it. The CTRL-P command then moves you up to the prior event in the list. The CTRL-N command moves you down the list. The ALT-< command moves you to the top of the list, and the ALT-> command moves you to the bottom. You can even use a pattern to search for a given event. The slash followed by a pattern searches backward in the list, and the question mark followed by a pattern searches forward in the list. The **n** command repeats the search.

Once you locate the event you want to edit, you use the Emacs command line editing commands to edit the line. CTRL-D deletes a character. CTRL-F or the RIGHT ARROW moves you forward a character, and CTRL-B or the LEFT ARROW moves you back a character. To add text, you position your cursor and type in the characters you want.

If you want to edit an event using a standard editor instead, you need to reference the event using the **fc** command and a specific event reference, such as an event number.

The editor used is the one specified by the shell in the FCDIT or EDITOR variable. This serves as the default editor for the **fc** command. You can assign to the FCDIT or EDITOR variable a different editor if you wish, such as Emacs instead of Vi. The next example will edit the fourth event, **cd reports**, with the standard editor and then execute the edited event:

```
$ fc 4
```

You can select more than one command at a time to be edited and executed by referencing a range of commands. You select a range of commands by indicating an identifier for the first command followed by an identifier for the last command in the range. An identifier can be the command number or the beginning characters in the command. In the next example, the range

of commands 2 through 4 is edited and executed, first using event numbers and then using beginning characters in those events: $ fc 2 4

$ fc vi c

The **fc** command uses the default editor specified in the FCEDIT special variable (If FCEDIT is not defined, it checks for the EDITOR variable. If neither is defined it uses Vi). Usually, this is the Vi editor. If you want to use the Emacs editor instead, you use the **-e** option and the term **emacs** when you invoke **fc**. The next example will edit the fourth event, **cd reports**, with the Emacs editor and then execute the edited event:

$ fc -e emacs 4

Configuring History: HISTFILE and HISTSAVE

The number of events saved by your system is kept in a special system variable called HISTSIZE. By default, this is usually set to 500. You can change this to another number by simply assigning a new value to HISTSIZE. In the next example, the user changes the number of history events saved to 10:

$ HISTSIZE=10

The actual history events are saved in a file whose name is held in a special variable called HISTFILE. By default, this file is the **.bash_history** file. You can change the file in which history events are saved, however, by assigning its name to the HISTFILE variable. In the next example, the value of HISTFILE is displayed. Then a new filename is assigned to it, **newhist**. History events are then saved in the **newhist** file.

$ **echo $HISTFILE**
.bash_history
$ **HISTFILE="newhist"**
$ **echo $HISTFILE**
Newhist

Filename Expansion: *, ?, [ ]

Filenames are the most common arguments used in a command. Often you may know only part of the filename, or you may want to reference several filenames that have the same extension or begin with the same characters. The shell provides a set of special characters that search out, match, and generate a list of filenames. These are the asterisk, the question mark, and brackets (*****, **?**, **[]**). Given a partial filename, the shell uses these matching operators to search for files and expand to a list of filenames found. The shell replaces the partial filename argument with the expanded list of matched filenames. These filenames can then become the arguments for commands such as **ls**, which can operate on many files. Table 3-5 lists the shell's file expansion characters.

Common Shell Symbols	Execution
ENTER	Execute a command line.
;	Separate commands on the same command line.
`command`	Execute a command.
$(command)	Execute a command.
[]	Match on a class of possible characters in filenames.
\	Quote the following character. Used to quote special characters.
|	Pipe the standard output of one command as input for another command.
&	Execute a command in the background.
!	History command.
File Expansion Symbols	**Execution**
*	Match on any set of characters in filenames.
?	Match on any single character in filenames.
[]	Match on a class of characters in filenames.
Redirection Symbols	**Execution**
>	Redirect the standard output to a file or device, creating the file if it does not exist and overwriting the file if it does exist.
>!	Force the overwriting of a file if it already exists. This overrides the `noclobber` option.
<	Redirect the standard input from a file or device to a program.
>>	Redirect the standard output to a file or device, appending the output to the end of the file.
Standard Error Redirection Symbols	**Execution**
2>	Redirect the standard error to a file or device.
2>>	Redirect and append the standard error to a file or device.
2>&1	Redirect the standard error to the standard output.
>&	Redirect the standard error to a file or device.
|&	Pipe the standard error as input to another command.

Matching Multiple Characters

The asterisk (*****) references files beginning or ending with a specific set of characters. You place the asterisk before or after a set of characters that form a pattern to be searched for in filenames. If the asterisk is placed before the pattern, filenames that end in that pattern are searched for. If the asterisk is placed after the pattern, filenames that begin with that pattern are searched for. Any matching filename is copied into a list of filenames generated by this operation. In the next example, all filenames beginning with the pattern “doc” are searched for and a list is generated. Then all filenames ending with the pattern “day” are searched for and a list is generated. The last example shows how the ***** can be used in any combination of characters.

$ **ls**

doc1 doc2 document docs mydoc monday tuesday

$ **ls doc***

doc1 doc2 document docs

```
$ ls *day
monday tuesday
$ ls m*d*
monday
$
```

Filenames often include an extension specified with a period and followed by a string denoting the file type, such as **.c** for C files, **.cpp** for C++ files, or even **.jpg** for JPEG image files. The extension has no special status and is only part of the characters making up the filename. Using the asterisk makes it easy to select files with a given extension. In the next example, the asterisk is used to list only those files with a **.c** extension. The asterisk placed before the **.c** constitutes the argument for **ls**.

```
$ ls *.c
calc.c main.c
```

You can use ***** with the **rm** command to erase several files at once. The asterisk first selects a list of files with a given extension or beginning or ending with a given set of characters and then it presents this list of files to the **rm** command to be erased. In the next example, the **rm** command erases all files beginning with the pattern "doc":

```
$ rm doc*
```

Matching Single Characters

The question mark (**?**) matches only a single character in filenames. Suppose you want to match the files **doc1** and **docA**, but not the file **document**. Whereas the asterisk will match filenames of any length, the question mark limits the match to just one extra character.

The next example matches files that begin with the word "doc" followed by a single differing letter:

```
$ ls
doc1 docA document
$ ls doc?
doc1 docA
```

Matching a Range of Characters

Whereas the ***** and **?** file expansion characters specify incomplete portions of a filename, the brackets (**[]**) enable you to specify a set of valid characters to search for. Any character placed within the brackets will be matched in the filename. Suppose you want to list files beginning with "doc", but only ending in 1 or A. You are not interested in filenames ending in 2 or B, or any other character. Here is how it's done:

```
$ ls
doc1 doc2 doc3 docA docB docD document
```

$ **ls doc[1A]**

doc1 docA

You can also specify a set of characters as a range, rather than listing them one by one. A dash placed between the upper and lower bounds of a set of characters selects all characters within that range.The range is usually determined by the character set in use. In an ASCII character set, the range "a-g" will select all lowercase alphabetic characters from a through g. In the next example, files beginning with the pattern "doc" and ending in characters 1 through 3 are selected. Then, those ending in characters B through E are matched.

$ **ls doc[1-3]**

doc1 doc2 doc3

$ **ls doc[B-E]**

docBdocD

You can combine the brackets with other file expansion characters to form flexible matching operators. Suppose you want to list only filenames ending in either a **.c** or **.o** extension, but no other extension. You can use a combination of the asterisk and brackets: ***. [co]**. The asterisk matches all filenames, and the brackets match only filenames with extension **.c** or **.o**.

$ **ls *.[co]**

main.c main.o calc.c

Matching Shell Symbols

At times, a file expansion character is actually part of a filename. In these cases, you need to quote the character by preceding it with a backslash to reference the file. In the next example, the user needs to reference a file that ends with the **?** character, **answers?**. The **?** is, however, a file expansion character and would match any filename beginning with "answers" that has one or more characters. In this case, the user quotes the **?** with a preceding backslash to reference the filename.

$ **ls answers\?**

answers?

Placing the filename in double quotes will also quote the character.

$ **ls "answers?"**

answers?

This is also true for filenames or directories that have white space characters like the space character. In this case you can either use the backslash to quote the space character in the file or directory name, or place the entire name in double quotes.

$ **ls My\ Documents**

My Documents

$ **ls "My Documents"**
My Documents

Generating Patterns

Though not a file expansion operation, **{}** is often useful for generating names that you can use to create or modify files and directories. The braces operation only generates a list of names. It does not match on existing filenames. Patterns are placed within the braces and separated with commas. Any pattern placed within the braces will generate a version of the pattern, using either the preceding or following pattern, or both. Suppose you want to generate a list of names beginning with “doc”, but only ending in the patterns “ument”, “final”, and “draft”. Here is how it’s done:

$ **echo doc{ument,final,draft}**
document docfinal docdraft

Since the names generated do not have to exist, you could use the **{}** operation in a command to create directories, as shown here:

$ **mkdir {fall,winter,spring}report**
$ **ls**
fallreport springreport winterreport

Standard Input/Output and Redirection

The data in input and output operations is organized like a file. Data input at the keyboard is placed in a data stream arranged as a continuous set of bytes. Data output from a command or program is also placed in a data stream and arranged as a continuous set of bytes. This input data stream is referred to in Linux as the standard input, and the output data stream is called the standard output. There is also a separate output data stream reserved solely for error messages, called the standard error .

Because the standard input and standard output have the same organization as that of a file, they can easily interact with files. Linux has a redirection capability that lets you easily move data in and out of files. You can redirect the standard output so that, instead of displaying the output on a screen, you can save it in a file. You can also redirect the standard input away from the keyboard to a file, so that input is read from a file instead of from your keyboard.

When a Linux command is executed that produces output, this output is placed in the standard output data stream. The default destination for the standard output data stream is a device—in this case, the screen. Devices, such as the keyboard and screen, are treated as files. They receive and send out streams of bytes with the same organization as that of a byte-stream file. The screen is a device that displays a continuous stream of bytes. By

default, the standard output will send its data to the screen device, which will then display the data.

For example, the **ls** command generates a list of all filenames and outputs this list to the standard output. Next, this stream of bytes in the standard output is directed to the screen device. The list of filenames is then printed on the screen. The **cat** command also sends output to the standard output. The contents of a file are copied to the standard output, whose default destination is the screen. The contents of the file are then displayed on the screen.

Redirecting the Standard Output: > and >>

Suppose that instead of displaying a list of files on the screen, you would like to save this list in a file. In other words, you would like to direct the standard output to a file rather than the screen. To do this, you place the output redirection operator, the greater-than sign (>), followed by the name of a file on the command line after the Linux command. Table 3-6 lists the different ways you can use the redirection operators. In the next example, the output of the **ls** command is redirected from the screen device to a file:

```
$ ls -l *.c > program list
```

The redirection operation creates the new destination file. If the file already exists, it will be overwritten with the data in the standard output. You can set the **noclobber** feature to prevent overwriting an existing file with the redirection operation. In this case, the redirection operation to an existing file will fail. You can overcome the **noclobber** feature by placing an exclamation point after the redirection operator. You can place the **noclobber** command in a shell configuration file to make it an automatic default operation (see Chapter 5). The next example sets the **noclobber** feature for the BASH shell and then forces the overwriting of the **oldletter** file if it already exists:

```
$ set -o noclobber
$ cat myletter >! oldletter
```

Although the redirection operator and the filename are placed after the command, the redirection operation is not executed after the command. In fact, it is executed before the command. The redirection operation creates the file and sets up the redirection before it receives any data from the standard output. If the file already exists, it will be destroyed and replaced by a file of the same name. In effect, the command generating the output is executed only after the redirected file has been created.

In the next example, the output of the **ls** command is redirected from the screen device to a file. First the **ls** command lists files, and in the next command, **ls** redirects its file list to the **listf** file. Then the **cat** command

displays the list of files saved in **listf**. Notice the list of files in **listf** includes the **listf** filename. The list of filenames generated by the **ls** command

Command	Execution
ENTER	Execute a command line.
;	Separate commands on the same command line.
command\ *opts args*	Enter backslash before pressing ENTER to continue entering a command on the next line.
`*command*`	Execute a command.
$(*command*)	Execute a command.
Special Characters for Filename Expansion	**Execution**
*	Match on any set of characters.
?	Match on any single characters.
[]	Match on a class of possible characters.
\	Quote the following character. Used to quote special characters.
Redirection	**Execution**
command > *filename*	Redirect the standard output to a file or device, creating the file if it does not exist and overwriting the file if it does exist.
command < *filename*	Redirect the standard input from a file or device to a program.
command >> *filename*	Redirect the standard output to a file or device, appending the output to the end of the file.
command >! *filename*	In the C shell and the Korn shell, the exclamation point forces the overwriting of a file if it already exists. This overrides the `noclobber` option.
command 2> *filename*	Redirect the standard error to a file or device in the Bourne shell.
command 2>> *filename*	Redirect and append the standard error to a file or device in the Bourne shell.
command 2>&1	Redirect the standard error to the standard output in the Bourne shell.
command >& *filename*	Redirect the standard error to a file or device in the C shell.

Pipes	Execution
command | *command*	Pipe the standard output of one command as input for another command.
command |& *command*	Pipe the standard error as input to another command in the C shell.

includes the name of the file created by the redirection operation—in this case, **listf**. The **listf** file is first created by the redirection operation, and then the **ls** command lists it along with other files.

$ **ls**

mydata intro preface

$ **ls > listf** $ **cat listf**

mydata intro listf preface

You can also append the standard output to an existing file using the >> redirection operator. Instead of overwriting the file, the data in the standard output is added at the end of the file. In the next example, the **myletter** and **oldletter** files are appended to the **alletters** file. The **alletters** file will then contain the contents of both **myletter** and **oldletter**.

$ cat myletter >> alletters

$ cat oldletter >> alletters

The Standard Input

Many Linux commands can receive data from the standard input. The standard input itself receives data from a device or a file. The default device for the standard input is the keyboard. Characters typed on the keyboard are placed in the standard input, which is then directed to the Linux command. Just as with the standard output, you can also redirect the standard input, receiving input from a file rather than the keyboard. The operator for redirecting the standard input is the less-than sign (<). In the next example, the standard input is redirected to receive input from the **myletter** file, rather than the keyboard device (use CTRL-D to end the typed input). The contents of **myletter** are read into the standard input by the redirection operation. Then the **cat** command reads the standard input and displays the contents of **myletter**.

```
$ cat < myletter
hello Christopher
How are you today
$
```

You can combine the redirection operations for both standard input and standard output. In the next example, the **cat** command has no filename arguments. Without filename arguments, the **cat** command receives input from the standard input and sends output to the standard output. However, in the example the standard input has been redirected to receive its data from a file, while the standard output has been redirected to place its data in a file.

```
$ cat < myletter > newletter
```

Pipes: |

You may find yourself in situations in which you need to send data from one command to another. In other words, you may want to send the standard output of a command to another command, not to a destination file. Suppose you want to send a list of your filenames to the printer to be printed. You need two commands to do this: the **ls** command to generate a list of filenames and the **lpr** command to send the list to the printer. In effect, you need to take the output of the **ls** command and use it as input for the **lpr** command. You can think of the data as flowing from one command to another. To form such a connection in Linux, you use what is called a pipe. The pipe operator (|, the vertical bar character) placed between two

commands forms a connection between them. The standard output of one command becomes the standard input for the other. The pipe operation receives output from the command placed before the pipe and sends this data as input to the command placed after the pipe. As shown in the next example, you can connect the **ls**command and the **lpr** command with a pipe. The list of filenames output by the **ls** command is piped into the **lpr** command.

$ ls | lpr

You can combine the **pipe** operation with other shell features, such as file expansion characters, to perform specialized operations. The next example prints only files with a **.c** extension. The **ls** command is used with the asterisk and ".c" to generate a list of filenames with the **.c** extension. Then this list is piped to the **lpr** command.

$ ls *.c | lpr

In the preceding example, a list of filenames was used as input, but what is important to note is that pipes operate on the standard output of a command, whatever that might be. The contents of whole files or even several files can be piped from one command to another. In the next example, the **cat** command reads and outputs the contents of the **mydata** file, which are then piped to the **lpr** command:

$ cat mydata | lpr

Linux has many commands that generate modified output. For example, the **sort** command takes the contents of a file and generates a version with each line sorted in alphabetic order. The **sort** command works best with files that are lists of items. Commands such as **sort** that output a modified version of its input are referred to as filters. Filters are often used with pipes. In the next example, a sorted version of **mylist** is generated and piped into the **more** command for display on the screen. Note that the original file, **mylist**, has not been changed and is not itself sorted. Only the output of **sort** in the standard output is sorted.

$ sort mylist | more

The standard input piped into a command can be more carefully controlled with the standard input argument (-). When you use the dash as an argument for a command, it represents the standard input.

Redirecting the Standard Error:2>, >>

When you execute commands, an error could possibly occur. You may give the wrong number of arguments, or some kind of system error could take place. When an error occurs, the system issues an error message. Usually such error messages are displayed on the screen, along with the standard output. Linux distinguishes between standard output and error the

standard error. In the next example, the **cat** command is given as its argument the name of a file that does not exist, **myintro**. In this case, the **cat** command simply issues an error:

$ **cat myintro**
cat : myintro not found
$

Because error messages are in a separate data stream from the standard output, error messages still appear on the screen for you to see even if you have redirected the standard output to a file. In the next example, the standard output of the **cat** command is redirected to the file **mydata**. However, the standard error, containing the error messages, is still directed to the screen.

$ **cat myintro > mydata**
cat : myintro not found
$

You can redirect the standard error, as you can the standard output. This means you can save your error messages in a file for future reference. This is helpful if you need a record of the error messages. Like the standard output, the standard error has the screen device for its default destination. However, you can redirect the standard error to any file or device you choose using special redirection operators. In this case, the error messages will not be displayed on the screen.

Redirection of the standard error relies on a special feature of shell redirection. You can reference all the standard byte streams in redirection operations with numbers. The numbers 0, 1, and 2 reference the standard input, standard output, and standard error, respectively. By default, an output redirection, >, operates on the standard output, 1. You can modify the output redirection to operate on the standard error, however, by preceding the output redirection operator with the number 2. In the next example, the **cat** command again will generate an error. The error message is redirected to the standard byte stream represented by the number 2, the standard error.

$ **cat nodata 2> myerrors**
$ **cat myerrors**
cat : nodata not found
$

You can also append the standard error to a file by using the number 2 and the redirection append operator (>>). In the next example, the user appends the standard error to the **myerrors** file, which then functions as a log of errors:

```
$ cat nodata 2>> myerrors
```

Jobs: Background, Kills, and Interruptions

In Linux, you not only have control over a command's input and output, but also over its execution. You can run a job in the background while you execute other commands. You can also cancel commands before they have finished executing. You can even interrupt a command, starting it again later from where you left off. Background operations are particularly useful for long jobs. Instead of waiting at the terminal until a command finishes execution, you can place it in the background. You can then continue executing other Linux commands.

Running Jobs in the Background

You execute a command in the background by placing an ampersand (**&**) on the command line at the end of the command. When you place a job in the background, a user job number and a system process number are displayed. The user job number, placed in brackets, is the number by which the user references the job. The system process number is the number by which the system identifies the job. In the next example, the command to print the file **mydata** is placed in the background:

```
$ lpr mydata &
[1] 534
$
```

Background Jobs	Execution
%*jobnum*	References job by job number, use the **jobs** command to display job numbers.
%	References recent job.
%*string*	References job by an exact matching string.
%?*string*?	References job that contains unique string.
%--	References job before recent job.
&	Execute a command in the background.
fg %*jobnum*	Bring a command in the background to the foreground or resume an interrupted program.
bg	Place a command in the foreground into the background.
CTRL-Z	Interrupt and stop the currently running program. The program remains stopped and waiting in the background for you to resume it.
notify %*jobnum*	Notifies you when a job ends.
kill %*jobnum* **kill** *processnum*	Cancel and end a job running in the background.
jobs	List all background jobs.
ps -a	List all currently running processes, including background jobs.
at *time date*	Execute commands at a specified time and date. The time can be entered with hours and minutes, and qualified as A.M. or P.M.

You can place more than one command in the background. Each is classified as a job and given a name and a job number. The command **jobs** list the jobs being run in the background. Each entry in the list consists of

the job number in brackets, whether it is stopped or running, and the name of the job. The + sign indicates the job currently being processed, and the - sign indicates the next job to be executed. In the next example, two commands have been placed in the background. The **jobs** command then lists those jobs, showing which one is currently being executed.

```
$ lpr intro &
 [1] 547
$ cat *.c > myprogs &
[2] 548
$ jobs
[1] + Running lpr intro
[2] - Running cat *.c > myprogs
$
```

Referencing Jobs

Normally jobs are referenced using the job number, preceded by a **%** symbol. You can obtain this number with the **jobs** command, which will list all background jobs, as shown in the preceding example. In addition you can also reference a job using an identifying string (see Table 3-7). The string must be either an exact match or a partial unique match. If there is no exact or unique match, you will receive an error message. Also, the **%** symbol itself without any job number references the recent background job. Followed by a -- it references the second previous background job. The following example brings job 1 in the previous example to the foreground.

```
fg %lpr
```

Job Notification

After you execute any command in Linux, the system tells you what background jobs, if you have any running, have been completed so far. The system does not interrupt any operation, such as editing, to notify you about a completed job. If you want to be notified immediately when a certain job ends, no matter what you are doing on the system, you can use the **notify** command to instruct the system to tell you. The **notify** command takes a job number as its argument. When that job is finished, the system interrupts what you are doing to notify you the job has ended. The next example tells the system to notify the user when job 2 finishes:

```
$ notify %2
```

Bringing Jobs to the Foreground

You can bring a job out of the background with the foreground command, **fg**. If only one job is in the background, the **fg** command alone will bring it to the foreground. If more than one job is in the background, you must use

the job's number with the command. You place the job number after the **fg** command, preceded by a percent sign. A **bg** command, usually used second command is now in the foreground and executing. When the command is finished executing, the prompt appears and you can execute another command.

```
$ fg %2
cat *.c > myprogs
$
```

Canceling Jobs

If you want to cancel a job running in the background, you can force it to end with the **kill** command. The **kill** command takes as its argument either the user job number or the system process number. The user job number must be preceded by a percent sign (**%**). You can find out the job number from the **jobs** command. In the next example, the **jobs** command lists the background jobs; then job 2 is canceled:

```
$ jobs
[1] + Running lpr intro
[2] - Running cat *.c > myprogs
$ kill %2
```

Suspending and Stopping Jobs

You can suspend a job and stop it with the CTRL-Z key. This places the job to the side until it is restarted. The job is not ended; it merely remains suspended until you want to continue. When you're ready, you can continue with the job in either the foreground or the background using the **fg** or **bg** command. The **fg** command restarts a suspended job in the foreground. The **bg** command places the suspended job in the background.

At times, you may need to place a currently running job in the foreground into the background. However, you cannot move a currently running job directly into the background. You first need to suspend it with CTRL-Z and then place it in the background with the **bg** command. In the next example, the current command to list and redirect **.c** files is first suspended with CTRL-Z. Then that job is placed in the background.

```
$ cat *.c > myprogs
^Z
$ bg
```

Ending Processes: ps and kill

You can also cancel a job using the system process number, which you can obtain with the **ps** command. The **ps** command will display your processes, and you can use a process number to end any running process. The **ps** command displays a great deal more information than the **jobs** command

does. The next example lists the processes a user is running. The PID is time is how long the process has taken so far. COMMAND is the name of the process.

```
$ ps
PID    TTY      TIME     COMMAND
523    tty24    0:05     sh
567    tty24    0:01     lpr
570    tty24    0:00     ps
```

You can then reference the system process number in a **kill** command. Use the process number without any preceding percent sign. The next example kills process 567:

$ kill 567

Check the **ps** Man page for more detailed information about detecting and displaying process information. To just display a PID number, use the output options **-o pid=.** Combined with the **-C** command option you can display just the PID for a particular command. If there is more than one process for that command, such as multiple bash shells, then all the PIDs will be displayed.

$ ps -C lpr -o pid=

For unique commands, those you know have only one process running, you can safely combine the previous command with the **kill** command to end the process on one line. This avoids interactively having to display and enter the PID to kill the process. The technique can be useful for noninteractive operations like **cron** and helpful for ending open-ended operations like video recording. In the following example, a command using just one process, getatse, is ended in a single kill operation. The **getatsc** is an hdtv recording command. Backquotes are used to first execute the ps command to obtain the PID.

kill `ps -C getatsc -o pid=`

The C Shell: Command Line Editing and History

The C shell was originally developed for use with BSD Unix. With Linux, it is available as an alternative shell, along with the Korn and Bourne shells. The C shell incorporates all the core commands used in the Bourne shell but differs significantly in more complex features such as shell programming. The C shell was developed after the Bourne shell and was the first to introduce new features such as command line editing and the history utility. The Korn shell then later incorporated many of these same features. Then the bash shell, in turn, incorporated many of the features of all these shells. However, the respective implementations differ significantly. The C shell has limited command line editing that allows you to perform a few basic editing operations. C shell command line editing is not nearly as powerful as Korn shell command line editing. The history

utility allows you to execute and edit previous commands. The history utility works in much the same way in the Korn, BASH, Z, and C shells. However, their command names differ radically, and the C shell has a very different set of history editing operations.
On most Linux distributions, an enhanced version of the C shell is used, called TCSH. Most of the commands are similar. You can access the C shell with the command **csh,** which is a link to the TCSH shell. The traditional prompt for the C shell is the **%** symbol. On some Linux distributions the prompt may remain the unchanged **$**.

$ **csh**
%

The command for entering the TCSH shell is **tcsh.**

C Shell Command Line Editing L

like the BASH shell, the C shell has only limited command line editing capabilities. They are, however, more powerful than those of the Bourne shell. Instead of deleting only a single character, you can delete a whole word. You can also perform limited editing operations using pattern substitution.
The CTRL-W key erases a recently entered word. The term "word" here is more of a technical concept that denotes how the shell parses a command. A word is parsed on a space or tab. Any character or set of characters surrounded by spaces or tabs is considered a word. With the CTRL-W key you can erase the text you have entered a word at a time.

% date who
% date

Other times you may need to change part of a word or several words in a command line. The C shell has a pattern substitution command that allows you to replace patterns in the command line. This substitution command is represented by a pattern enclosed in ^ symbols. The pattern to be replaced is enclosed between two ^. The replacement text immediately follows.

% ^pattern^newtext

The pattern substitution operation is not solely an editing command. It is also an execution command. Upon replacing the pattern, the corrected command will be displayed and then executed. In the next example, the date command has been misspelled. The shell displays an error message saying that such a command cannot be found. You can edit that command using the ^ symbols to replace the incorrect text. The command is then executed.

% **dte**
dte: not found

```
% ^dt^dat
 Date
 Sun July 5 10:30:21 PST 1992
%
```

C Shell History Utility

As in the BASH shell, the C shell history utility keeps a record of the most recent commands you have executed. Table 3-8 lists the C shell history commands. The history utility keeps track of a limited number of the most recent commands, which are numbered from 1. The history utility

C Shell Event References	
!*event num*	References an event by its event number.
!*characters*	References an event beginning with specified characters.
!?*pattern*?	References an event containing the specified pattern.
!-*event num*	References an event with an offset from the first event.
!*num-num*	References a range of events.
C Shell Event Word References	
!*event num*:*word num*	References a particular word in an event.
!*event num*:^	References first argument (second word) in an event.
!*event num*:$	References last argument in an event.
!*event num*:^-$	References all arguments in an event.
!*event num*:*	References all arguments in an event.
C Shell Event Editing Substitutions	
!*event num*:s/*pattern*/*newtext*/	Edits an event with a pattern substitution. References a particular word in an event.
!*event num*:sg/*pattern*/*newtext*/	Performs a global substitution on all instances of a pattern in the event.
!*event num*:s/*pattern*/*newtext*/p	Suppresses execution of the edited event.

is not automatically turned on. You first have to define history with a **set** command and assign to it the number of commands you want recorded. This is often done as part of your shell configuration. In the next example, the history utility is defined and set to remember the last five commands.

```
% set history=5
```

As in the BASH shell, the commands remembered are referred to as events. To see the set of your most recent events, enter the word **history** on the command line and press ENTER. A list of your most recent commands is then displayed, with each event preceded by an event number.

```
% history
1 ls
2 vi mydata
3 mv mydata reports
4 cd reports
5 ls -F
```

Each of these events can be referenced by its event number, the beginning characters of the event, or a pattern of characters in the event. A pattern reference is enclosed in question marks, **?.** You can re-execute any event using the history command**!.** The exclamation point is followed by an event reference such as an event number, beginning characters, or a pattern. In the next examples, the second command in the history list is referenced first by an event number, then by the beginning characters of the event, and then by a pattern in the event.

%**!2**

vi mydata

% **!vi**

vi mydata

% **!?myd?**

vi mydata

You can also reference a command using an offset from the end of the list. Preceding a number with a minus sign will offset from the end of the list to that command. In the next example, the second command, **vi mydata**, is referenced using an offset.

% **!-4**

vi mydata

An exclamation point is also used to identify the last command executed. It is equivalent to an offset of -1. In the next examples, both the offset of 1 and the exclamation point reference the last command, Is -F.

% **!!**

ls -F

mydata /reports

% **!-1**

ls -F mydata /reports

C Shell History Event Substitutions

An event reference should be thought of as a representation of the characters making up the event. The event reference**!1** actually represents the characters “ls”. As such, you can use an event reference as part of another command. The history operation can be thought of as a substitution. The characters making up the event replace the exclamation point and event reference entered on the command line. In the next example, the list of events is first displayed. Then a reference to the first event is used as part of a new command. The event reference!1 evaluates to ls, becoming part of the command ls > myfiles.

% history

1 ls

2 vi mydata
3 mv mydata reports
4 cd reports
5 ls -F
% !1 > myfiles
ls > myfiles

You can also reference particular words in an event. An event is parsed into separated words, each word identified sequentially by a number starting from 0. An event reference followed by a colon and a number references a word in the event. The event reference**!3:2** references the second word in the third event. It first references the third event, **mv mydata reports**, and the second word in that event **mydata**. You can use such word references as part of a command. In the next example, **2:0** references the first word in the second event, **vi**, and replaces it with preface.

% **!2:0 preface**
vi preface

Using a range of numbers, you can reference several words in an event. The number of the first and last word in the range are separated by a dash. In the next example, **3:0-1** references the first two words of the third event, **mv mydata.**

% !3:0-1 oldletters

The metacharacters ^ and $ represent the second word and the last word in an event. They are used to reference arguments of the event. If you need just the first argument of an event, then ^ references it. **$** references the last argument. The range of ^-$ references all the arguments. (The first word, the command name, is not included.) In the next example, the arguments used in previous events are referenced and used as arguments in the current command. First, the first argument (the second word) in the second event, **mydata**, is used as an argument in an lp command, to print a file. Then, the last argument in the third event, **reports**, is used as an argument in the ls command, to list the filenames in reports. Then the arguments used in the third event, **mydata** and **reports**, are used as arguments in a **copy** command.

% **lpr !2:^**
lpr mydata
% **ls !3:$**
ls reports
% **cp !3:^-$**
cp mydata reports

The asterisk is a special symbol that represents all the arguments in a former command. It is equivalent to the range **^-$**. The last example can be rewritten using the asterisk, **!3***.

% **cp !3***

cp mydata reports

In the C shell, whenever the exclamation point is used in a command, it is interpreted as a history command reference. If you need to use the exclamation point for other reasons, such as an electronic mail address symbol, you have to quote the exclamation point by placing a backslash in front of it.

% mail garnet\!chris < mydata

C Shell History Event Editing

You can edit history commands with a substitution command. The substitution command operates in the same way as the ^ command for command line editing. It replaces a pattern in a command with new text. To change a specific history command, enter an exclamation point and the event number of that command followed by a colon and the substitution command. The substitution command begins with the character **s** and is followed by a pattern enclosed in two slashes. The replacement text immediately follows, ending with a slash.

% !num:s/pattern/newtext/

In the next example, the pattern “my” in the third event is changed to “your”. The changed event is then displayed and executed.

% history

1 ls

2 vi mydata

3 mv mydata reports

4 cd reports

5 ls –F

% !3:s/my/your/

mv yourdata reports

%

Preceding the **s** command with a **g** will perform a global substitution on an event. Every instance of the pattern in the event will be changed. In the next example, the extension of every filename in the first event is changed from **.c** to **.p** and then executed.

% lpr calc.c

lib.c % **!1:gs/.c/.p/**

lpr calc.p lib.p

%

The **&** command will repeat the previous substitution. In the next example the same substitution is performed on two commands, changing the filename **mydata** to **yourdata** in both the third and second events.

% !3:s/my/your/
mv yourdata reports
% !2:&
vi yourdata

When you perform a history operation on a command, it is automatically executed. You can suppress execution with a **p** qualifier. The **p** qualifier will only display the modified command, not execute it. This allows you to perform several operations on a command before you execute it. In the next example, two substitution commands are performed on the third command before it is executed.

% !3:s/mv/cp/:p Does not execute the command
cp mydata reports
% !3:s/reports/books/ Changes and executes the command
cp mydata books
%

The TCSH Shell

The TCSH shell is essentially a version of the C shell with added features. It is fully compatible with the standard C shell and incorporates all of its capabilities, including the shell language and the history utility. TCSH has more advanced command line and history editing features than those found in the original C shell. You can use either Vi or Emacs key bindings to edit commands or history events. The TCSH shell also supports command line completion, automatically completing a command using just the few first characters you type in. TCSH shell has native language support, extensive terminal management, new built-in commands, and system variables. See the Man page for TCSH for more detailed information.

TCSH Command Line Completion

The command line has a built-in feature that performs command and filename completion. If you enter an incomplete pattern as a filename argument, you can press TAB to activate this feature, which will then complete the pattern to generate a filename. To use this feature, you type the partial name of the file on the command line and then press TAB. The shell will automatically look for the file with that partial prefix and complete it for you on the command line. In the next example, the user issues a **cat** command with an incomplete filename. When the user presses TAB, the system searches for a match and, upon finding one, fills in the filename.

> cat pre TAB
> cat preface
If more than one file has the same prefix, the shell will match the name as far as the filenames agree and then beep. You can then add more characters to select one or the other.
For example:
> ls
document docudrama
> **cat doc** TAB
> **cat docu** beep
If, instead, you want a list of all the names that your incomplete filename matches, you can press CTRL-D on the command line. In the next example, the CTRL-D after the incomplete filename generates a list of possible filenames.
> **cat doc** Ctrl-d
Document
Docudrama
> **cat docu**
The shell redraws the command line, and you can then type in the remainder of the filename, or type in distinguishing characters, and press TAB to have the filename completed.
> cat docudrama

TCSH History Editing

As in the C shell, the TCSH shell's history utility keeps a record of the most recent commands you have executed. The history utility is a kind of short-term memory, keeping track of a limited number of the most recent commands. The history utility lets you reference a former event by placing it on your command line and allowing you to execute it. However, you do not need to display the list first with history. The easiest way to do this is to use your UP ARROW and DOWN ARROW keys to place history events on your command line one at a time. Pressing the UP ARROW key once will place the last history event on your command line. Pressing it again places the next history event on your command line. The DOWN ARROW key will place the next command on the command line. You can also edit the command line. The LEFT ARROW and RIGHT ARROW keys move you along the command line. You can then insert text wherever you stop your cursor. With the BACKSPACE and DELETE keys, you can delete characters. CTRL-A moves your cursor to the beginning of the command line, and CTRL-E moves it to the end. CTRL-K deletes the

remainder of a line from the position of the cursor, and CTRL-U erases the entire line.

The Z-Shell

The Z-shell includes all of the features of the Korn shell and adds command line and history event features. The Z-shell performs automatic expansion on the command line after it has been parsed. Expansions are performed on filenames, processes, parameters, commands, arithmetic expressions, braces, and filename generation. The Z-shell supports the use of Vi and Emacs key bindings for referencing history events, much like the BASH shell does. The UP ARROW and CTRL-P move you up to the previous event, and the DOWN ARROW and CTRL-N move you down to the next one. ESC < moves you to the first event and ESC > moves you to the last. The RIGHT and LEFT ARROWS move through an event line. CTRL-R CTRL-X performs a search of the history events. History events can also be referenced using the **!** symbol, much like C shell history. When you enter the history command, a list of previous commands (called events) will be displayed, each with a number. To reference an event, enter the **!** symbol and its number. The following example references the third event.

!3

You can reference an event in several ways. You can use an offset from the current command, use a pattern to identify an event, or specify the beginning characters of an event. Table lists these alternatives.

You can use word designators to include just segments of a history event in your command. A word designator indicates which word or words of a given command line will be included in a history reference. A colon separates the event number from the word designator. It can be omitted if the word designator begins with a ^, **$**, ***** , **-**, or **%**. The words are numbered from 0, with 0 referring to the first word in an event, and 1 to the second word. **$** references the last word. A caret, ^, references the first argument, the first word after the command word (same as 1). You can reference a range of words or, with *****, the remaining words in an event. To reference all the words from the third one to the end, use **3***. The ***** by itself references all the arguments (from 1 on). The following example references the second, third, and fourth words in the sixth event.

!6:2-4

Z-Shell History Commands	
!	Starts a history substitution, except when followed by a blank, newline, =, or (.
!!	Refers to the previous command. By itself, repeats the previous command.
!*num*	Refers to command line *num*.
!-*num*	Refers to the current command line minus num.
!*str*	Refers to the most recent command starting with str.
!?str[?]	Refers to the most recent command containing.
!#	Refers to the current command line typed so far.
!{...}	Insulates a history reference from adjacent characters (if necessary).
Z-Shell Word Designators	
0	The first input word (command).
num	The *num*th argument.
^	The first argument, that is, 1.
$	The last argument.
%	The word matched by (the most recent) ?*str* search.
str-str	A range of words; -str abbreviates 0-str.
*	All the arguments, or a null value if there is just one word in the event.
*str**	Abbreviates str-$.
str-	Like str* but omitting word $.

Review and Self Assessment Question:

Q1- What do you mean by shell?
Q2-Describe the term Comman Line ?
Q3-Define Filename Expansion with *,? & [ ]?
Q4- Describe the term “The Standard Input”?
Q5-What is C shell ?
Q6- What is TCSH shell?

Further Readings

Linux Operating System Richard Petersen
Linux Operating System Paul S. Wang
Linux Operating System by David Maxwell and Andrew Bedford
Linux Operating System by Richard Blum and Christine Bresnahan
Linux Operating System by Bhatt P.C.P

UNIT-4 THE SHELL SCRIPTS AND PROGRAMMING

Contents

Introduction

A shell script combines Linux commands in such a way as to perform a specific task. The different kinds of shells provide many programming tools that you can use to create shell programs. You can define variables and assign values to them. You can also define variables in a script file and have a user interactively enter values for them when the script is executed.The shell provides loop and conditional control structures that repeat Linux commands or make decisions on which commands you want to execute.You can also construct expressions that perform arithmetic or comparison operations. All these shell programming tools operate in ways similar to those found in other programming languages, so if you're already familiar with programming, you might find shell programming simple to learn.

The BASH, TCSH, and Z shells described in Chapter 3 are types of shells. You can have many instances of a particular kind of shell. A shell, by definition, is an interpretive environment within which you execute commands. You can have many environments running at the same time, of either the same or different types of shells; you can have several shells running at the same time that are of the BASH shell type, for example.

This chapter will cover the basics of creating a shell program using the BASH and TCSH shells, the shells used on most Linux systems. You will learn how to create your own scripts, define shell variables, and develop user interfaces, as well as learn the more difficult task of combining

control structures to create complex programs. Tables throughout the chapter list shell commands and operators, and numerous examples show how they are implemented.

Usually, the instructions making up a shell program are entered into a script file that can then be executed. You can even distribute your program among several script files, one of which will contain instructions on how to execute others. You can think of variables, expressions, and control structures as tools you use to bring together several Linux commands into one operation. In this sense, a shell program is a new and complex Linux command that you have created.

The BASH shell has a flexible and powerful set of programming commands that allows you to build complex scripts. It supports variables that can be either local to the given shell or exported to other shells. You can pass arguments from one script to another. The BASH shell has a complete set of control structures, including loops and **if** statements as well as case structures, all of which you'll learn about as you read this book. All shell commands interact easily with redirection and piping operations that allow them to accept input from the standard input or send it to the standard output. Unlike the Bourne shell, the first shell used for Unix, BASH incorporates many of the features of the TCSH and Z shells. Arithmetic operations in particular are easier to perform in BASH.

The TCSH shell, like the BASH shell, also has programming language capabilities. You can define variables and assign values to them. You can place variable definitions and Linux commands in a script file and then execute that script. You can use loop and conditional control structures to repeat Linux commands or make decisions on which commands you want to execute. You can also place traps in your program to handle interrupts.

The TCSH shell differs from other shells in that its control structures conform more to a programming-language format. For example, the test condition for a TCSH shell's control structure is an expression that evaluates to true or false, not to a Linux command. A TCSH shell expression uses the same operators as those found in the C programming language. You can perform a variety of assignment, arithmetic, relational, and bitwise operations. The TCSH shell also allows you to declare numeric variables that can easily be used in such operations.

Shell Variables

Within each shell, you can enter and execute commands. You can further enhance the capabilities of a shell using shell variables. With a shell variable, you can hold data that you can reference over and over again as you execute different commands within a given shell. For example, you

can define a shell variable to hold the name of complex filename. Then, instead of retyping the filename in different commands, you can reference it with the shell variable.

You define variables within a shell, and such variables are known as shell variables. Some utilities, such as the Mail utility, have their own shells with their own shell variables. You can also create your own shell using what are called shell scripts. You have a user shell that becomes active as soon as you log in. This is often referred to as the login shell. Special system-level parameter variables are defined within this login shell. Shell variables can also be used to define a shell's environment.

Definition and Evaluation of Variables: =, $, set, unset

You define a variable in a shell when you first use the variable's name. A variable's name may be any set of alphabetic characters, including the underscore. The name may also include a number, but the number cannot be the first character in the name. A name may not have any other type of character, such as an exclamation point, an ampersand, or even a space. Such symbols are reserved by the shell for its own use. Also, a variable name may not include more than one word. The shell uses spaces on the command line to distinguish different components of a command such as options, arguments, and the name of the command.

You assign a value to a variable with the assignment operator (=). You type the variable name, the assignment operator, and then the value assigned. Do not place any spaces around the assignment operator. The assignment operation **poet = Virgil**, for example, will fail. (The C shell has a slightly different type of assignment operation.) You can assign any set of characters to a variable. In the next example, the variable **poet** is assigned the string **Virgil**:

$ **poet=Virgil**

Once you have assigned a value to a variable, you can then use the variable name to reference the value. Often you use the values of variables as arguments for a command. You can reference the value of a variable using the variable name preceded by the **$** operator. The dollar sign is a special operator that uses the variable name to reference a variable's value, in effect evaluating the variable. Evaluation retrieves a variable's value, usually a set of characters. This set of characters then replaces the variable name on the command line. Wherever a **$** is placed before the variable name, the variable name is replaced with the value of the variable. In the next example, the shell variable **poet** is evaluated and its contents, **Virgil**, are then used as the argument for an **echo** command. The **echo** command simply echoes or prints a set of characters to the screen.

$ **echo $poet**

Virgil

You must be careful to distinguish between the evaluation of a variable and its name alone. If you leave out the **$** operator before the variable name, all you have is the variable name itself. In the next example, the **$** operator is absent from the variable name. In this case, the **echo** command has as its argument the word "poet", and so prints out "poet":

$ **echo poet**

poet

The contents of a variable are often used as command arguments. A common command argument is a directory pathname. It can be tedious to retype a directory path that is being used over and over again. If you assign the directory pathname to a variable, you can simply use the evaluated variable in its place. The directory path you assign to the variable is retrieved when the variable is evaluated with the **$** operator. The next example assigns a directory pathname to a variable and then uses the evaluated variable in a copy command. The evaluation of **ldir** (which is **$ldir**) results in the pathname **/home/chris/letters**. The copy command evaluates to **cp myletter /home/chris/letters**.

$ **ldir=/home/chris/letters**

$ **cp myletter $ldir**

You can obtain a list of all the defined variables with the **set** command. If you decide you do not want a certain variable, you can remove it with the **unset** command. The **unset** command undefines a variable.

Variable Values: Strings

The values that you assign to variables may consist of any set of characters. These characters may be a character string that you explicitly type in or the result obtained from executing a Linux command. In most cases, you will need to quote your values using either single quotes double quotes, backslashes, or back quotes. Single quotes, double quotes, and backslashes allow you to quote strings in different ways. Back quotes have the special function of executing a Linux command and using its results as arguments on the command line.

Quoting Strings: Double Quotes, Single Quotes, and Backslashes

Variable values can be made up of any characters. However, problems occur when you want to include characters that are also used by the shell as operators. Your shell has certain meta characters that it uses in evaluating the command line. A space is used to parse arguments on the

command line. The asterisk, question mark, and brackets are meta characters used to generate lists of filenames. The period represents the current directory. The dollar sign, **$**, is used to evaluate variables, and the greater-than (>) and less-than (<) characters, are redirection operators. The ampersand is used to execute background commands and the bar pipes output. If you want to use any of these characters as part of the value of a variable, you first need to quote them. Quoting a meta character on a command line makes it just another character. It is not evaluated by the shell.

You can use double quotes, single quotes, and backslashes to quote such metacharacters. Double and single quotes allow you to quote several metacharacters at a time. Any metacharacters within double or single quotes are quoted.A backslash quotes the single character that follows it.

If you want to assign more than one word to a variable, you need to quote the spaces separating the words. You can do so by enclosing all the words within double quotes. You can think of this as creating a character string to be assigned to the variable. Of course, any other metacharacters enclosed within the double quotes are also quoted.

In the following first example, the double quotes enclose words separated by spaces. Because the spaces are enclosed within double quotes, they are treated as characters, not as delimiters used to parse command line arguments. In the second example, double quotes also enclose a period, treating it as just a character. In the third example, an asterisk is also enclosed within the double quotes. The asterisk is considered just another character in the string and is not evaluated.

$ **notice="The meeting will be tomorrow"**

$ **echo $notice**

The meeting will be tomorrow

$ **message="The project is on time."**

$ **echo $message**

The project is on time.

$ **notice="You can get a list of files with ls *.c"**

$ **echo $notice**

You can get a list of files with ls *.c

Double quotes, however, do not quote the dollar sign, the operator that evaluates variables. A **$** operator next to a variable name enclosed within double quotes will still be evaluated, replacing the variable name with its value. The value of the variable will then become part of the string, not the variable name. There may be times when you want a variable within

quotes to be evaluated. In the next example, the double quotes are used so that the winner's name will be included in the notice.

$ **winner=dylan**
$ **notice="The person who won is $winner"**
$ **echo $notice**
The person who won is dylan

On the other hand, there may be times when you do not want a variable within quotes to be evaluated. In that case you have to use the single quotes. Single quotes suppress any variable evaluation and treat the dollar sign as just another character. In the next example, single quotes prevent the evaluation of the winner variable.

$ **winner=Dylan**
$ **result='The name is in the $winner variable'**
$ **echo $result**
The name is in the $winner variable

If, in this case, the double quotes were used instead, an unintended variable evaluation would take place. In the next example, the characters "$winner" are interpreted as a variable evaluation.

$ **winner=dylan**
$ **result="The name is in the $winner variable"**
$ **echo $result**
The name is in the dylan variable

You can always quote any metacharacter, including the **$** operator, by preceding it with a backslash. The use of the backslash is to quote ENTER keys (newlines). The backslash is useful when you want to both evaluate variables within a string and include the **$** character. In the next example, the backslash is placed before the **$** to treat it as a dollar sign character: **\$.** At the same time the variable **$winner** is evaluated because the double quotes that are used do not quote the **$** operator.

$ **winner=Dylan**
$ **result="$winner won \$100.00"**
$ **echo $result**
dylan won $100.00

Quoting Commands: Single Quotes

There are, however, times when you may want to use single quotes around a Linux command. Single quotes allow you to assign the written command to a variable. If you do so, you can then use that variable name as another name for the Linux command. Entering the variable name preceded by the **$** operator on the command line will execute the command. In the next example, a shell variable is assigned the characters that make up a Linux

command to list files, **'ls -F'**. Notice the single quotes around the command. When the shell variable is evaluated on the command line, the Linux command it contains will become a command line argument, and it will be executed by the shell.

$ **lsf='ls -F'**
$ **$lsf**
mydata /reports /letters
$

In effect you are creating another name for a command, like an alias.

Values from Linux Commands: Back Quotes

Although you can create variable values by typing in characters or character strings, you can also obtain values from other Linux commands. To assign the result of Linux command to a variable, you first need to execute the command. If you place a Linux command within back quotes on the command line, that command is first executed and its result becomes an argument on the command line. In the case of assignments, the result of a command can be assigned to a variable by placing the command within back quotes to first execute it. The back quotes can be thought of as an expression consisting of a command to be executed whose result is then assigned to the variable. The characters making up the command itself are not assigned. In the next example, the command **ls *.c** is executed and its result is then assigned to the variable **listc**. **ls *.c** generates a list of all files with a **.c** extension. This list of files is then assigned to the **listc** variable.

$ **listc=`ls *.c`**
$ **echo $listc**
main.c prog.c lib.c

You need to keep in mind the difference between single quotes and back quotes. Single quotes treat a Linux command as a set of characters. Back quotes force execution of the Linux command. There may be times when you accidentally enter single quotes when you mean to use back quotes. In the following first example, the assignment for the **lscc** variable has single quotes, not back quotes, placed around the **ls *.c** command. In this case, **ls *.c** are just characters to be assigned to the variable **lscc**. In the second example, back quotes are placed around the **ls *.c** command, forcing evaluation of the command. A list of filenames ending in **.c** is generated and assigned as the value of **lscc**.

$ **lscc='ls *.c'**
$ **echo $lscc**
ls *.c
$ **lscc=`ls *.c`**

$ **echo $lscc**
main.c prog.c

Shell Scripts: User-Defined Commands

You can place shell commands within a file and then have the shell read and execute the commands in the file. In this sense, the file functions as a shell program, executing shell commands as if they were statements in a program. A file that contains shell commands is called a shell script.

You enter shell commands into a script file using a standard text editor such as the Vi editor. The **sh** or**.** command used with the script's filename will read the script file and execute the commands. In the next example, the text file called lsc contains an **ls** command that displays only files with the extension **.c**:

lsc
ls *.c

A run of the **lsc** script is shown here:

$ **sh lsc**
main.c calc.c
$ **. lsc**
main.c calc.c

Executing Scripts

You can dispense with the **sh** and. commands by setting the executable permission of a script file. When the script file is first created by your text editor, it is given only read and write permission. The **chmod** command with the **+x** option will give the script file executable permission. Once it is executable, entering the name of the script file at the shell prompt and pressing ENTER will execute the script file and the shell commands in it. In effect, the script's filename becomes a new shell command. In this way, you can use shell scripts to design and create your own Linux commands. You need to set the permission only once. In the next example, the **lsc** file's executable permission for the owner is set to on. Then the **lsc** shell script is directly executed like any Linux command.

$ **chmod u+x lsc**
$ **lsc**
main.c calc.c

You may have to specify that the script you are using is in your current working directory. You do this by prefixing the script name with a period and slash combination, **./**, as in **./lsc**. The period is a special character representing the name of your current working directory. The slash is a directory pathname separator. The following example shows how to execute the **lsc** script:

NOTES

$ **./lsc**
main.c calc.c

Script Arguments

Just as any Linux command can take arguments, so also can a shell script. Arguments on the command line are referenced sequentially starting with 1. An argument is referenced using the **$** operator and the number of its position. The first argument is referenced with **$1**, the second with **$2**, and so on. In the next example, the **lsext** script prints out files with a specified extension. The first argument is the extension. The script is then executed with the argument **c** (of course, the executable permission must have been set).

lsext ls *.$1

A run of the **lsext** script with an argument is shown here:

$ **lsext c**
main.c calc.c

In the next example, the commands to print out a file with line numbers have been placed in an executable file called **lpnum**, which takes a filename as its argument. The **cat** command with the **-n** option first outputs the contents of the file with line numbers. Then this output is piped into the **lpr** command, which prints it. The command to print out the line numbers is executed in the background.

lpnum
cat -n $1 | lpr &

A run of the **lpnum** script with an argument is shown here:

$ lpnum mydata

You may need to reference more than one argument at a time. The number of arguments used may vary. In **lpnum**, you may want to print out three files at one time and five files at some other time. The **$** operator with the asterisk, **$***, references all the arguments on the command line. Using **$*** enables you to create scripts that take a varying number of arguments. In the next example, **lpnum** is rewritten using **$*** so that it can take a different number of arguments each time you use it:

lpnum
cat -n $* | lpr &

A run of the **lpnum** script with multiple arguments is shown here:

$ **lpnum mydata preface**

TCSH Argument Array: argv

The TCSH/C shell uses a different set of argument variables to reference arguments. These are very similar to those used in the C programming language. When a TCSH shell script is invoked, all the words on the command line are parsed and placed in elements of an array called **argv.** The **argv[0]** array will hold the name of the shell script, and beginning with **argv[1]**, each element will hold an argument entered on the command line. In the case of shell scripts, **argv[0]** will always contain the name of the shell script. As with any array element, you can access the contents of an argument array element by preceding it with a **$** operator. For example, **$argv[1]** accesses the contents of the first element in the **argv** array, the first argument. In the **greetarg** script, a greeting is passed as the first argument on the command line. This first argument is accessed with **$argv[1]**.

Greetarg

echo "The greeting you entered was: $argv[1]"

A run of the **greetarg** script follows:

% **greetarg Hello**

The greeting you entered was: Hello

Each word is parsed on the command line unless it's quoted. In the next example, the **greetarg** script is invoked with an unquoted string and then a quoted string. Notice that the quoted string, "Hello, how are you", is treated as one argument.

% **greetarg Hello, how are you**

The greeting you entered was: Hello,

% **greetarg "Hello, how are you"**

The greeting you entered was: Hello, how are you

If more than one argument is entered, the arguments can each be referenced with a corresponding element in the **argv** array. In the next example, the **myargs** script prints out four arguments. Four arguments are then entered on the command line.

Myargs

#

echo "The first argument is: $argv[1]"

echo "The second argument is: $argv[2]"

echo "The third argument is: $argv[3]"

echo "The fourth argument is: $argv[4]"

The run of the **myargs** script is shown here:

% **myargs Hello Hi yo "How are you"**

The first argument is: Hello

The second argument is: Hi
The third argument is: yo
The fourth argument is: How are you

Environment Variables and Subshells: export and setenv

When you log in to your account, your Linux system generates your user shell. Within this shell, you can issue commands and declare variables. You can also create and execute shell scripts. However, when you execute a shell script, the system generates a subshell. You then have two shells, the one you logged in to and the one generated for the script. Within the script shell you can execute another shell script, which will then have its own shell. When a script has finished execution, its shell terminates and you enter back to the shell from which it was executed. In this sense, you can have many shells, each nested within the other.

Variables that you define within a shell are local to it. If you define a variable in a shell script, then, when the script is run, the variable is defined with that script's shell and is local to it. No other shell can reference it. In a sense, the variable is hidden within its shell.

To illustrate this situation more clearly, the next example will use two scripts, one of which is called from within the other. When the first script executes, it generates its own shell. From within this shell, another script is executed which, in turn, generates its own shell. In the next example, the user first executes the dispfirst script, which displays a first name. When the dispfirst script executes, it generates its own shell and then, within that shell, it defines the firstname variable. After it displays the contents of firstname, the script executes another script: displast. When displast executes, it generates its own shell. It defines the lastname variable within its shell and then displays the contents of lastname. It then tries to reference firstname and display its contents. It cannot do so because firstname is local to the dispfirst shell and cannot be referenced outside it. An error message is displayed indicating that for the displast shell, firstname is an undefined variable.

dispfirst firstname="Charles"
echo "First name is $firstname"
displast
displast lastname="Dickens"
echo "Last name is $lastname"

echo "$firstname $lastname"

The run of the **dispfirst** script is shown here:

$ dispfirst
First name is Charles
Last name is Dickens
Dickens
sh: firstname: not found
$
dispfile
myfile="List"
echo "Displaying $myfile"
pr -t -n $myfile
printfile
printfile
myfile="List"
echo "Printing $myfile"
lp $myfile &

The run of the **dispfile** script is shown here:

$ **dispfile**
Displaying List
1 screen
2 modem
3 paper
Printing List
$

If you want the same value of a variable used both in a script's shell and a subshell, you can simply define the variable twice, once in each script, and assign it the same value. In the previous example, there is a myfile variable defined in dispfile and in printfile. The user executes the b script, which first displays the list file with line numbers. When the dispfile script executes, it generates its own shell and then, within that shell, it defines the myfile variable. After it displays the contents of the file, the script then executes another script, printfile. When printfile executes, it generates its own shell. It defines its own myfile variable within its shell and then sends a file to the printer.

What if you want to define a variable in one shell and have its value referenced in any subshell? For example, what if you want to define the myfile variable in the dispfile script and have its value, List, referenced from within the printfile script, rather than explicitly defining another variable in printfile? Since variables are local to the shell they are defined

in, there is no way you can do this with ordinary variables. However, there is a type of variable called an environment variable that allows its value to be referenced by any subshell. Environment variables constitute an environment for the shell and any subshell it generates, no matter how deeply nested.

You can define environment variables in the three major types of shells: Bourne, Korn, and C. However, the strategy used to implement environmental variables in the Bourne and Korn shells is very different from that of the C shell. In the Bourne and Korn shells, environmental variables are exported. That is to say, a copy of an environmental variable is made in each subshell. In a sense, if the **myfile** variable is exported, a copy is automatically defined in each subshell for you. In the C shell, on the other hand, an environmental variable is defined only once and can be directly referenced by any subshell.

Shell Environment Variables

In the Bourne, BASH, and Korn shells, an environment variable can be thought of as a regular variable with added capabilities. To make an environment variable, you apply the **export** command to a variable you have already defined. The **export** command instructs the system to define a copy of that variable for each new shell generated. Each new shell will have its own copy of the environment variable. This process is called exporting variables.

In the next example, the variable **myfile** is defined in the **dispfile** script. It is then turned into an environment variable using the **export** command. The **myfile** variable will consequently be exported to any subshell, such as that generated when **printfile** is executed.

```
dispfile
myfile="List"
export myfile
echo "Displaying $myfile"
pr -t -n $myfile
printfile
printfile
echo "Printing $myfile"
lp $myfile &
```

The run of the **dispfile** script is shown here:

```
$ dispfile
Displaying List
1 screen
2 modem
```

3 paper
Printing List
$

When **printfile** is executed it will be given its own copy of **myfile** and can reference that copy within its own shell. You no longer need to explicitly define another **myfile** variable in **printfile**. It is a mistake to think of exported environment variables as global variables. A new shell can never reference a variable outside of itself. Instead, a copy of the variable with its value is generated for the new shell. You can think of exported variables as exporting their values to a shell, not themselves. For those familiar with programming structures, exported variables can be thought of as a form of call-by-value.

TCSH and C Shell Environment Variables

In the TCSH and C shells, an environment variable is defined using a separate definition command, **setenv**. In this respect, an environment variable is really a very different type of variable from that of a regular local variable. A C shell environment variable operates more like a global variable. It can be directly referenced by any subshell. This differs from the Bourne, BASH, and Korn shells in which only a copy of the environment variable is passed down and used by the subshell.

To define an environment variable you first enter the **setenv** command followed by the variable name and then the value. There is no assignment operator. In the next example, the **myfile** environment variable is defined and assigned the value **List**.

% **setenv myfile list**
dispfile
setenv myfile "List"
echo "Displaying $myfile"
cat -n $myfile
printfile
printfile
echo "Printing $myfile"
lpr $myfile &

The run of the **dispfile** script is shown here:

% **dispfile**
Displaying List
1 screen
2 modem
3 paper
Printing List

$

In the previous example, the variable **myfile** is defined as an environment variable in the **dispfile** script. Notice the use of the **setenv** command instead of **set**. The **myfile** variable can now be referenced in any subshell, such as that generated when **printfile** is executed.

When **printfile** is executed, it will be able to directly access the **myfile** variable defined in the shell of the **dispfile** script.

Control Structures

You can control the execution of Linux commands in a shell script with control structures. Control structures allow you to repeat commands and to select certain commands over others. A control structure consists of two major components: a test and commands. If the test is successful, then the commands are executed. In this way, you can use control structures to make decisions as to whether commands should be executed.

There are two different kinds of control structures: loops and conditions. A loop repeats commands, whereas a condition executes a command when certain conditions are met. The BASH shell has three loop control structures: **while**, **for**, and **for-in**. There are two condition structures: **if** and **case**. The control structures have as their test the execution of a Linux command. All Linux commands return an exit status after they have finished executing. If a command is successful, its exit status will be 0. If the command fails for any reason, its exit status will be a positive value referencing the type of failure that occurred. The control structures check to see if the exit status of a Linux command is 0 or some other value. In the case of the **if** and **while** structures, if the exit status is a 0 value, then the command was successful and the structure continues.

Test Operations

With the **test** command, you can compare integers, compare strings, and even perform logical operations. The command consists of the keyword **test** followed by the values being compared, separated by an option that specifies what kind of comparison is taking place. The option can be thought of as the operator, but it is written, like other options, with a minus sign and letter codes. For example, **-eq** is the option that represents the equality comparison. However, there are two string operations that actually use an operator instead of an option. When you compare two strings for equality, you use the equal sign (=). For inequality you use **!=**. Table 4-1

lists some of the commonly used options and operators used by **test**. The syntax for the **test** command is shown here:

test value -option value

test string = string

In the next example, the user compares two integer values to see if they are equal. In this case, you need to use the equality option, **-eq**. The exit status of the **test** command is examined to find out the result of the test operation. The shell special variable **$?** holds the exit status of the most recently executed Linux command.

$ **num=5**

$ **test $num -eq 10**

$ **echo $?**

1

Instead of using the keyword **test** for the **test** command, you can use enclosing brackets. The command **test $greeting = "hi"** can be written as

$ **[$greeting = "hi"]**

Similarly, the test command **test $num -eq 10** can be written as

$ **[$num -eq 10]**

Integer Comparisons	Function
-gt	Greater-than
-lt	Less-than
-ge	Greater-than-or-equal-to
-le	Less-than-or-equal-to
-eq	Equal
-ne	Not-equal
String Comparisons	
-z	Tests for empty string
=	Tests for equality of strings
!=	Tests for inequality of strings
Logical Operators	
-a	Logical AND
-o	Logical OR
!	Logical NOT
File Tests	
-f	File exists and is a regular file
-s	File is not empty
-r	File is readable
-w	File can be written to and modified
-x	File is executable
-d	Filename is a directory name

The brackets themselves must be surrounded by white space: a space, TAB, or ENTER. Without the spaces, they are invalid.

Conditional Control Structures

The BASH shell has a set of conditional control structures that allow you to choose what Linux commands to execute. Many of these are similar to conditional control structures found in programming languages, but there are some differences. The **if** condition tests the success of a Linux command, not an expression. Furthermore, the end of an **if-then** command must be indicated with the keyword **fi**, and the end of a **case** command is indicated with the keyword **esac**.

The **if** structure places a condition on commands. That condition is the exit status of a specific Linux command. If a command is successful, returning an exit status of 0, then the commands within the **if** structure are executed. If the exit status is anything other than 0,

Condition Control Structures: if, else, elif, case	**Function**
`if` *command* `then` *command* `fi`	`if` executes an action if its test command is true.
`if` *command* `then` *command* `else` *command* `fi`	`if-else` executes an action if the exit status of its test command is true; if false, then the `else` action is executed.
`if` *command* `then` *command* `elif` *command* `then` *command* `else` *command* `fi`	`elif` allows you to nest `if` structures, enabling selection among several alternatives; at the first true `if` structure, its commands are executed and control leaves the entire `elif` structure.
`case` *string* `in` *pattern)* *command;;* `esac`	`case` matches the string value to any of several patterns; if a pattern is matched, its associated commands are executed.
command && *command*	The logical AND condition returns a true 0 value if both commands return a true 0 value; if one returns a nonzero value, then the AND condition is false and also returns a nonzero value.
command || *command*	The logical OR condition returns a true 0 value if one or the other command returns a true 0 value; if both commands return a nonzero value, then the OR condition is false and also returns a nonzero value.
! *command*	The logical NOT condition inverts the return value of the command.

Loop Control Structures: while, until, for, for-in, select	
`while` *command* `do` *command* `done`	`while` executes an action as long as its test command is true.
`until` *command* `do` *command* `done`	`until` executes an action as long as its test command is false.

Loop Control Structures: while, until, for, for-in, select	
for *variable* in *list-values* do *command* done	for-in is designed for use with lists of values; the variable operand is consecutively assigned the values in the list.
for *variable* do *command* done	for is designed for reference script arguments; the variable operand is consecutively assigned each argument value.
select *string* in *item-list* do *command* done	select creates a menu based on the items in the *item-list*; then it executes the command; the command is usually a case.

then the command has failed and the commands within the **if** structure are not executed. The **if** command begins with the keyword **if** and is followed by a Linux command whose exit condition will be evaluated. The keyword **fi** ends the command. The **elsels** script in the next example executes the **ls** command to list files with two different possible options, either by size or with all file information. If the user enters an **s**, files are listed by size; otherwise, all file information is listed.

```
elsels
echo Enter s to list file by sizes
echo otherwise all file information is listed.
echo -n "Please enter option: "
read choice
if [ "$choice" = s ]
then
ls -s
else
ls –l
fi
echo Good-bye
```

A run of the program follows:

```
$ elsels
Enter s to list file sizes,
otherwise all file information is listed.
Please enter option: s
total 2
1 monday 2 today
$
```

Loop Control Structures

The **while** loop repeats commands. A **while** loop begins with the keyword **while** and is followed by a Linux command. The keyword **do** follows on

the next line. The end of the loop is specified by the keyword **done**. The Linux command used in **while** structures is often a test command indicated by enclosing brackets.

The **for-in** structure is designed to reference a list of values sequentially. It takes two operands: a variable and a list of values. The values in the list are assigned one by one to the variable in the **for-in** structure. Like the **while** command, the **for-in** structure is a loop. Each time through the loop, the next value in the list is assigned to the variable. When the end of the list is reached, the loop stops. Like the **while** loop, the body of a **for-in** loop begins with the keyword **do** and ends with the keyword **done**. The **cbackup** script makes a backup of each file and places it in a directory called **sourcebak**. Notice the use of the ***** special character to generate a list of all filenames with a **.c** extension.

cbackup
for backfile in *.c
do
cp $backfile sourcebak/$backfile
echo $backfile
done A run of the program follows:

$ **cbackup**
io.c
lib.c
main.c
$

The **for** structure without a specified list of values takes as its list of values the command line arguments. The arguments specified on the command line when the shell file is invoked become a list of values referenced by the **for** command. The variable used in the **for** command is set automatically to each argument value in sequence. The first time through the loop, the variable is set to the value of the first argument. The second time, it is set to the value of the second argument.

TCSH/C Shell Control Structures

As in other shells, the TCSH shell has a set of control structures that let you control the execution of commands in a script. There are loop and conditional control structures with which you can repeat Linux commands or make decisions about which commands you want to execute. The **while** and **if** control structures are more general purpose control structures, performing iterations and making decisions using a variety of different tests. The **switch** and **foreach** control structures are more specialized operations. The **switch** structure is a restricted form of the **if** condition that

checks to see if a value is equal to one of a set of possible values. The **foreach** structure is a limited type of loop that runs through a list of values, assigning a new value to a variable with each iteration.

The TCSH shell differs from other shells in that its control structures conform more to a programming-language format. The test condition for a TCSH shell control structure is an expression that evaluates to true or false, not a Linux command. One key difference between BASH shell and TCSH shell control structures is that TCSH shell structures cannot redirect or pipe their output. They are strictly control structures, controlling the execution of commands.

Test Expressions

The **if** and **while** control structures use an expression as their test. A true test is any expression that results in a nonzero value. A false test is any expression that results in a 0 value. In the TCSH shell, relational and equality expressions can be easily used as test expressions, because they result in 1 if true and 0 if false. There are many possible operators that you can use in an expression.The test expression can also be arithmetic or a string comparison, but strings can only be compared for equality or inequality.

Unlike the BASH shell, you must enclose the TCSH shell **if** and **while** test expressions within parentheses. The next example shows a simple test expression testing to see if two strings are equal.

```
if ( $greeting == "hi" ) then
echo Informal Greeting
endif
```

The TCSH shell has a separate set of operators for testing strings against other strings or against regular expressions. The **==** and **!=** operators test for the equality and inequality of strings. The **=~** and **!~** operators test a string against a regular expression and test to see if a pattern match is successful. The regular expression can contain any of the shell special characters. In the next example, any value of **greeting** that begins with an upper or lowercase **h** will match the regular expression **[Hh]***.

```
 if ( $greeting =~ [Hh]* ) then
echo Informal Greeting
endif
```

Like the BASH shell, the TCSH shell has several special operators that test the status of files. Many of these operators are the same. In the next example, the **if** command tests to see if the file **mydata** is readable.

```
 if ( -r mydata ) then
 echo Informal Greeting
```

endif

TCSH Shell Conditions: if-then, if-then-else, switch

The TCSH shell has a set of conditional control structures with which you make decisions about what Linux commands to execute. Many of these conditional control structures are similar to conditional control structures found in the BASH shell. There are, however, some key differences. The TCSH shell **if** structure ends with the keyword **endif**. The **switch** structure uses the keyword **case** differently. It ends with the keyword **endsw** and uses the

String Comparisons	Function/Description
==	Tests for equality of strings
!=	Tests for inequality of strings
=~	Compares string to a pattern to test if equal; the pattern can be any regular expression
!~	Compares string to a pattern to test if not equal; the pattern can be any regular expression
Logical Operators	
&&	Logical AND
||	Logical OR
!	Logical NOT
File Tests	
-e	File exists
-r	File is readable
-w	File can be written to, modified
-x	File is executable
-d	Filename is a directory name
-f	File is an ordinary file
-o	File is owned by user
-z	File is empty

Relational Operators	
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!=	Not equal
==	Equal

Keyword breaksw instead of two semicolons.Furthermore, there are two if control.

structures: a simple version that executes only one command and a more complex version that can execute several commands as well as alternative commands. The simple version of **if** consists of the keyword **if** followed by a test and a single Linux command. The complex version ends with the keyword **endif**.

Control Structures	Description
`if`(*expression*) `then` *commands* `endif`	If the expression is true, the following commands are executed. You can specify more than one Linux command.
`if`(*expression*) `then` *command* `else` *command* `endif`	If the expression is true, the command after `then` is executed. If the expression is false, the command following `else` is executed.
`switch`(*string*) `case` *pattern*: *command* `breaksw` `default`: *command* `endsw`	Allows you to choose among several alternative commands.

The if-then Structure

The **if-then** structure places a condition on several Linux commands. That condition is an expression. If the expression results in a value other than 0, the expression is true and the commands within the **if** structure are executed. If the expression results in a 0 value, the expression is false and the commands within the **if** structure are not executed.

The **if-then** structure begins with the keyword **if** and is followed by an expression enclosed in parentheses. The keyword then follows the expression. You can then specify any number of Linux commands on the following lines. The keyword **endif** ends the **if** command. Notice that, whereas in the BASH shell the **then** keyword is on a line of its own, in the TCSH shell, **then** is on the same line as the test expression. The syntax for the **if-then** structure is shown here:

if (Expression **) then**
Commands
endif

The **ifls** script shown next allows you to list files by size. If you enter an **s** at the prompt, each file in the current directory is listed, followed by the number of blocks it uses. If you enter anything else at the prompt, the **if** test fails and the script does nothing.

Ifls
#
echo -n "Please enter option: "
set option = $<
if ($option == "s")
then
echo Listing files by size
ls –s
endif

A run of the **ifls** script is shown here:

% **ifls**

Please enter option: s

Listing files by size

total 2

1 monday 2 today

Often, you need to choose between two alternatives based on whether an expression is true. The **else** keyword allows an **if** structure to choose between two alternative commands. If the expression is true, those commands immediately following the test expression are executed. If the expression is false, those commands following the **else** keyword are executed. The syntax for the **if-else** command is shown here:

```
if ( expression ) then
 commands
else
commands
endif
```

The **elsels** script in the next example executes the **ls** command to list files with two different possible options: by size or with all file information. If the user enters an **s**, files are listed by size; otherwise, all file information is listed.

```
elsels
 #
echo Enter s to list file sizes.
echo otherwise all file information is listed.
echo -n "Please enter option : "
set option = $<
if ($option == "s") then
 ls -s
else
 ls –l
 endif
echo Goodbye
```

A run of the **elsels** script follows:

> **elsels**

Enter s to list file sizes,

otherwise all file information is listed.

Please enter option: s

total 2

1 monday 2 today

Good-bye

The switch Structure

The **switch** structure chooses among several possible alternative commands. It is similar to the BASH shell's case structure in that the choice is made by comparing a string with several possible patterns. Each possible pattern is associated with a set of commands. If a match is found, the associated commands are performed.

The **switch** structure begins with the keyword **switch** followed by a test string within parentheses. The string is often derived from a variable evaluation. A set of patterns then follows—each pattern preceded by the keyword **case** and terminated with a colon. Commands associated with this choice are listed after the colon. The commands are terminated with the keyword **breaksw**. After all the listed patterns, the keyword **endsw** ends the switch structure. The syntax for the switch structure is shown here:

switch (test-string**)**
case pattern:
commands
breaksw
case pattern:
commands
breaksw
default:
commands
breaksw
endsw

TCSH Shell Loops: while, foreach, repeat

The TCSH shell has a set of loop control structures that allow you to repeat Linux commands: **while**, **foreach**, and **repeat**.

The **while** structure operates in a way similar to corresponding structures found in programming languages. Like the TCSH shell's **if** structure, the **while** structure tests the result of an expression. The TCSH shell's **foreach** structure, like the **for** and **for-in** structures in the BASH shell, does not perform any tests. It simply progresses through a list of values, assigning each value in turn to a specified variable. In this respect, the **foreach** structure is very different from corresponding structures found in programming languages. The **repeat** structure is a simple and limited control structure. It repeats one command a specified number of times. It has no test expression, and it cannot repeat more than one command.

Loop Control Structures	Description
`while`(*expression*) *command* `end`	Executes commands as long as the expression is true.
`foreach` *variable (arg-list)* *command* `end`	Iterates the loop for as many arguments as exist in the argument list. Each time through the loop, the variable is set to the next argument in the list; operates like `for-in` in the BASH shell.
`repeat` *num command*	Repeats a command the specified number of times.
`continue`	Jumps to next iteration, skipping the remainder of the loop commands.
`break`	Breaks out of a loop.

The while Structure

The while loop repeats commands. A while loop begins with the keyword while and is followed by an expression enclosed in parentheses. The end of the loop is specified by the keyword **end**. The syntax for the while loop is shown here:

while (expression **)**
commands
end

The **while** structure can easily be combined with a **switch** structure to drive a menu.

The foreach Structure

The **foreach** structure is designed to sequentially reference a list of values. It is very similar to the BASH shell's **for-in** structure. The **foreach** structure takes two operands: a variable and a list of values enclosed in parentheses. Each value in the list is assigned to the variable in the **foreach** structure. Like the **while** structure, the **foreach** structure is a loop. Each time through the loop, the next value in the list is assigned to the variable. When the end of the list is reached, the loop stops. Like the **while** loop, the body of a **foreach** loop ends with the keyword **end**. The syntax for the **foreach** loop is shown here:

foreach variable **(** list of values **)**
commands
end

In the **mylist** script, in the next example, the script simply outputs a list of each item with today's date. The list of items makes up the list of values read by the **foreach** loop. Each item is consecutively assigned to the variable **grocery**.

mylist

```
#
set tdate=`date '+%D'`
foreach grocery ( milk cookies apples cheese )
```

```
 echo "$grocery $tdate"
end $
mylist
milk 12/23/96
cookies 12/23/96
apples 12/23/96
cheese 12/23/96
$
```

The **foreach** loop is useful for managing files. In the **foreach** structure, you can use shell special characters in a pattern to generate a list of filenames for use as your list of values. This generated list of filenames then becomes the list referenced by the **foreach** structure. An asterisk by itself generates a list of all files and directories. ***.c*** lists files with the **.c** extension. These are usually C source code files. The next example makes a backup of each file and places the backup in a directory called **sourcebak**. The pattern ***.c*** generates a list of filenames that the **foreach** structure can operate on.

```
cbackup
#
foreach backfile ( *.c )
cp $backfile sourcebak/$backfile
 echo $backfile
end
 % cbackup
io.c
lib.c
main.c
```

The **foreach** structure without a specified list of values takes as its list of values the command line arguments. The arguments specified on the command line when the shell file was invoked become a list of values referenced by the **foreach** structure. The variable used in the **foreach** structure is set automatically to each argument value in sequence. The first time through the loop, the variable is set to the value of the first argument. The second time, it is set to the value of the second argument, and so on.

In the **mylistarg** script in the next example, there is no list of values specified in the **foreach** loop. Instead, the **foreach** loop consecutively reads the values of command line arguments into the **grocery** variable. When all the arguments have been read, the loop ends.

```
Mylistarg
 #
```

```
set tdate=`date '+%D'`
foreach grocery ( $argv[*] )
 echo "$grocery $tdate"
 end
$ mylistarg milk cookies apples cheese
milk 12/23/96
 cookies 12/23/96
apples 12/23/96
 cheese 12/23/96
 $
```

Review & Self Assessment Question:

Q1- What is Shell Script ?
Q2- What do you mean by scripts arguments?
Q3-What do you mean by TCSH argument array ?
Q4-Explain Test Comman briefly ?
Q5-What is switch structure ?

Further Readings

Linux Operating System Richard Petersen
Linux Operating System Paul S. Wang
Linux Operating System by David Maxwell and Andrew Bedford
Linux Operating System by Richard Blum and Christine Bresnahan
Linux Operating System by Bhatt P.C.P

UNIT- 5 SHELL CONFIGURATION

Contents

Introduction

Four different major shells are commonly used on Linux systems: the Bourne Again shell (BASH), the AT&T Korn shell, the TCSH shell, and the Z shell. The BASH shell is an advanced version of the Bourne shell, which includes most of the advanced features developed for the Korn shell and the C shell. TCSH is an enhanced version of the C shell, originally developed for BSD versions of Unix. The AT&T Unix Korn shell is open source. The Z shell is an enhanced version of the Korn shell. Although their Unix counterparts differ greatly, the Linux shells share many of the same features. In Unix, the Bourne shell lacks many capabilities found in the other Unix shells. In Linux, however, the BASH shell incorporates all the advanced features of the Korn shell and C shell, as well as the TCSH shell. All four shells are available for your use, though the BASH shell is the default.

The BASH shell is the default shell for most Linux distributions. If you are logging in to a command line interface, you will be placed in the default shell automatically and given a shell prompt at which to enter your commands. The shell prompt for the BASH shell is a dollar sign (**$**). In a GUI interface, such as GNOME or KDE, you can open a terminal window that will display a command line interface with the prompt for the default shell (BASH). Though you log in to your default shell or display it automatically in a terminal window, you can change to another shell by entering its name. **tcsh** invokes the TCSH shell, **bash** the

BASH shell, **ksh** the Korn shell, and **zsh** the Z shell. You can leave a shell by pressing CTRL-D or using the **exit** command. You only need one type of shell to do your work. Table 5-1 shows the different commands you can use to invoke different shells. Some shells have added links you can use the invoke the same shell, like **sh** and **bsh**, which link to and invoke the **bash** command for the BASH shell.

This chapter describes common features of the BASH shell, such as aliases, as well as how to configure the shell to your own needs using shell variables and initialization files. The other shells share many of the same features and use similar variables and initialization files.

Though the basic shell features and configurations are shown here, you should consult the respective online manuals and FAQs for each shell for more detailed examples and explanations.

Shells	Description
bash	BASH shell, **/bin/bash**
bsh	BASH shell, **/bin/bsh** (link to **/bin/bash**)
sh	BASH shell, **/bin/sh** (link to **/bin/bash**)
tcsh	TCSH shell, **/usr/tcsh**
csh	TCSH shell, **/bin/csh** (link to **/bin/tcsh**)
ksh	Korn shell, **/bin/ksh** (also added link **/usr/bin/ksh**)
zsh	Z shell, **/bin/zsh**

Shell Initialization and Configuration Files

Each type of shell has its own set of initialization and configuration files. The BASH shell configuration files were discussed previously. The TCSH shell uses .login, .tcshrc, and .logout files in place of .bash_profile, .bashrc, and .bash_logout. Instead of .bash_profile, some distributions use the name .profile. The Z shell has several initialization files: .zshenv, .zlogin, .zprofile, .zschrc, and .zlogout.Check the Man pages for each shell to see how they are usually configured. When you install a shell, default versions of these files are automatically placed in the users' home directories. Except for the TCSH shell, all shells use much the same syntax for variable definitions and assigning values.

Configuration Directories and Files

Applications often install configuration files in a user's home directory that contain specific configuration information, which tailors the application to the needs of that particular user. This may take the form of a single configuration file that begins with a period, or a directory that contains several configuration files. The directory name will also begin with a period. For example, Mozilla installs a directory called **.mozilla** in the user's home directory that contains configuration files. On the other hand, many mail application uses a single file called **.mailrc** to hold alias and

feature settings set up by the user, though others like Evolution also have their own, **.evolution**. Most single configuration files end in the letters **rc**. **FTP** uses a file called **.netrc**. Most newsreaders use a file called **.newsrc**. Entries in configuration files are usually set by the application, though you can usually make entries directly by editing the file. Applications have their own set of special variables to which you can define and assign values. You can list the configuration files in your home directory with the **ls -a** command.

Filename	Function
BASH Shell	
.bash_profile	Login initialization file
.profile	Login initialization file (same as .bash_profile)
.bashrc	BASH shell configuration file
.bash_logout	Logout name
.bash_history	History file
/etc/profile	System login initialization file
/etc/bashrc	System BASH shell configuration file
/etc/profile.d	Directory for specialized BASH shell configuration files
TCSH Shell	
.login	Login initialization file
.tcshrc	TCSH shell configuration file
.logout	Logout file
Z Shell	
.zshenv	Shell login file (first read)
.zprofile	Login initialization file
.zlogin	Shell login file
.zshrc	Z shell configuration file
.zlogout	Logout file
Korn Shell	
.profile	Login initialization file
.kshrc	Korn shell configuration file

Aliases

You use the **alias** command to create another name for a command. The **alias** command operates like a macro that expands to the command it represents. The alias does not literally replace the name of the command; it simply gives another name to that command. An **alias** command begins with the keyword **alias** and the new name for the command, followed by an equal sign and the command the alias will reference.

In the next example, **list** becomes another name for the **ls** command:

$ **alias list=ls**

$ **ls**

mydata today

$ **list**

mydata today

$

Aliasing Commands and Options

You can also use an alias to substitute for a command and its option, but you need to enclose both the command and the option within single quotes. Any command you alias that contains spaces must be enclosed in single quotes as well. In the next example, the alias **lss** references the **ls** command with its **-s** option, and the alias **lsa** references the **ls** command with the **-F** option. The **ls** command with the **-s** option lists files and their sizes in blocks, and **ls** with the **-F** option places a slash after directory names. Notice how single quotes enclose the command and its option.

$ **alias lss='ls -s'**
$ **lss**
mydata 14 today 6 reports 1
$ **alias lsa='ls -F'**
$ **lsa**
mydata today reports/
$

Aliases are helpful for simplifying complex operations. In the next example, **listlong** becomes another name for the **ls** command with the **-l** option (the long format that lists all file information), as well as the **-h** option for using a human-readable format for file sizes. Be sure to encase the command and its arguments within single quotes so that they are taken as one argument and not parsed by the shell.

$ **alias listlong='ls -lh'**
$ **listlong**
-rw-r--r-- 1 root root 51K Sep 18 2003 mydata
-rw-r--r-- 1 root root 16K Sep 27 2003 today

Aliasing Commands and Arguments

You may often use an alias to include a command name with an argument. If you execute a command that has an argument with a complex combination of special characters on a regular basis, you may want to alias it. For example, suppose you often list just your source code and object code files—those files ending in either a **.c** or **.o**. You would need to use as an argument for **ls** a combination of special characters such as ***.[co]**. Instead, you can alias **ls** with the **.[co]** argument, giving it a simple name. In the next example, the user creates an alias called **lsc** for the command **ls.[co]**:

$ **alias lsc='ls *.[co]'**

```
$ lsc
main.c main.o lib.c lib.o
```

Aliasing Commands

You can also use the name of a command as an alias. This can be helpful in cases in which you should use a command only with a specific option. In the case of the **rm**, **cp**, and **mv** commands, the **-i** option should always be used to ensure an existing file is not overwritten. Instead of always being careful to use the **-i** option each time you use one of these commands; you can alias the command name to include the option. In the next example, the **rm**, **cp**, and **mv** commands have been aliased to include the **-i** option:

```
$ alias rm='rm -i'
$ alias mv='mv -i'
$ alias cp='cp -i'
```

The **alias** command by itself provides a list of all aliases that have been defined, showing the commands they represent. You can remove an alias by using the **unalias** command. In the next example, the user lists the current aliases and then removes the **lsa** alias:

```
$ alias
lsa=ls -F
list=ls
rm=rm -i
$ unalias lsa
```

Controlling Shell Operations

The BASH shell has several features that enable you to control the way different shell operations work. For example, setting the **noclobber** feature prevents redirection from overwriting files. You can turn these features on and off like a toggle, using the **set** command. The **set** command takes two arguments: an option specifying on or off and the name of the feature. To set a feature on, you use the **-o** option, and to set it off, you use the **+o** option. Here is the basic form:

```
$ set -o feature turn the feature on
$ set +o feature turn the feature off
```

Three of the most common features are **ignoreeof**, **noclobber**, and **noglob**. Table 5-3 lists these different features, as well as the **set** command. Setting **ignoreeof** enables a feature that prevents you from logging out of the user shell with CTRL-D. CTRL-D is not only

Features	Description
`$ set -+o` *feature*	BASH shell features are turned on and off with the `set` command; `-o` sets a feature on and `+o` turns it off: `$ set -o noclobber` *set noclobber on* `$ set +o noclobber` *set noclobber off*
`ignoreeof`	Disables CTRL-D logout
`noclobber`	Does not overwrite files through redirection
`noglob`	Disables special characters used for filename expansion: `*`, `?`, `-`, and `[]`

used to log out of the user shell, but also to end user input entered directly into the standard input. CTRL-D is used often for the Mail program or for utilities such as **cat**. You can easily enter an extra CTRL-D in such circumstances and accidentally log yourself out. The **ignoreeof** feature prevents such accidental logouts. In the next example, the **ignoreeof** feature is turned on using the **set** command with the **-o** option. The user can then log out only by entering the **logout** command.

$ **set -o ignoreeof**
$ CTRL-D
Use exit to logout
$

Environment Variables and Subshells: export

When you log in to your account, Linux generates your user shell. Within this shell, you can issue commands and declare variables. You can also create and execute shell scripts. When you execute a shell script, however, the system generates a subshell. You then have two shells, the one you logged in to and the one generated for the script. Within the script shell, you can execute another shell script, which then has its own shell. When a script has finished execution, its shell terminates and you return to the shell from which it was executed. In this sense, you can have many shells, each nested within the other. Variables you define within a shell are local to it. If you define a variable in a shell script, then, when the script is run, the variable is defined with that script's shell and is local to it. No other shell can reference that variable. In a sense, the variable is hidden within its shell.

You can define environment variables in all types of shells, including the BASH shell, the Z shell, and the TCSH shell. The strategy used to implement environment variables in the BASH shell, however, is different from that of the TCSH shell. In the BASH shell, environment variables are exported. That is to say, a copy of an environment variable is made in each subshell. For example, if the **EDITOR** variable is exported, a copy is automatically defined in each subshell for you. In the TCSH shell, on the

other hand, an environment variable is defined only once and can be directly referenced by any subshell.

In the BASH shell, an environment variable can be thought of as a regular variable with added capabilities. To make an environment variable, you apply the **export** command to a variable you have already defined. The **export** command instructs the system to define a copy of that variable for each new shell generated. Each new shell will have its own copy of the environment variable. This process is called exporting variables. To think of exported environment variables as global variables is a mistake. A new shell can never reference a variable outside of itself. Instead, a copy of the variable with its value is generated for the new shell.

Configuring Your Shell with Shell Parameters

When you log in, Linux will set certain parameters for your login shell. These parameters can take the form of variables or features. See the earlier section "Controlling Shell Operations" for a description of how to set features. Linux reserves a predefined set of variables for shell and system use. These are assigned system values, in effect setting parameters. Linux sets up parameter shell variables you can use to configure your user shell. Many of these parameter shell variables are defined by the system when you log in. Some parameter shell variables are set by the shell automatically, and others are set by initialization scripts, described later. Certain shell variables are set directly by the shell, and others are simply used by it. Many of these other variables are application specific, used for such tasks as mail, history, or editing. Functionally, it may be better to think of these as system-level variables, as they are used to configure your entire system, setting values such as the location of executable commands on your system, or the number of history commands allowable. See Table 5-4 for a list of those shell variables set by the shell for shell-specific tasks; Table 5-5 lists those used by the shell for supporting other applications.

A reserved set of keywords is used for the names of these system variables. You should not use these keywords as the names of any of your own variable names. The system shell variables are all specified in uppercase letters, making them easy to identify. Shell feature variables are in lowercase letters. For example, the keyword HOME is used by the system to define the HOME variable. HOME is a special environment variable that holds the pathname of the user's home directory. On the other hand, the keyword noclobber is used to set the noclobber feature on or off.

Shell Parameter Variables

Many of the shell parameter variables automatically defined and assigned initial values by the system when you log in can be changed, if you wish.

However, some parameter variables exist whose values should not be changed. For example, the **HOME** variable holds the

Shell Variables	Description
BASH	Holds full pathname of BASH command
BASH_VERSION	Displays the current BASH version number
GROUPS	Groups that the user belongs to
HISTCMD	Number of the current command in the history list
HOME	Pathname for user's home directory
HOSTNAME	The hostname
HOSTTYPE	Displays the type of machine the host runs on
OLDPWD	Previous working directory
OSTYPE	Operating system in use
PATH	List of pathnames for directories searched for executable commands
PPID	Process ID for shell's parent shell
PWD	User's working directory
RANDOM	Generates random number when referenced
SHLVL	Current shell level, number of shells invoked
UID	User ID of the current user

Shell Variables	Description
BASH_VERSION	Displays the current BASH version number
CDPATH	Search path for the **cd** command
EXINIT	Initialization commands for Ex/Vi editor
FCEDIT	Editor used by the history **fc** command.
GROUPS	Groups that the user belongs to
HISTFILE	The pathname of the history file
HISTSIZE	Number of commands allowed for history
HISTFILESIZE	Size of the history file in lines
HISTCMD	Number of the current command in the history list
HOME	Pathname for user's home directory
HOSTFILE	Sets the name of the hosts file, if other than **/etc/hosts**
IFS	Interfield delimiter symbol
IGNOREEOF	If not set, EOF character will close the shell. Can be set to the number of EOF characters to ignore before accepting one to close the shell (default is 10)
INPUTRC	Set the **inputrc** configuration file for Readline (command line). Default is current directory, **.inputrc**. Most Linux distributions set this to **/etc/inputrc**
KDEDIR	The pathname location for the KDE desktop
LOGNAME	Login name
MAIL	Name of specific mail file checked by Mail utility for received messages, if MAILPATH is not set
MAILCHECK	Interval for checking for received mail
MAILPATH	List of mail files to be checked by Mail for received messages
HOSTTYPE	Linux platforms, such as I686, x86_64, or ppc
PROMPT_COMMAND	Command to be executed before each prompt, integrating the result as part of the prompt

HISTFILE	The pathname of the history file
PS1	Primary shell prompt
PS2	Secondary shell prompt
QTDIR	Location of the Qt library (used for KDE)
SHELL	Pathname of program for type of shell you are using
TERM	Terminal type
TMOUT	Time that the shell remains active awaiting input
USER	Username
UID	Real user ID (numeric)
EUID	Effective user ID (EUID, numeric). This is usually the same as the UID but can be different when the user changes IDs, as with the su command, which allows a user to become an effective root user

pathname for your home directory. Commands such as **cd** reference the pathname in the **HOME** shell variable to locate your home directory. Some of the more common of these parameter variables are described in this section. Other parameter variables are defined by the system and given an initial value that you are free to change. To do this, you redefine them and assign a new value. For example, the PATH variable is defined by the system and given an initial value; it contains the pathnames of directories where commands are located. Whenever you execute a command, the shell searches for it in these directories. You can add a new directory to be searched by redefining the PATH variable yourself, so that it will include the new directory's pathname. Still other parameter variables exist that the system does not define. These are usually optional features, such as the EXINIT variable that enables you to set options for the Vi editor. Each time you log in, you must define and assign a value to such variables. Some of the more common parameter variables are SHELL, PATH, PS1, PS2, and MAIL. The SHELL variable holds the pathname of the program for the type of shell you log in to. The PATH variable lists the different directories to be searched for a Linux command. The PS1 and PS2 variables hold the prompt symbols. The MAIL variable holds the pathname of your mailbox file. You can modify the values for any of them to customize your shell.

Using Initialization Files

You can automatically define parameter variables using special shell scripts called initialization files. An initialization file is a specially named shell script executed whenever you enter a certain shell. You can edit the initialization file and place in it definitions and assignments for parameter variables. When you enter the shell, the initialization file will execute these definitions and assignments, effectively initializing parameter variables with your own values. For example, the BASH shell's **.bash_profile** file is an initialization file executed every time you log in. It contains definitions and assignments of parameter variables. However, the

.bash_profile file is basically only a shell script, which you can edit with any text editor such as the Vi editor; changing, if you wish, the values assigned to parameter variables.

In the BASH shell, all the parameter variables are designed to be environment variables.When you define or redefine a parameter variable, you also need to export it to make it an environment variable. This means any change you make to a parameter variable must be accompanied by an **export** command. You will see that at the end of the login initialization file, **.bash_profile**, there is usually an **export** command for all the parameter variables defined in it.

Your Home Directory: HOME

The **HOME** variable contains the pathname of your home directory. Your home directory is determined by the parameter administrator when your account is created. The pathname for your home directory is automatically read into your **HOME** variable when you log in. In the next example, the **echo** command displays the contents of the **HOME** variable:

$ **echo $HOME**

/home/chris

The **HOME** variable is often used when you need to specify the absolute pathname of your home directory. In the next example, the absolute pathname of **reports** is specified using **HOME** for the home directory's path:

$ **ls $HOME/reports**

Command Locations:

PATH

The **PATH** variable contains a series of directory paths separated by colons. Each time a command is executed, the paths listed in the **PATH** variable are searched one by one for that command. For example, the **cp** command resides on the system in the directory **/bin**. This directory path is one of the directories listed in the **PATH** variable. Each time you execute the **cp** command, this path is searched and the **cp** command located. The system defines and assigns **PATH** an initial set of pathnames. In Linux, the initial pathnames are **/bin** and **/usr/bin**.

The shell can execute any executable file, including programs and scripts you have created. For this reason, the **PATH** variable can also reference your working directory; so if you want to execute one of your own scripts or programs in your working directory, the shell can locate it. No spaces are allowed between the pathnames in the string. A colon with no pathname specified references your working directory. Usually, a single

colon is placed at the end of the pathnames as an empty entry specifying your working directory.

```
$ echo $PATH
 /bin:/usr/sbin:
```

You can add any new directory path you want to the **PATH** variable. This can be useful if you have created several of your own Linux commands using shell scripts. You can place these new shell script commands in a directory you create and then add that directory to the **PATH** list. Then, no matter what directory you are in, you can execute one of your shell scripts. The **PATH** variable will contain the directory for that script, so that directory will be searched each time you issue a command.

You add a directory to the **PATH** variable with a variable assignment. You can execute this assignment directly in your shell. In the next example, the user **chris** adds a new directory, called **mybin,** to the **PATH**. Although you could carefully type in the complete pathnames listed in **PATH** for the assignment, you can also use an evaluation of **PATH**—**$PATH**—in their place. In this example, an evaluation of **HOME** is also used to designate the user's **home** directory in the new directory's pathname. Notice the empty entry between two colons, which specifies the working directory:

```
 $ PATH=$PATH:$HOME/mybin:
 $ export PATH
 $ echo $PATH
 /bin:/usr/bin::/home/chris/mybin
```

If you add a directory to **PATH** yourself while you are logged in, the directory will be added only for the duration of your login session. When you log back in, the login initialization file, **.bash_profile**, will again initialize your **PATH** with its original set of directories.

The **.bash_profile** file is described in detail a bit later in this chapter. To add a new directory to your **PATH** permanently, you need to edit your **.bash_profile** file and find the assignment for the **PATH** variable. Then, you simply insert the directory, preceded by a colon, into the set of pathnames assigned to **PATH**.

Specifying the BASH Environment: BASH_ENV

The **BASH_ENV** variable holds the name of the BASH shell initialization file to be executed whenever a BASH shell is generated. For example, when a BASH shell script is executed, the **BASH_ENV** variable is checked and the name of the script that it holds is executed before the shell script. The **BASH_ENV** variable usually holds **$HOME/.bashrc**. This is the **.bashrc** file in the user's home directory.

(The **.bashrc** file is discussed later in this chapter.) You can specify a different file if you wish, using that instead of the **.bashrc** file for BASH shell scripts.

Configuring the Shell Prompt

The **PS1** and **PS2** variables contain the primary and secondary prompt symbols, respectively. The primary prompt symbol for the BASH shell is a dollar sign (**$**). You can change the prompt symbol by assigning a new set of characters to the **PS1** variable. In the next example, the shell prompt is changed to the -> symbol:

$ **PS1= '->'**
-> export PS1 ->

You can change the prompt to be any set of characters, including a string, as shown in the next example:

$ **PS1="Please enter a command: "**
Please enter a command: **export PS1**
Please enter a command: **ls**
mydata /reports
Please enter a command:

The **PS2** variable holds the secondary prompt symbol, which is used for commands that take several lines to complete. The default secondary prompt is >. The added command lines begin with the secondary prompt instead of the primary prompt. You can change the secondary prompt just as easily as the primary prompt, as shown here:

$ PS2="@"

Like the TCSH shell, the BASH shell provides you with a predefined set of codes you can use to configure your prompt. With them you can make the time, your username, or your directory pathname a part of your prompt. You can even have your prompt display the history event number of the current command you are about to enter. Each code is preceded by a \ symbol: **\w** represents the current working directory, **\t** the time, and **\u** your username; **\!** will display the next history event number. In the next example, the user adds the current working directory to the prompt:

$ **PS1="\w $"**
/home/dylan $

The codes must be included within a quoted string. If no quotes exist, the code characters are not evaluated and are themselves used as the prompt. **PS1=\w** sets the prompt to the characters **\w**, not the working directory. The next example incorporates both the time and the history event number with a new prompt:

$ **PS1="\t \! ->"**

The following table lists the codes for configuring your prompt:

Prompt Codes	Description
\!	Current history number
\$	Use $ as prompt for all users except the root user, which has the # as its prompt
\d	Current date
\#	History command number for just the current shell
\h	Hostname
\s	Shell type currently active
\t	Time of day in hours, minutes, and seconds
\u	Username
\v	Shell version
\w	Full pathname of the current working directory
\W	Name of the current working directory
\\	Displays a backslash character
\n	Inserts a newline
\[\]	Allows entry of terminal-specific display characters for features like color or bold font
nnn	Character specified in octal format

The default BASH prompt is **\s-\v\$** to display the type of shell, the shell version, and the **$** symbol as the prompt. Some distributions like Fedora and Red Hat have changed this to a more complex command consisting of the user, the hostname, and the name of the current working directory. The actual operation is carried out in the **/etc/bashrc** file discussed in the later section "The System **/etc/ bashrc** BASH Script and the **/etc/profile.d** Directory." A sample configuration is shown here. The **/etc/ bashrc** file uses USER, HOSTNAME, and PWD environment variables to set these values. A simple equivalent is show here with an @ sign in the hostname and a $ for the final prompt symbol. The home directory is represented with a tilde (~).

```
$ PS1="\u@\h:\w$"
richard@turtle.com:~$
```

Specifying Your News Server

Several shell parameter variables are used to set values used by network applications, such as web browsers or newsreaders. **NNTPSERVER** is used to set the value of a remote news server accessible on your network. If you are using an ISP, the ISP usually provides a Usenet news server you can access with your newsreader applications. However, you first have to provide your newsreaders with the Internet address of the news server. This is the role of the **NNTPSERVER** variable. News servers on the Internet usually use the NNTP protocol. **NNTPSERVER** should hold the address of such a news server. For many ISPs, the news server address is

a domain name that begins with **nntp**. The following example assigns the news server address **nntp.myservice.com** to the **NNTPSERVER** shell variable. Newsreader applications automatically obtain the news server address from **NNTPSERVER**. Usually, this assignment is placed in the shell initialization file, **.bash_profile**, so that it is automatically set each time a user logs in.

```
NNTPSERVER=news.myservice.com
export NNTPSERVER
```

Configuring Your Login Shell: .bash_profile

The **.bash_profile** file is the BASH shell's login initialization file, which can also be named **.profile** (as in SUSE or Ubuntu Linux). It is a script file that is automatically executed whenever a user logs in. The file contains shell commands that define system environment variables used to manage your shell. They may be either redefinitions of system-defined variables or definitions of user-defined variables. For example, when you log in, your user shell needs to know what directories hold Linux commands. It will reference the **PATH** variable to find the pathnames for these directories. However, first, the **PATH** variable must be assigned those pathnames. In the **.bash_profile** file, an assignment operation does just this. Because it is in the **.bash_profile** file, the assignment is executed automatically when the user logs in.

Exporting Variables

Parameter variables also need to be exported, using the **export** command, to make them accessible to any subshell you may enter. You can export several variables in one **export command** by listing them as arguments. Usually, the **.bash_profile** file ends with an **export** command with a list of all the variables defined in the file. If a variable is missing from this list, you may be unable to access it. Notice the **export** command at the end of the **.profile** file in the first example in the next section. You can also combine the assignment and **export** command into one operation as shown here for **NNTPSERVER**:

```
export NNTPSERVER=news.myservice.com
```

Variable Assignments

A copy of the standard **.bash_profile** file provided for you when your account is created is listed in the next example. Notice how **PATH** is assigned, as is the value of **$HOME**. Both **PATH** and **HOME** are parameter variables the system has already defined. **PATH** holds the pathnames of directories searched for any command you enter, and **HOME** holds the pathname of your home directory. The assignment **PATH=$PATH:$HOME/bin** has the effect of redefining **PATH** to

include your **bin** directory within your home directory so that your **bin** directory will also be searched for any commands, including ones you create yourself, such as scripts or programs. Notice **PATH** is then exported, so that it can be accessed by any subshell. **.bash_profile**

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH
```

The root user version of **.bash_profile** adds an entry to unset the **USERNAME** variable, which contains the user's text name.

```
unset USERNAME
```

Should you want to have your home directory searched also, you can use any text editor to modify this line in your **.bash_profile** file to **PATH=$PATH:$HOME/bin:$HOME**, adding **:$HOME** at the end. In fact, you can change this entry to add as many directories as you want searched. If you add a colon at the end, then your current working directory will also be searched for commands. Making commands automatically executable in your current working directory could be a security risk, allowing files in any directory to be executed, instead of in certain specified directories. An example of how to modify your **.bash_profile** file is shown in the following section.

```
PATH=$PATH:$HOME/bin:$HOME:
```

Editing Your BASH Profile Script

Your **.bash_profile** initialization file is a text file that can be edited by a text editor, like any other text file. You can easily add new directories to your **PATH** by editing **.bash_profile** and using editing commands to insert a new directory pathname in the list of directory pathnames assigned to the **PATH** variable. You can even add new variable definitions. If you do so, however, be sure to include the new variable's name in the **export** command's argument list. For example, if your **.bash_profile** file does not have any definition of the **EXINIT** variable, you can edit the file and add a new line that assigns a value to **EXINIT**. The definition **EXINIT='set nu ai'** will configure the Vi editor with line numbering and indentation. You then need to add **EXINIT** to the **export** command's argument list. When the **.bash_profile** file executes again, the **EXINIT** variable will be set to the command **set nu ai**. When the Vi editor is

invoked, the command in the **EXINIT** variable will be executed, setting the line number and auto-indent options automatically.

In the following example, the user's **.bash_profile** has been modified to include definitions of **EXINIT** and redefinitions of **PATH**, **PS1**, and **HISTSIZE**. The **PATH** variable has **$HOME**: added to its value. **$HOME** is a variable that evaluates to the user's home directory, and the ending colon specifies the current working directory, enabling you to execute commands that may be located in either the home directory or the working directory. The redefinition of **HISTSIZE** reduces the number of history events saved, from 1000 defined in the system's **.profile** file, to 30. The redefinition of the **PS1** parameter variable changes the prompt to include the pathname of the current working directory. Any changes you make to parameter variables within your **.bash_profile** file override those made earlier by the system's **.profile** file. All these parameter variables are then exported with the **export** command.

.bash_profile

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ];
then
. ~/.bashrc
fi
# User-specific environment and startup programs
 PATH=$PATH:$HOME/bin:$HOME:
unset USERNAME
HISTSIZE=30
NNTPSERVER=news.myserver.com
EXINIT='set nu ai'
PS1="\w \$"
export PATH HISTSIZE EXINIT PS1 NNTPSERVER
```

Manually Re-executing the .bash_profile Script

Although **.bash_profile** is executed each time you log in, it is not automatically re-execute after you make changes to it. The **.bash_profile** file is an initialization file that is executed only whenever you log in. If you want to take advantage of any changes you make to it without having to log out and log in again, you can re-execute **.bash_profile** with the dot (**.**) command. The **.bash_profile** file is a shell script and, like any shell script, can be executed with the **.** command.

```
$ . .bash_profile
```

Alternatively, you can use the **source** command to execute the **.bash_profile** initialization file or any initialization file such as **.login** used in the TCSH shell or **.bashrc**.

$ **source .bash_profile**

System Shell Profile Script

Your Linux system also has its own profile file that it executes whenever any user logs in. This system initialization file is simply called **profile** and is found in the **/etc** directory, **/etc/ profile**. This file contains parameter variable definitions the system needs to provide for each user. A copy of the system's **profile** file follows at the end of this section. On some distributions, this will be a very simple file, and on others much more complex. Some distributions like Fedora and Red Hat use a **pathmunge** function to generate a directory list for the **PATH** variable. Normal user paths will lack the system directories but include the name of their home directory, along with **/usr/kerberos/bin** for Kerberos tools. The path generated for the root user (EUID of 0) will include both system and user application directories, adding **/usr/kerberos/sbin**, **/sbin**, **/usr/sbin**, and **/usr/local/ sbin**, as well as the root user local application directory, **/root/bin**.

echo $PATH

/usr/kerberos/bin/usr/local/bin:usr/sbin:/bin:/usr/X11R6/bin:/home/richard/bin

A special work-around is included for the Korn Shell to set the User and Effective User IDs (**EUID** and **UID)**.

The **USER**, **MAIL**, and **LOGNAME** variables are then set, provided that **/usr/bin/id**, which provides the user ID, is executable. The **id** command with the **-un** option provides the user ID's text name only, like **chris** or **richard**.

HISTSIZE is also redefined to include a larger number of history events. An entry has been added here for the **NNTPSERVER** variable. Normally, a news server address is a value that needs to be set for all users. Such assignments should be made in the system's **/etc/profile** file by the system administrator, rather than in each individual user's own **.bash_profile** file.

The **/etc/profile** file also runs the **/etc/inputrc** file, which configures your command line editor. Here you will find key assignments for different tasks, such as moving to the end of a line or deleting characters. Global options are set as well. Keys are represented in hexadecimal format.

The number of aliases and variable settings needed for different applications would make the **/etc/profile** file much too large to manage. Instead, application- and task-specific aliases and variables are placed in separate configuration files located in the **/etc/profile.d** directory. There are corresponding scripts for both the BASH and C shells. The BASH shell scripts are run by **/etc/profile**. The scripts are named for the kinds of tasks and applications they configure. For example, on Red Hat, sets the file type color coding when the **ls** command displays files and directories. The vim.sh file sets the an alias for the **vi** command, executing **vim** whenever the user enters just **vi**. The kde.sh file sets the global environment variable **KDEDIR**, specifying the KDE applications directory, in this case **/usr**. The krb5.sh file adds the pathnames for Kerberos, **/usr/kerberos**, to the **PATH** variable. Files run by the BASH shell end in the extension **.sh**, and those run by the C shell have the extension **.csh**.

/etc/profile

```
# /etc/profile
# Systemwide environment and startup programs, for login setup
# Functions and aliases go in /etc/bashrc
pathmunge () {
if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
if [ "$2" = "after" ] ; then
PATH=$PATH:$1
else
PATH=$1:$PATH
fi
fi
}
# ksh workaround
if [ -z "$EUID" -a -x /usr/bin/id ]; then
EUID=`id -u`
UID=`id -ru`
fi
# Path manipulation
if [ "$EUID" = "0" ]; then
pathmunge /sbin
pathmunge /usr/sbin
pathmunge /usr/local/sbin
fi
```

```
        # No core files by default
        ulimit -S -c 0 > /dev/null 2>&1
        if [ -x /usr/bin/id ]; then
        USER="`id -un`"
        LOGNAME=$USER
        MAIL="/var/spool/mail/$USER"
        fi
        HOSTNAME=`/bin/hostname`
        HISTSIZE=1000
        if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
INPUTRC=/etc/inputrc
fi
export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE
INPUTRC
         for i in /etc/profile.d/*.sh ; do
         if [ -r "$i" ]; then
        . $i
        fi
        done
         unset i
         unset pathmunge
```

Configuring the BASH Shell: .bashrc

The **.bashrc** file is a configuration file executed each time you enter the BASH shell or generate a subshell. If the BASH shell is your login shell, **.bashrc** is executed along with your **.bash_login** file when you log in. If you enter the BASH shell from another shell, the **.bashrc** file is automatically executed, and the variable and alias definitions it contains will be defined. If you enter a different type of shell, the configuration file for that shell will be executed instead. For example, if you were to enter the TCSH shell with the **tcsh** command, the **.tcshrc** configuration file would be executed instead of **.bashrc**.

The User .bashrc BASH Script

The **.bashrc** shell configuration file is actually executed each time you generate a BASH shell, such as when you run a shell script. In other words, each time a subshell is created, the **.bashrc** file is executed. This has the effect of exporting any local variables or aliases you have defined in the **.bashrc** shell initialization file. The **.bashrc** file usually contains the definition of aliases and any feature variables used to turn on shell features. Aliases and feature variables are locally defined within the shell. But the **.bashrc** file defines them in every shell. For this reason, the **.bashrc** file

usually holds aliases and options you want defined for each shell. In this example, the standard **.bashrc** for users includes only the execution of the system **/etc/bashrc** file. As an example of how you can add your own aliases and options, aliases for the **rm**, **cp**, and **mv** commands and the shell **noclobber** and **ignoreeof** options have been added. For the root user **.bashrc**, the **rm**, **cp**, and **mv** aliases have already been included in the root's **.bashrc** file.

.bashrc

```
# Source global definitions
if [ -f /etc/bashrc ];
then
. /etc/bashrc
fi
set -o ignoreeof
set -o noclobber
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
```

You can add any commands or definitions of your own to your **.bashrc** file. If you have made changes to **.bashrc** and you want them to take effect during your current login session, you need to re-execute the file with either the **.** or the **source** command.

$ **. .bashrc**

The System /etc/bashrc BASH Script and the /etc/profile.d Directory

Linux systems usually contain a system **bashrc** file executed for all users. The file contains certain global aliases and features needed by all users whenever they enter a BASH shell. This is located in the **/etc** directory, **/etc/bashrc**. A user's own **.bashrc** file, located in the home directory, contains commands to execute this system **.bashrc** file. The **./etc/bashrc** command in the previous example of **.bashrc** does just that. Currently the **/etc/bashrc** file sets the default shell prompt, one for a terminal window and another for a screen interface. Several other specialized aliases and variables are then set using configuration files located in the **/etc/profile.d** directory. These scripts are executed by **/etc/bashrc** if the shell is not the user login shell.

The BASH Shell Logout File: .bash_logout

The **.bash_logout** file is also a configuration file, but it is executed when the user logs out. It is designed to perform any operations you want done whenever you log out. Instead of variable definitions, the **.bash_logout**

file usually contains shell commands that form a kind of shutdown procedure—actions you always want taken before you log out. One common logout command is to clear the screen and then issue a farewell message.

As with **.bash_profile**, you can add your own shell commands to **.bash_logout**. In fact, the **.bash_logout** file is not automatically set up for you when your account is first created.

You need to create it yourself, using the Vi or Emacs editor. You could then add a farewell message or other operations. In the next example, the user has a **clear** command and an **echo** command in the **.bash_logout** file. When the user logs out, the **clear** command clears the screen, and then the **echo** command displays the message "Good-bye for now."

```
.bash_logout
# ~/.bash_logout
clear
echo "Good-bye for now"
```

The TCSH Shell Configuration

The TCSH shell is essentially a version of the C shell with added features. Configuration operations perform much the same tasks but with slightly different syntax. The **alias** command operates the same but uses a different command format. System variables are assigned values using TCSH shell assignment operators, and the initialization and configuration files have different names.

TCSH/C Aliases

You use the **alias** command to create another name for a command. The alias operates like a macro that expands to the command it represents. The alias does not literally replace the name of the command; it simply gives another name to that command. An **alias** command begins with the keyword alias and the new name for the command, followed by the command that the alias will reference. In the next example, the **ls** command is aliased with the name **list**. **list** becomes another name for the **ls** command.

```
> alias list ls   > ls
mydata intro
> list
mydata intro
>
```

Should the command you are aliasing have options, you will need to enclose the command and the option within single quotes. An aliased

command that has spaces will need quotation marks as well. In the next example, **ls** with the **-l** option is given the alias **longl:**

> **alias longl 'ls -l'**
> **ls -l**
-rw-r--r-- 1 chris weather 207 Feb 20 11:55 mydata
> **longl**
-rw-r--r-- 1 chris weather 207 Feb 20 11:55 mydata
>

You can also use the name of a command as an alias. In the case of the **rm**, **cp**, and **mv** commands, the **-i** option should always be used to ensure that an existing file is not overwritten. Instead of always being careful to use the **-i** option each time you use one of these commands, you can alias the command name to include the option. In the next examples, the **rm**, **cp**, and **mv** commands have been aliased to include the **-i** option.

> alias rm 'rm -i'
> alias mv 'mv -i'
> alias cp 'cm -i'

The **alias** command by itself provides a list of all aliases in effect and their commands. An alias can be removed with the **unalias** command.

> **alias**
lss ls -s
list ls
rm rm -i
> **unalias lss**

TCSH/C Shell Feature Variables: Shell Features

The TCSH shell has several features that allow you to control how different shell operations work. The TCSH shell's features include those in the PDSKH shell as well as many of its own. For example, the TCSH shell has a **noclobber** option to prevent redirection from overwriting files. Some of the more commonly used features are **echo**, **noclobber**, **ignoreeof**, and **noglob**. The TCSH shell features are turned on and off by defining and undefining a variable associated with that feature. A variable is named for each feature, for example, the **noclobber** feature is turned on by defining the **noclobber** variable. You use the **set** command to define a variable and the **unset** command to undefine a variable. To turn on the **noclobber** feature you issue the command **set noclobber**. To turn it off you use the command **unset noclobber**.

> set feature-variable
> unset feature-variable

These variables are also sometimes referred to as toggles since they are used to turn features on and off.

echo

Setting **echo** enables a feature that displays a command before it is executed. The command **set echo** turns the **echo** feature on, and the command **unset echo** turns it off.

ignoreeof

Setting **ignoreeof** enables a feature that prevents users from logging out of the user shell with a CTRL-D. It is designed to prevent accidental logouts. With this feature turned off, you can log out by pressing CTRL-D. However, CTRL-D is also used to end user input entered directly into the standard input. It is used often for the Mail program or for utilities such as **cat**. You can easily enter an extra CTRL-D in such circumstances and accidentally log yourself out. The **ignoreeof** feature prevents such accidental logouts. When it is set, you have to explicitly log out, using the **logout** command:

$ **set ignoreeof**
$ **^D**
Use logout to logout
$

noclobber

Setting **noclobber** enables a feature that safeguards existing files from redirected output.With the **noclobber** feature, if you redirect output to a file that already exists, the file will not be overwritten with the standard output. The original file will be preserved. There may be situations in which you use a name that you have already given to an existing file as the name for the file to hold the redirected output. The **noclobber** feature prevents you from accidentally overwriting your original file:

> **set noclobber**
> cat preface > myfile
myfile: file exists
$

There may be times when you want to overwrite a file with redirected output. In this case, you can place an exclamation point after the redirection operator. This will override the **noclobber** feature, replacing the contents of the file with the standard output:

> **cat preface >! myfile**

noglob

Setting **noglob** enables a feature that disables special characters in the user shell. The characters *****, **?**, **[]**, and **~** will no longer expand to matched filenames. This feature is helpful if, for some reason, you have special characters as part of a filename. In the next example, the user needs to reference a file that ends with the **?** character, **answers?**. First the user turns off special characters, using the **noglob** option. Now the question mark on the command line is taken as part of the filename, not as a special character, and the user can reference the **answers?** file.

$ **set noglob**
$ **ls answers**?
answers?

TCSH/C Special Shell Variables for Configuring Your System

As in the BASH shell, you can use special shell variables in the TCSH shell to configure your system. Some are defined initially by your system, and you can later redefine them with a new value. There are others that you must initially define yourself. One of the more commonly used special variables is the **prompt** variable that allows you to create your own command line prompts. Another is the **history** variable with which you determine how many history events you want to keep track of.

In the TCSH shell, many special variables have names and functions similar to those in the BASH or Public Domain Korn Shell (PDKSH) shells. Some are in uppercase, but most are written in lowercase. The **EXINIT** and **TERM** variables retain their uppercase form. However, **history** and **cdpath** are written in lowercase. Other special variables may perform similar functions but have very different implementations. For example, the **mail** variable holds the same information as the BASH MAIL, MAILPATH, and **MAILCHECK** variables together.

Prompt, prompt2, prompt3

The **prompt**, **prompt2**, and **prompt3** variables hold the prompts for your command line. You can configure your prompt to be any symbol or string that you want. To have your command line display a different symbol as a prompt, you simply use the **set** command to assign that symbol to the prompt variable. In the next example, the user assigns a + sign to the **prompt** variable, making it the new prompt.

> **set prompt = "+"**
+

You can use a predefined set of codes to make configuring your prompt easier. With them, you can make the time, your username, or your directory pathname a part of your prompt. You can even have your prompt display the history event number of the current command you are about to enter. Each code is preceded by a **%** symbol, for example, **%/** represents the current working directory, **%t** the time, and **%n** your username. **%!** will display the next history event number. In the next example, the user adds the current working directory to the prompt.

> **set prompt = "%/ >"**

/home/dylan >

The next example incorporates both the time and the history event number with a new prompt. > **set prompt = "%t %! $"**

Here is a list of the codes:

%/	Current working directory
%h, %!, !	Current history number
%t	Time of day
%n	Username
%d	Day of the week
%w	Current month
%y	Current year

The prompt2 variable is used in special cases when a command may take several lines to input. **prompt2** is displayed for the added lines needed for entering the command. **prompt3** is the prompt used if the spell check feature is activated.

cdpath

The **cdpath** variable holds the pathnames of directories to be searched for specified subdirectories referenced with the **cd** command. These pathnames form an array just like the array of pathnames assigned to the TCSH shell **path** variable. Notice the space between the pathnames.

> set cdpath=(/usr/chris/reports /usr/chris/letters)

History and savehist

As you learned earlier, the **history** variable can be used to determine the number of history events you want saved. You simply assign to it the maximum number of events that **history** will record. When the maximum is reached, the count starts over again from 1. The **savehist** variable, however, holds the number of events that will be saved in the file **.history** when you log out. When you log in again, these events will become the initial history list.

In the next example, up to 20 events will be recorded in your history list while you are logged in. However, only the last 5 will be

saved in the **.history** file when you log out. Upon logging in again, your history list will consist of your last 5 commands from the previous session.

> **set history=20**

> **set savehist=5**

Mail

In the TCSH shell, the **mail** variable combines the features of the MAIL, MAILCHECK, and MAILPATH variables in the BASH and PDKSH shells. The TCSH shell **mail** variable is assigned as its value an array whose elements contain both the time interval for checking for mail and the directory pathnames for mailbox files to be checked. To assign values to these elements, you assign an array of values to the **mail** variable. The array of new values is specified with a list of words separated by spaces and enclosed in parentheses. The first value is a number that sets the number of seconds to wait before checking for mail again. This value is comparable to that held by the BASH shell's MAILCHECK variable. The remaining values consist of the directory pathnames of mailbox files that are to be checked for your mail. Notice that these values combine the functions of the BASH and Korn shells' MAIL and MAILPATH variables.

In the next example, the **mail** variable is set to check for mail every 20 minutes (1200 seconds), and the mailbox file checked is in **usr/mail/chris**. The first value in the array assigned to mail is 1200, and the second value in the array is the pathname of the mailbox file to be checked.

> **set mail (1200 /usr/mail/chris)**

You can, just as easily, add more mailbox file pathnames to the **mail** array. In the next example, two mailboxes are designated. Notice the spaces surrounding each element.

> **set mail (1200 /usr/mail/chris /home/mail/chris)**

TCSH/C Shell Initialization Files: .login, .tcshrc, .logout

The TCSH shell has three initialization files: **.login**, **.logout**, and **.tcshrc**. The files are named for the operation they execute. The **.login** file is a login initialization file that executes each time you log in. The **.logout** file executes each time you log out. The **.tcshrc** file is a shell initialization file that executes each time you enter the TCSH shell, either from logging in or by explicitly changing to the TCSH shell from another shell with the **tcsh** command.

.login

The TCSH shell has its own login initialization file called the **.login** file that contains shell commands and special variable definitions used to configure your shell. The **.login** file corresponds to the **.profile** file used in the BASH and PDKSH shells.

A **.login** file contains **setenv** commands that assign values to special environment variables, such as **TERM**. You can change these assigned values by editing the **.login** file with any of the standard editors. You can also add new values. Remember, however, that in the TCSH shell, the command for assigning a value to an environment variable is **setenv**. In the next example, the **EXINIT** variable is defined and assigned the Vi editor's line numbering and auto-indent options.

> **setenv EXINIT 'set nu ai'**

Be careful when editing your **.login** file. Inadvertent editing changes could cause variables to be set incorrectly or not at all. It is wise to make a backup of your **.login** file before editing it.

If you have made changes to your **.login** file and you want the changes to take effect during your current login session, you will need to re-execute the file. You do so using the **source** command. The **source** command will actually execute any initialization file, including the **.tcshrc** and **.logout** files. In the next example, the user re-executes the **.login** file.

> **source .login**

If you are also planning to use the PDKSH shell on your Linux system, you need to define a variable called **ENV** within your **.login** file and assign it the name of the PDKSH shell initialization file. If you should later decide to enter the PDKSH shell from your TCSH shell, the PDKSH shell initialization file can be located and executed for you. In the example of the **.login** file shown next, you will see that the last command sets the PDKSH shell initialization file to **.kshrc** to the **ENV** variable: **setenv ENV $HOME/.kshrc**.

.login
setenv term vt100
setenv EXINIT 'set nu ai'
setenv ENV $HOME/.kshrc

.tcshrc

The **.tcshrc** initialization file is executed each time you enter the TCSH shell or generate any subshell. If the TCSH shell is your login shell, then the **.tcshrc** file is executed along with your **.login** file when you log in. If you enter the TCSH shell from another shell, the **.tcshrc** file

is automatically executed, and the variable and alias definitions it contains will be defined.

The **.tcshrc** shell initialization file is actually executed each time you generate a shell, such as when you run a shell script. In other words, each time a subshell is created, the **.tcshrc** file is executed. This allows you to define local variables in the **.tcshrc** initialization file and have them, in a sense, exported to any subshell. Even though such user-defined special variables as **history** are local, they will be defined for each subshell generated. In this way, **history** is set for each subshell. However, each subshell has its own local **history** variable. You could even change the local **history** variable in one subshell without affecting any of those in other subshells. Defining special variables in the shell initialization file allows you to treat them like BASH shell exported variables. An exported variable in a BASH or PDKSH shell only passes a copy of itself to any subshell. Changing the copy does not affect the original definition.

The **.tcshrc** file also contains the definition of aliases and any feature variables used to turn on shell features. Aliases and feature variables are locally defined within the shell. But the **.tcshrc** file will define them in every shell. For this reason, **.tcshrc** usually holds such aliases as those defined for the **rm**, **cp**, and **mv** commands. The next example is a **.tcshrc** file with many of the standard definitions.

```
.tcshrc
set shell=/usr/bin/csh
set path= $PATH (/bin /usr/bin . )
set cdpath=( /home/chris/reports /home/chris/letters )
set prompt="! $cwd >"
set history=20
set ignoreeof
set noclobber
alias rm 'rm -i'
alias mv 'mv -i'
alias cp 'cm -i'
```

Local variables, unlike environment variables, are defined with the **set** command. Any local variables that you define in **.tcshrc** should use the **set** command. Any variables defined with **setenv** as environment variables, such as **TERM**, should be placed in the **.login** file. The next example shows the kinds of definitions found in the **.tcshrc** file. Notice that the **history** and **noclobber** variables are defined using the **set** command.

```
set history=20
set noclobber
```

You can edit any of the values assigned to these variables. However, when editing the pathnames assigned to **path** or **cdpath**, bear in mind that these pathnames are contained in an array. Each element in an array is separated by a space. If you add a new pathname, you need to be sure that there is a space separating it from the other pathnames.

If you have made changes to **.tcshrc** and you want them to take effect during your current login session, remember to re-execute the **.tcshrc** file with the **source** command:

> **source .tcshrc**

.logout

The **.logout** file is also an initialization file, but it is executed when the user logs out. It is designed to perform any operations you want done whenever you log out. Instead of variable definitions, the **.logout** file usually contains shell commands that form a shutdown procedure. For example, one common logout command is the one to check for any active background jobs; another is to clear the screen and then issue a farewell message.

As with **.login**, you can add your own shell commands to the **.logout** file. Using the Vi editor, you can change the farewell message or add other operations. In the next example, the user has a **clear** and an **echo** command in the **.logout** file. When the user logs out, the **clear** command will clear the screen, and **echo** will display the message "Good-bye for now".

```
.logout
clear
echo "Good-bye for now"
```

Review & Self Assessment Question :

Q1- What is Aliasing Command ?
Q2- What do youn mean by shell parametrized variable ?
Q3- What do you mean by Initialization files?
Q4- What is PATH variable ?
Q5- What do you mean by Exporting Variable ?

Further Readings

Linux Operating System Richard Petersen
Linux Operating System Paul S. Wang
Linux Operating System by David Maxwell and Andrew Bedford
Linux Operating System by Richard Blum and Christine Bresnahan
Linux Operating System by Bhatt P.C.P

UNIT: 6- LINUX FILES, DIRECTORIES, AND ARCHIVES

Contents

Introduction

In Linux, all files are organized into directories that, in turn, are hierarchically connected to each other in one overall file structure. A file is referenced not according to just its name, but also according to its place in this file structure. You can create as many new directories as you want, adding more directories to the file structure. The Linux file commands can perform sophisticated operations, such as moving or copying whole directories along with their subdirectories. You can use file operations such as **find**, **cp**, **mv**, and **ln** to locate files and copy, move, or link them from one directory to another. Desktop file managers, such as Konqueror and Nautilus used on the KDE and GNOME desktops, provide a graphical user interface to perform the same operations using icons, windows,and menus. This chapter will focus on the commands you use in the shell command line to manage files, such as **cp** and **mv**. However, whether you use the command line or a GUI file manager, the underlying file structure is the same.

The organization of the Linux file structure into its various system and network administration directories is discussed in detail in Chapter 32. Though not part of the Linux file structure, there are also special tools you can use to access Windows partitions and floppy disks. These follow much the same format as Linux file commands.

Archives are used to back up files or to combine them into a package, which can then be transferred as one file over the Internet or posted on an FTP site for easy downloading. The standard archive utility used on Linux and Unix systems is tar, for which several GUI front ends exist. You have several compression programs to choose from, including GNU zip (gzip), Zip, bzip, and compress.

Linux Files

You can name a file using any letters, underscores, and numbers. You can also include periods and commas. Except in certain special cases, you should never begin a filename with a period. Other characters, such as slashes, question marks, or asterisks, are reserved for use as special characters by the system and should not be part of a filename. Filenames can be as long as 256 characters. Filenames can also include spaces, though to reference such filenames from the command line, be sure to encase them in quotes. On a desktop like GNOME or KDE, you do not need quotes.

You can include an extension as part of a filename. A period is used to distinguish the filename proper from the extension. Extensions can be useful for categorizing your files. You are probably familiar with certain standard extensions that have been adopted by convention. For example, C source code files always have an extension of **.c**. Files that contain compiled object code have a**.o** extension. You can, of course, make up your own file extensions. The following examples are all valid Linux filenames. Keep in mind that to reference the last of these names on the command line, you would have to encase it in quotes as “New book review”:

preface
chapter2
9700info
New_Revisions
calc.c
intro.bk1
New book review

Special initialization files are also used to hold shell configuration commands. These are the hidden, or dot, files, which begin with a period. Dot files used by commands and applications have predetermined names, such as the **.mozilla** directory used to hold your Mozilla data and configuration files. Recall that when you use **ls** to display your filenames, the dot files will not be displayed. To include the dot files, you need to use **ls** with the **-a** option.

The **ls -l** command displays detailed information about a file. First the permissions are displayed, followed by the number of links, the owner of the file, the name of the group the user belongs to, the file size in bytes, the date and time the file was last modified, and the name of the file. Permissions indicate who can access the file: the user, members of a group, or all other users. Permissions are discussed in detail later in this chapter. The group name indicates the group permitted to access the file object. In the example in the next paragraph, the file type for **mydata** is that of an ordinary file. Only one link exists, indicating the file has no other names and no other links. The owner's name is **chris**, the same as the login name, and the group name is **weather**. Other users probably also belong to the **weather** group. The size of the file is 207 bytes, and it was last modified on February 20 at 11:55 A.M. The name of the file is **mydata**.

If you want to display this detailed information for all the files in a directory, simply use the **ls -l** command without an argument.

$ **ls -l**

-rw-r--r-- 1 chris weather 207 Feb 20 11:55 mydata

-rw-rw-r-- 1 chris weather 568 Feb 14 10:30 today

-rw-rw-r-- 1 chris weather 308 Feb 17 12:40 Monday

All files in Linux have one physical format—a byte stream. A byte stream is just a sequence of bytes. This allows Linux to apply the file concept to every data component in the system. Directories are classified as files, as are devices. Treating everything as a file allows Linux to organize and exchange data more easily. The data in a file can be sent directly to a device such as a screen because a device interfaces with the system using the same byte-stream file format as regular files.

This same file format is used to implement other operating system components. The interface to a device, such as the screen or keyboard, is designated as a file. Other components, such as directories, are themselves byte-stream files, but they have a special internal organization. A directory file contains information about a directory, organized in a special directory format. Because these different components are treated as files, they can be said to constitute different file types. A character device is one file type. A directory is another file type. The number of these file types may vary according to your specific implementation of Linux. Five common types of files exist, however: ordinary files, directory files, first-in first-out pipes, character device files, and block device files. Although you may rarely reference a file's type, it can be useful when searching for directories or devices. Later in the chapter, you'll see how to use the file

type in a search criterion with the **find** command to search specifically for directory or device names.

Although all ordinary files have a byte-stream format, they may be used in different ways. The most significant difference is between binary and text files. Compiled programs are examples of binary files. However, even text files can be classified according to their different uses. You can have files that contain C programming source code or shell commands, or even a file that is empty. The file could be an executable program or a directory file. The Linux **file** command helps you determine what a file is used for. It examines the first few lines of a file and tries to determine a classification for it. The **file** command looks for special keywords or special numbers in those first few lines, but it is not always accurate. In the next example, the **file** command examines the contents of two files and determines a classification for them:

$ **file monday reports**
monday: text
reports: directory

If you need to examine the entire file byte by byte, you can do so with the **od** (octal dump) command. The **od** command performs a dump of a file. By default, it prints every byte in its octal representation. However, you can also specify a character, decimal, or hexadecimal representation. The **od** command is helpful when you need to detect any special character in your file or if you want to display a binary file.

The File Structure

Linux organizes files into a hierarchically connected set of directories. Each directory may contain either files or other directories. In this respect, directories perform two important functions. A directory holds files, much like files held in a file drawer, and a directory connects to other directories, much as a branch in a tree is connected to other branches. Because of the similarities to a tree, such a structure is often referred to as a tree structure

The Linux file structure branches into several directories beginning with a root directory, /. Within the root directory, several system directories contain files and programs that are features of the Linux system. The root directory also contains a directory called **/home** that contains the home directories of all the users in the system. Each user's home directory, in turn, contains the directories the user has made for his or her own use. Each of these can also contain directories. Such nested directories branch out from the user's home directory..

Home Directories

When you log in to the system, you are placed within your home directory. The name given to this directory by the system is the same as your login name. Any files you create when you first log in are organized within your home directory. Within your home directory, however, you can create more directories. You can then change to these directories and store files in them. The same is true for other users on the system. Each user has his or her own home directory, identified by the appropriate login name. Users, in turn, can create their own directories.

You can access a directory either through its name or by making it your working directory. Each directory is given a name when it is created. You can use this name in file operations to access files in that directory. You can also make the directory your working directory. If you do not use any directory names in a file operation, the working directory will be accessed. The working directory is the one from which you are currently working. When you log in, the working directory is your home directory, usually having the same name as your login name. You can change the working directory by using the **cd** command to designate another directory as the working directory.

Pathnames

The name you give to a directory or file when you create it is not its full name. The full name of a directory is its pathname. The hierarchically nested relationship among directories forms paths, and these paths can be used to identify and reference any directory or file uniquely or absolutely. Each directory in the file structure can be said to have its own unique path. The actual name by which the system identifies a directory always begins with the root directory and consists of all directories nested below that directory.

In Linux, you write a pathname by listing each directory in the path separated from the last by a forward slash. A slash preceding the first directory in the path represents the root. The pathname for the **robert** directory is **/home/robert**. The pathname for the **reports** directory is **/home/chris/reports**. Pathnames also apply to files. When you create a file within a directory, you give the file a name. The actual name by which the system identifies the file, however, is the filename combined with the path of directories from the root to the file's directory. As an example, the pathname for **monday** is **/home/chris/reports/monday** (the root directory is represented by the first slash). The path for the **monday** file consists of the root, **home**, **chris**, and **reports** directories and the filename **monday**.

Pathnames may be absolute or relative. An absolute pathname is the complete pathname of a file or directory beginning with the root

directory. A relative pathname begins from your working directory; it is the path of a file relative to your working directory. The working directory is the one you are currently operating in. Using the previous example, if **chris** is your working directory, the relative pathname for the file **monday** is **reports/monday**.

The absolute pathname for **monday** is **/home/chris/reports/monday**. The absolute pathname from the root to your home directory can be especially complex and, at times, even subject to change by the system administrator. To make it easier to reference, you can use a special character, the tilde (**~**), which represents the absolute pathname of your home directory. In the next example, from the **thankyou** directory, the user references the **weather** file in the home directory by placing a tilde and slash before **weather**:

```
$ pwd
/home/chris/letters/thankyou
$ cat ~/weather
raining and warm
$
```

You must specify the rest of the path from your home directory. In the next example, the user references the **monday** file in the **reports** directory. The tilde represents the path to the user's home directory, **/home/chris**, and then the rest of the path to the **monday** file is specified.

```
$ cat ~/reports/monday
```

System Directories

The root directory that begins the Linux file structure contains several system directories. The system directories contain files and programs used to run and maintain the system. Many contain other subdirectories with programs for executing specific features of Linux. For example, the directory **/usr/bin** contains the various Linux commands that users execute, such as **lp**l. The directory **/bin** holds system-level commands.

Listing, Displaying, and Printing Files: ls, cat, more, less, and lpr

One of the primary functions of an operating system is the management of files. You may need to perform certain basic output operations on your files, such as displaying them on your screen or printing them. The Linux system provides a set of commands that perform basic file-management operations, such as listing, displaying, and printing files, as well as copying, renaming, and erasing files. The command names are usually made up of abbreviated versions of words. For example, the **ls** command is a shortened form of "list" and lists the files in your directory. The **lpr**

command is an abbreviated form of "line print" and will print a file. The **cat**, **less**, and **more** commands display the contents of a file on the screen. Table 6-2 lists these commands with their different options. When you log in to your Linux system, you may want a list of the files in your home directory. The **ls** command, which outputs a list of your file and directory names, is useful for this. The **ls** command has many possible options for displaying filenames according to specific features.

Directory	Function
/	Begins the file system structure, called the *root*.
/home	Contains users' home directories.
/bin	Holds all the standard commands and utility programs.
/usr	Holds those files and commands used by the system; this directory breaks down into several subdirectories.
/usr/bin	Holds user-oriented commands and utility programs.
/usr/sbin	Holds system administration commands.
/usr/lib	Holds libraries for programming languages.
/usr/share/doc	Holds Linux documentation.
/usr/share/man	Holds the online Man files.
/var/spool	Holds spooled files, such as those generated for printing jobs and network transfers.
/sbin	Holds system administration commands for booting the system.
/var	Holds files that vary, such as mailbox files.
/dev	Holds file interfaces for devices such as the terminals and printers (dynamically generated by udev, do not edit).
/etc	Holds system configuration files and any other system files.

Displaying Files: cat, less, and more

You may also need to look at the contents of a file. The **cat** and **more** commands display the contents of a file on the screen. The name **cat** stands for concatenate.

$ **cat mydata**

computers

The **cat** command outputs the entire text of a file to the screen at once. This presents a problem when the file is large because its text quickly speeds past on the screen. The **more** and **less** commands are designed to overcome this limitation by displaying one screen of text at a time. You can then move forward or backward in the text at your leisure. You invoke the **more** or **less** command by entering the command name followed by the name of the file you want to view (**less** is a more powerful and configurable display utility).

$ less mydata

When **more** or **less** invokes a file, the first screen of text is displayed. To continue to the next screen, you press the F key or the

SPACEBAR. To move back in the text, you press the B key. You can quit at any time by pressing the Q key.

Command	Execution
`ls`	Lists file and directory names.
`cat` *filenames*	Displays a file. It can take filenames for its arguments. It outputs the contents of those files directly to the standard output, which, by default, is directed to the screen.
`more` *filenames*	Displays a file screen by screen. Press the SPACEBAR to continue to the next screen and `q` to quit.
`less` *filenames*	Displays a file screen by screen. Press the SPACEBAR to continue to the next screen and `q` to quit.
`lpr` *filenames*	Sends a file to the line printer to be printed; a list of files may be used as arguments. Use the `-P` option to specify a printer.
`lpq`	Lists the print queue for printing jobs.
`lprm`	Removes a printing job from the print queue.

Printing Files: lpr, lpq, and lprm

With the printer commands such as **lpr** and **lprm**, you can perform printing operations such as printing files or canceling print jobs (see Table 6-2). When you need to print files, use the **lpr** command to send files to the printer connected to your system. In the next example, the user prints the **mydata** file:

```
$ lpr mydata
```

If you want to print several files at once, you can specify more than one file on the command line after the **lpr** command. In the next example, the user prints out both the **mydata** and **preface** files:

```
$ lpr mydata preface
```

Printing jobs are placed in a queue and printed one at a time in the background. You can continue with other work as your files print. You can see the position of a particular printing job at any given time with the **lpq** command, which gives the owner of the printing job (the login name of the user who sent the job), t he print job ID, the size in bytes, and the temporary file in which it is currently held.

If you need to cancel an unwanted printing job, you can do so with the **lprm** command, which takes as its argument either the ID number of the printing job or the owner's name. It then removes the print job from the print queue. For this task, **lpq** is helpful, for it provides you with the ID number and owner of the printing job you need to use with **lprm**.

Managing Directories: mkdir, rmdir, ls, cd, and pwd

You can create and remove your own directories, as well as change your working directory, with the **mkdir**, **rmdir**, and **cd** commands. Each of these commands can take as its argument the pathname for a directory. The **pwd** command displays the absolute pathname of your single dot, a

double dot, and a tilde can be used to reference the working directory, the parent of the working directory, and the home directory, respectively. Taken together, these commands enable you to manage your directories. You can create nested directories, move from one directory to another, and use pathnames to reference any of your directories.

Creating and Deleting Directories

You create and remove directories with the **mkdir** and **rmdir** commands. In either case, you can also use pathnames for the directories. In the next example, the user creates the directory **reports**. Then the user creates the directory **letters** using a pathname:

$ mkdir reports

$ mkdir /home/chris/letters

Command	Execution
`mkdir` *directory*	Creates a directory.
`rmdir` *directory*	Erases a directory.
`ls -F`	Lists directory name with a preceding slash.
`ls -R`	Lists working directory as well as all subdirectories.
`cd` *directory name*	Changes to the specified directory, making it the working directory. `cd` without a directory name changes back to the home directory: `$ cd reports`
`pwd`	Displays the pathname of the working directory.
directory name/filename	A slash is used in pathnames to separate each directory name. In the case of pathnames for files, a slash separates the preceding directory names from the filename.
`..`	References the parent directory. You can use it as an argument or as part of a pathname: `$ cd ..` `$ mv ../larisa oldletters`
`.`	References the working directory. You can use it as an argument or as part of a pathname: `$ ls .`
~/pathname	The tilde is a special character that represents the pathname for the home directory. It is useful when you need to use an absolute pathname for a file or directory: `$ cp monday ~/today`

You can remove a directory with the **rmdir** command followed by the directory name. In the next example, the user removes the directory **reports** with the **rmdir** command:

$ rmdir reports

To remove a directory and all its subdirectories, you use the **rm** command with the **-r** option. This is a very powerful command and can easily be used to erase all your files. If your **rm** command is aliased as **rm -i** (interactive mode), you will be prompted for each file. To simply remove all files and subdirectories without prompts, add the **-f** option. The following example deletes the **reports** directory and all its subdirectories:

rm -rf reports

Displaying Directory Contents

You have seen how to use the **ls** command to list the files and directories within your working directory. To distinguish between file and directory names, however, you need to use the **ls** command with the **-F** option. A slash is then placed after each directory name in the list.

$ **ls**
weather reports letters
$ **ls -F** weather reports/ letters/

The **ls** command also takes as an argument any directory name or directory pathname. This enables you to list the files in any directory without first having to change to that directory. In the next example, the **ls** command takes as its argument the name of a directory, **reports**. Then the **ls** command is executed again, only this time the absolute pathname of **reports** is used.

$ **ls reports**
monday tuesday
$ **ls /home/chris/reports**
monday tuesday
$

Moving Through Directories

The **cd** command takes as its argument the name of the directory to which you want to change. The name of the directory can be the name of a subdirectory in your working directory or the full pathname of any directory on the system. If you want to change back to your home directory, you only need to enter the **cd** command by itself, without a filename argument.

$ **cd props**
$ **pwd**
/home/dylan/props

Referencing the Parent Directory

A directory always has a parent (except, of course, for the root). For example, in the preceding listing, the parent for **props** is the **dylan** directory. When a directory is created, two entries are made: one represented with a dot (**.**), and the other with double dots (**..**). The dot represents the pathname of the directory, and the double dots represent the pathname of its parent directory. Double dots, used as an argument in a

command, reference a parent directory. The single dot references the directory itself.

You can use the single dot to reference your working directory, instead of using its pathname. For example, to copy a file to the working directory, retaining the same name, the dot can be used in place of the working directory's pathname. In this sense, the dot is another name for the working directory. In the next example, the user copies the **weather** file from the **chris** directory to the **reports** directory. The **reports** directory is the working directory and can be represented with the single dot.

```
$ cd reports
$ cp /home/chris/weather .
```

The **..** symbol is often used to reference files in the parent directory. In the next example, the **cat** command displays the **weather** file in the parent directory. The pathname for the file is the **..** symbol followed by a slash and the filename.

```
$ cat ../weather
raining and warm
```

File and Directory Operations: find, cp, mv, rm, and ln

As you create more and more files, you may want to back them up, change their names, erase some of them, or even give them added names. Linux provides you with several file commands that enable you to search for files, copy files, rename files, or remove files. If you have a large number of files, you can also search them to locate a specific one. The command names shortened forms of full words, consisting of only two characters. The **cp** command stands for "copy" and copies a file, **mv** stands for "move" and renames or moves a file, **rm** stands for "remove" and erases a file, and **ln** stands for "link" and adds another name for a file, often used as a shortcut to the original. One exception to the two-character rule is the **find** command, which performs searches of your filenames to find a file. All these operations can be handled by the GUI desktops such as GNOME and KDE.

Searching Directories: find

Once you have a large number of files in many different directories, you may need to search them to locate a specific file, or files, of a certain type. The **find** command enables you to perform such a search from the command line. The **find** command takes as its arguments directory names followed by several possible options that specify the type of search and the criteria for the search; it then searches within the directories listed and their subdirectories for files that meet these criteria. The **find** command can

search for a file by name, type, owner, and even the time of the last update.

$ find directory-list -option criteria

The **-name** option has as its criteria a pattern and instructs **find** to search for the filename that matches that pattern. To search for a file by name, you use the **find** command with the directory name followed by the **-name** option and the name of the file.

$ find directory-list -name filename

The **find** command also has options that merely perform actions, such as outputting the results of a search. If you want **find** to display the filenames it has found, you simply include the **-print** option on the command line along with any other options. The **-print** option is an action that instructs **find** to write to the standard output the names of all the files it locates (you can also use the **-ls** option instead to list files in the long format). In the next example, the user searches for all the files in the **reports** directory with the name **monday**. Once located, the file, with its relative pathname, is printed.

$ **find reports -name monday -print**

reports/monday

The **find** command prints out the filenames using the directory name specified in the directory list. If you specify an absolute pathname, the absolute path of the found directories will be output. If you specify a relative pathname, only the relative pathname will be output. In the preceding example, the user specified a relative pathname, **reports**, in the directory list. Located filenames were output beginning with this relative pathname. In the next example, the user specifies an absolute pathname in the directory list. Located filenames are then output using this absolute pathname.

$ **find /home/chris -name monday -print**

/home/chris/reports/Monday

Searching the Working Directory

If you want to search your working directory, you can use the dot in the directory pathname to represent your working directory. The double dots represent the parent directory. The next example searches all files and subdirectories in the working directory, using the dot to represent the working directory. If you are located in your home directory, this is a convenient way to search through all your own directories. Notice the located filenames are output beginning with a dot.

$ **find . -name weather –print**

./weather

You can use shell wildcard characters as part of the pattern criterion for searching files. The special character must be quoted, however, to avoid evaluation by the shell. In the next example, all files with the **.c** extension in the **programs** directory are searched for and then displayed in the long format using the **-ls** action:

```
$ find programs -name '*.c' -ls
```

Locating Directories

You can also use the **find** command to locate other directories. In Linux, a directory is officially classified as a special type of file. Although all files have a byte-stream format, some files, such as directories, are used in special ways. In this sense, a file can be said to have a file type. The **find** command has an option called **-type** that searches for a file of a given type. The **-type** option takes a one-character modifier that represents the file type. The modifier that represents a directory is a **d**. In the next example, both the directory name and the directory file type are used to search for the directory called **thankyou**:

```
$ find /home/chris -name thankyou -type d -print
/home/chris/letters/thankyou
$
```

File types are not so much different types of files as they are the file format applied to other components of the operating system, such as devices. In this sense, a device is treated as a type of file, and you can use **find** to search for devices and directories, as well as ordinary files. Table 6-4 lists the different types available for the **find** command's **-type** option.

You can also use the find operation to search for files by ownership or security criteria, like those belonging to a specific user or those with a certain security context. The **user** option lets you locate all files belonging to a certain user. The following example lists all files that the user **chris** has created or owns on the entire system. To list those just in the users' home directories, you use **/home** for the starting search directory. This finds all files in a user's home directory as well as any owned by that user in other user directories.

```
$ find / -user chris -print
```

Copying Files

To make a copy of a file, you simply give **cp** two filenames as its arguments. The first filename is the name of the file to be copied—the one that already exists. This is often referred to as the source file. The second filename is the name you want for the copy. This will be a new file containing a copy of all the data in the source file. This second argument

is often referred to as the destination file. The syntax for the **cp** command follows:

$ cp source-file destination-file

Command or Option	Execution
`find`	Searches directories for files according to search criteria. This command has several options that specify search criteria and actions to be taken.
`-name` *pattern*	Searches for files with *pattern* in the name.
`-lname` *pattern*	Searches for symbolic link files.
`-group` *name*	Searches for files belonging to the group *name*.
`-gid` *name*	Searches for files belonging to a group according to group ID.
`-user` *name*	Searches for files belonging to a user.
`-uid` *name*	Searches for files belonging to a user according to user ID.
`-size` *numc*	Searches for files with the size *num* in blocks. If **c** is added after *num*, the size in bytes (characters) is searched for.
`-mtime` *num*	Searches for files last modified *num* days ago.
`-newer` *pattern*	Searches for files modified after the one matched by *pattern*.
`-context` *scontext*	Searches for files according to security context (SE Linux).
`-print`	Outputs the result of the search to the standard output. The result is usually a list of filenames, including their full pathnames.
`-type` *filetype*	Searches for files with the specified file type. File type can be **b** for block device, **c** for character device, **d** for directory, **f** for file, or **l** for symbolic link.
`-perm` *permission*	Searches for files with certain permissions set. Use octal or symbolic format for permissions.
`-ls`	Provides a detailed listing of each file, with owner, permission, size, and date information.
`-exec` *command*	Executes command when files found.

In the next example, the user copies a file called **proposal** to a new file called **oldprop**:

$ cp proposal oldprop

You can unintentionally destroy another file with the **cp** command. The **cp** command generates a copy by first creating a file and then copying data into it. If another file has the same name as the destination file, that file will be destroyed and a new file with that name created. By default Red Hat configures your system to check for an existing copy by the same name. To copy a file from your working directory to another directory, you only need to use that directory name as the

Command	Execution
`cp` *filename filename*	Copies a file. `cp` takes two arguments: the original file and the name of the new copy. You can use pathnames for the files to copy across directories: `$ cp today reports/Monday`
`cp -r` *dirname dirname*	Copies a subdirectory from one directory to another. The copied directory includes all its own subdirectories: `$ cp -r letters/thankyou oldletters`
`mv` *filename filename*	Moves (renames) a file. The `mv` command takes two arguments: the first is the file to be moved. The second argument can be the new filename or the pathname of a directory. If it is the name of a directory, then the file is literally moved to that directory, changing the file's pathname: `$ mv today /home/chris/reports`
`mv` *dirname dirname*	Moves directories. In this case, the first and last arguments are directories: `$ mv letters/thankyou oldletters`
`ln` *filename filename*	Creates added names for files referred to as links. A link can be created in one directory that references a file in another directory: `$ ln today reports/Monday`
`rm` *filenames*	Removes (erases) a file. Can take any number of filenames as its arguments. Removes links to a file. If a file has more than one link, you need to remove all of them to erase a file: `$rm today weather weekend`

second argument in the **cp** command. In the next example, the **proposal** file is overwritten by the **newprop** file. The **proposal** file already exists.

$ cp newprop proposal

You can use any of the wildcard characters to generate a list of filenames to use with **cp** or **mv**. For example, suppose you need to copy all your C source code files to a given directory. Instead of listing each one individually on the command line, you can use an ***** character with the **.c** extension to match and generate a list of C source code files (all files with a **.c** extension). In the next example, the user copies all source code files in the current directory to the **sourcebks** directory:

$ cp *.c sourcebks

If you want to copy all the files in a given directory to another directory, you can use ***** to generate a list of all those files in a **cp** command. In the next example, the user copies all the files in the **props** directory to the **oldprop** directory. Notice the use of the **props** pathname preceding the ***** special characters. In this context, **props** is a pathname that will be appended before each file in the list that ***** generates.

$ cp props/* oldprop

You can, of course, use any of the other special characters, such as **.**, **?**, or **[]**. In the next example, the user copies both source code and object code files (**.c** and **.o**) to the **projbk** directory:

$ cp *.[oc] projbk

When you copy a file, you may want to give the copy a different name than the original. To do so, place the new filename after the directory name, separated by a slash. $ cp filename directory-name/new-filename

Moving Files

You can use the **mv** command to either rename a file or move a file from one directory to another. When using **mv** to rename a file, you simply use the new filename as the second argument. The first argument is the current name of the file you are renaming. If you want to rename a file when you move it, you can specify the new name of the file after the directory name. In the next example, the **proposal** file is renamed with the name **version1**:

$ mv proposal version1

As with **cp**, it is easy for **mv** to erase a file accidentally. When renaming a file, you might accidentally choose a filename already used by another file. In this case, that other file will be erased. The **mv** command also has an **-i** option that checks first to see if a file by that name already exists.

You can also use any of the special characters described in Chapter 3 to generate a list of filenames to use with **mv**. In the next example, the user moves all C source code files in the current directory to the **newproj** directory:

$ mv *.c newproj

If you want to move all the files in a given directory to another directory, you can use ***** to generate a list of all those files. In the next example, the user moves all the files in the **reports** directory to the **repbks** directory:

$ mv reports/* repbks

Copying and Moving Directories

You can also copy or move whole directories at once. Both **cp** and **mv** can take as their first argument a directory name, enabling you to copy or move subdirectories from one directory into another. The first argument is the name of the directory to be moved or copied, while the second argument is the name of the directory within which it is to be placed. The same pathname structure used for files applies to moving or copying directories.

You can just as easily copy subdirectories from one directory to another. To copy a directory, the **cp** command requires you to use the **-r** option. The **-r** option stands for "recursive." It directs the **cp** command to copy a directory, as well as any subdirectories it may contain. In other words, the entire directory subtree, from that directory on, will be copied. In the next example, the **thankyou** directory is copied to the **oldletters** directory.

Now two **thankyou** subdirectories exist, one in **letters** and one in **oldletters**.

$ **cp -r letters/thankyou oldletters**

$ **ls -F letters**

/thankyou

$ **ls -F oldletters**

/thankyou

Erasing Files and Directories: The rm Command

As you use Linux, you will find the number of files you use increases rapidly. Generating files in Linux is easy. Applications such as editors, and commands such as **cp**, easily create files. Eventually, many of these files may become outdated and useless. You can then remove them with the **rm** command. The **rm** command can take any number of arguments, enabling you to list several filenames and erase them all at the same time. In the next example, the user erases the file **oldprop**:

$ rm oldprop

Be careful when using the **rm** command, because it is irrevocable. Once a file is removed, it cannot be restored (there is no undo). With the **-i** option, you are prompted separately for each file and asked whether to remove it. If you enter **y**, the file will be removed. If you enter anything else, the file is not removed. In the next example, the **rm** command is instructed to erase the files **proposal** and **oldprop**. The **rm** command then asks for confirmation for each file. The user decides to remove **oldprop,** but not **proposal**.

$ **rm -i proposal oldprop**

Remove proposal? **n**
Remove oldprop? **y**
$

Links: The ln Command

You can give a file more than one name using the **ln** command. You might want to reference a file using different filenames to access it from different directories. The added names are often referred to as links. Linux supports two different types of links, hard and symbolic. Hard links are literally another name for the same file, whereas symbolic links function like shortcuts referencing another file. Symbolic links are much more flexible and can work over many different file systems, whereas hard links are limited to your local file system. Furthermore, hard links introduce security concerns, as they allow direct access from a link that may have public access to an original file that you may want protected. Because of this, links are usually implemented as symbolic links.

Symbolic Links

To set up a symbolic link, you use the **ln** command with the **-s** option and two arguments: the name of the original file and the new, added filename. The **ls** operation lists both filenames, but only one physical file will exist.

$ ln -s original-file-name added-file-name

In the next example, the **today** file is given the additional name **weather**. It is just another name for the **today** file.

$ **ls**
today
$ **ln -s today weather**
$ **ls**
today weather

You can give the same file several names by using the **ln** command on the same file many times. In the next example, the file **today** is given both the names **weather** and **weekend**:

$ **ln -s today weather**

$ **ln -s today weekend**
$ **ls**
today weather weekend

If you list the full information about a symbolic link and its file, you will find the information displayed is different. In the next example, the user lists the full information for both **lunch** and **/home/george/veglist** using the **ls** command with the **-l** option. The first character in the line specifies the file type. Symbolic links have their own file type, represented by an **l**. The file type for **lunch** is **l**, indicating it is a symbolic link, not an ordinary file. The number after the term "group" is the size of the file. Notice the sizes differ. The size of the **lunch** file is only four bytes. This is because **lunch** is only a symbolic link—a file that holds the pathname of another file—and a pathname takes up only a few bytes. It is not a direct hard link to the **veglist** file.

$ **ls -l lunch /home/george/veglist**
-rw-rw-r-- 1 george group 793 Feb 14 10:30 veglist
lrw-rw-r-- 1 chris group 4 Feb 14 10:30 lunch

To erase a file, you need to remove only its original name (and any hard links to it). If any symbolic links are left over, they will be unable to access the file. In this case, a symbolic link will hold the pathname of a file that no longer exists.

Hard Links

You can give the same file several names by using the **ln** command on the same file many times. To set up a hard link, you use the **ln** command with no **-s** option and two arguments: the name of the original file and the new, added filename. The **ls** operation lists both filenames, but only one physical file will exist.

$ ln original-file-name added-file-name

In the next example, the **monday** file is given the additional name **storm**. It is just another name for the **monday** file.

$ **ls**
today
$ **ln monday storm**
$ **ls**
monday storm

To erase a file that has hard links, you need to remove all its hard links. The name of a file is actually considered a link to that file—hence the command **rm** that removes the link to the file. If you have several links to the file and remove only one of them, the others stay in place and you can reference the file through them. The same is true even if you remove the

original link—the original name of the file. Any added links will work just as well. In the next example, the **today** file is removed with the **rm** command. However, a link to that same file exists, called **weather**. The file can then be referenced under the name **weather**.

$ **ln today weather**
$ **rm today**
$ **cat weather**
The storm broke today
and the sun came out.
$

The mtools Utilities: msdos

Your Linux system provides a set of utilities, known as mtools, that enable you to easily access floppy and hard disks formatted for MS-DOS. They work only with the old MS-DOS or FAT32 file systems, not with Windows Vista, XP, NT, or 2000, which use the NTFS file system. The **mcopy** command enables you to copy files to and from an MS-DOS floppy disk in your floppy drive or a Windows FAT32 partition on your hard drive. No special operations, such as mounting, are required. With mtools, you needn't mount an MS-DOS partition to access it. For an MS-DOS floppy disk, once you place the disk in your floppy drive, you can use mtool commands to access those files. For example, to copy a file from an MS-DOS floppy disk to your Linux system, use the **mcopy** command. You specify the MS-DOS disk with **a:** for the A drive. Unlike normal DOS pathnames, pathnames used with mtool commands use forward slashes instead of backslashes. The directory **docs** on the A drive would be referenced by the pathname **a:/docs**, not **a:\docs**. Unlike MS-DOS, which defaults the second argument to the current directory, you always need to supply the second argument for **mcopy**. The next example copies the file **mydata** to the MS-DOS disk and then copies the **preface** file from the disk to the current Linux directory.

$ mcopy mydata a:
$ mcopy a:/preface

Archiving and Compressing Files

Archives are used to back up files or to combine them into a package, which can then be transferred as one file over the Internet or posted on an FTP site for easy downloading. The standard archive utility used on Linux and Unix systems is tar, for which several GUI front ends exist. You have several compression programs to choose from, including GNU zip (gzip), Zip, bzip, and compress.

Archiving and Compressing Files with File Roller

GNOME provides the File Roller tool (accessible from the Accessories menu, labeled Archive Manager) that operates as a GUI front end to archive and compress files, letting you perform Zip, gzip, tar, and bzip2 operations using a GUI. You can examine the contents of archives, extract the files you want, and create new compressed archives. When you create an archive, you determine its compression method by specifying its filename extension, such as **.gz** for gzip or **.bz2** for bzip2. You can select the different extensions from the File Type menu or enter the extension yourself. To both archive and compress files, you can choose a combined extension like **.tar.bz2**, which both archives with tar and compresses with bzip2. Click Add to add files to your archive. To extract files from an archive, open the archive to display the list of archive files. You can then click Extract to extract particular files or the entire archive.

Archive Files and Devices: tar

The tar utility creates archives for files and directories. With tar, you can archive specific files, update them in the archive, and add new files to an archive. You can even archive entire directories with all their files and subdirectories, all of which can be restored from the archive. The tar utility was originally designed to create archives on tapes. (The term “tar” stands for tape archive. However, you can create archives on any device, such as a floppy disk, or you can create an archive file to hold the archive.) The tar utility is ideal for making backups of your files or combining several files into a single file for transmission across a network (File Roller is a GUI for tar).

Displaying Archive Contents

Both file managers in GNOME and the K Desktop have the capability to display the contents of a tar archive file automatically. The contents are displayed as though they were files in a directory. You can list the files as icons or with details, sorting them by name, type, or other fields. You can even display the contents of files. Clicking a text file opens it with a text editor, and an image is displayed with an image viewer. If the file manager cannot determine what program to use to display the file, it prompts you to select an application.Both file managers can perform the same kinds of operations on archives residing on remote file systems, such as tar archives on FTP sites. You can obtain a listing of their contents and even read their readme files. The Nautilus file manager (GNOME) can also extract an archive. Right-click the Archive icon and select Extract.

Creating Archives

On Linux, tar is often used to create archives on devices or files. You can direct tar to archive files to a specific device or a file by using the **f** option with the name of the device or file. The syntax for the **tar** command using the **f** option is shown in the next example. The device or filename is often referred to as the archive name. When creating a file for a tar archive, the filename is usually given the extension **.tar**. This is a convention only and is not required. You can list as many filenames as you want. If a directory name is specified, all its subdirectories are included in the archive.

$ tar optionsf archive-name.tar directory-and-file-names

To create an archive, use the **c** option. Combined with the **f** option, **c** creates an archive on a file or device. You enter this option before and right next to the **f** option. Notice no dash precedes a tar option.In the next example, the directory **mydir** and all its subdirectories are saved in the file **myarch.tar**. In this example, the **mydir** directory holds two files, **mymeeting** and **party**, as well as a directory called **reports** that has three files: **weather**, **monday**, and **friday**.

$ **tar cvf myarch.tar mydir**

mydir/

mydir/reports/

mydir/reports/weather

Commands	Execution
`tar` *options files*	Backs up files to tape, device, or archive file.
`tar` *optionsf archive_name filelist*	Backs up files to a specific file or device specified as *archive_name*. *filelist*; can be filenames or directories.
Options	
`c`	Creates a new archive.
`t`	Lists the names of files in an archive.
`r`	Appends files to an archive.
`u`	Updates an archive with new and changed files; adds only those files modified since they were archived or files not already present in the archive.
`--delete`	Removes a file from the archive.
`w`	Waits for a confirmation from the user before archiving each file; enables you to update an archive selectively.
`x`	Extracts files from an archive.
`m`	When extracting a file from an archive, no new timestamp is assigned.
`M`	Creates a multiple-volume archive that may be stored on several floppy disks.
`f` *archive-name*	Saves the tape archive to the file archive-name, instead of to the default tape device. When given an archive name, the `f` option saves the tar archive in a file of that name.
`f` *device-name*	Saves a tar archive to a device such as a floppy disk or tape. **/dev/fd0** is the device name for your floppy disk; the default device is held in **/etc/default/tar-file**.
`v`	Displays each filename as it is archived.
`z`	Compresses or decompresses archived files using gzip.
`j`	Compresses or decompresses archived files using bzip2.

mydir/reports/monday

mydir/reports/friday

```
mydir/mymeeting
mydir/party
```

Extracting Archives

The user can later extract files and directories from the archive using the **x** option. The **xf** option extracts files from an archive file or device. The tar extraction operation generates all subdirectories. In the next example, the **xf** option directs **tar** to extract all the files and subdirectories from the tar file **myarch.tar**:

```
$ tar xvf myarch.tar
mydir/
mydir/reports/
mydir/reports/weather
mydir/reports/monday
mydir/reports/friday
mydir/mymeeting
mydir/party
```

You use the **r** option to add files to an already-created archive. The **r** option appends the files to the archive. In the next example, the user appends the files in the **letters** directory to the **myarch.tar** archive. Here, the directory **mydocs** and its files are added to the **myarch.tar** archive:

```
$ tar rvf myarch.tar mydocs
mydocs/
mydocs/doc1
```

Updating Archives

If you change any of the files in your directories you previously archived, you can use the **u** option to instruct tar to update the archive with any modified files. The **tar** command compares the time of the last update for each archived file with those in the user's directory and copies into the archive any files that have been changed since they were last archived. Any newly created files in these directories are also added to the archive. In the next example, the user updates the **myarch.tar** file with any recently modified or newly created files in the **mydir** directory. In this case, the **gifts** file is added to the **mydir** directory.

```
tar uvf myarch.tar mydir
mydir/
mydir/gifts
```

If you need to see what files are stored in an archive, you can use the **tar** command with the **t** option. The next example lists all the files stored in the **myarch.tar** archive:

```
tar tvf myarch.tar
```

```
drwxr-xr-x root/root 0 2000-10-24 21:38:18 mydir/
drwxr-xr-x root/root 0 2000-10-24 21:38:51 mydir/reports/
-rw-r--r-- root/root 22 2000-10-24 21:38:40 mydir/reports/weather
-rw-r--r-- root/root 22 2000-10-24 21:38:45 mydir/reports/monday
-rw-r--r-- root/root 22 2000-10-24 21:38:51 mydir/reports/Friday
-rw-r--r-- root/root 22 2000-10-24 21:38:18 mydir/mymeeting
-rw-r--r-- root/root 22 2000-10-24 21:36:42 mydir/party
drwxr-xr-x root/root 0 2000-10-24 21:48:45 mydocs/
-rw-r--r-- root/root 22 2000-10-24 21:48:45 mydocs/doc1
drwxr-xr-x root/root 0 2000-10-24 21:54:03 mydir/
-rw-r--r-- root/root 22 2000-10-24 21:54:03 mydir/gifts
```

Archiving to Floppies

To back up the files to a specific device, specify the device as the archive. For a floppy disk, you can specify the floppy drive. Be sure to use a blank floppy disk. Any data previously placed on it will be erased by this operation. In the next example, the user creates an archive on the floppy disk in the **/dev/fd0** device and copies into it all the files in the **mydir** directory:

```
$ tar cf /dev/fd0 mydir
```

To extract the backed-up files on the disk in the device, use the **xf** option:

```
$ tar xf /dev/fd0
```

Compressing Archives

The **tar** operation does not perform compression on archived files. If you want to compress the archived files, you can instruct tar to invoke the gzip utility to compress them. With the lowercase **z** option, tar first uses gzip to compress files before archiving them. The same **z** option invokes gzip to decompress them when extracting files.

```
$ tar czf myarch.tar.gz mydir
```

To use bzip instead of gzip to compress files before archiving them, you use the **j** option. The same **j** option invokes bzip to decompress them when extracting files.

```
$ tar cjf myarch.tar.bz2 mydir
```

Remember, a difference exists between compressing individual files in an archive and compressing the entire archive as a whole. Often, an archive is created for transferring several files at once as one tar file. To shorten transmission time, the archive should be as small as possible. You can use the compression utility gzip on the archive tar file to compress it, reducing its size, and then send the compressed version. The person receiving it can decompress it, restoring the tar file. Using gzip on a tar file often results in a file with the extension **.tar.gz**. The extension **.gz** is added to a

compressed gzip file. The next example creates a compressed version of **myarch.tar** using the same name with the extension **.gz**:

```
$ gzip myarch.tar
$ ls
$ myarch.tar.gz
```

Instead of retyping the **tar** command for different files, you can place the command in a script and pass the files to it. Be sure to make the script executable. In the following example, a simple **myarchprog** script is created that will archive filenames listed as its arguments.

```
myarchprog
tar cvf myarch.tar $*
```

A run of the **myarchprog** script with multiple arguments is shown here:

```
$ myarchprog mydata preface
mydata
preface
```

Archiving to Tape

If you have a default device specified, such as a tape, and you want to create an archive on it, you can simply use **tar** without the **f** option and a device or filename. This can be helpful for making backups of your files. The name of the default device is held in a file called **/etc/default/tar**. The syntax for the **tar** command using the default tape device is shown in the following example. If a directory name is specified, all its subdirectories are included in the archive.

```
$ tar option directory-and-file-names
```

In the next example, the directory **mydir** and all its subdirectories are saved on a tape in the default tape device:

```
$ tar c mydir
```

In this example, the **mydir** directory and all its files and subdirectories are extracted from the default tape device and placed in the user's working directory:

```
$ tar x mydir
```

File Compression: gzip, bzip2, and zip

Several reasons exist for reducing the size of a file. The two most common are to save space or, if you are transferring the file across a network, to save transmission time. You can effectively reduce a file size by creating a compressed copy of it. Anytime you need the file again, you decompress it. Compression is used in combination with archiving to enable you to compress whole directories and their files at once. Decompression generates a copy of the archive file, which can then be extracted,

generating a copy of those files and directories. File Roller provides a GUI for these tasks.

Compression with gzip

Several compression utilities are available for use on Linux and Unix systems. Most software for Linux systems uses the GNU gzip and gunzip utilities. The gzip utility compresses files, and gunzip decompresses them. To compress a file, enter the command **gzip** and the filename. This replaces the file with a compressed version of it with the extension **.gz**.

$ **gzip mydata**

$ **ls**

mydata.gz

To decompress a gzip file, use either **gzip** with the **-d** option or the command **gunzip**. These commands decompress a compressed file with the **.gz** extension and replace it with a decompressed version with the same root name but without the **.gz** extension. You needn't even type in the **.gz** extension; **gunzip** and **gzip -d** assume it. Table lists the different gzip options.

$ **gunzip mydata.gz**

$ **ls**

Mydata

You can also compress archived tar files. This results in files with the extensions **.tar.gz**. Compressed archived files are often used for transmitting extremely large files across networks.

$ **gzip myarch.tar**

$ **ls**

myarch.tar.gz

Option	Execution
`-c`	Sends compressed version of file to standard output; each file listed is separately compressed: `gzip -c mydata preface > myfiles.gz`
`-d`	Decompresses a compressed file; or you can use gunzip: `gzip -d myfiles.gz` `gunzip myfiles.gz`
`-h`	Displays help listing.
`-l` *file-list*	Displays compressed and uncompressed size of each file listed: `gzip -l myfiles.gz`
`-r` *directory-name*	Recursively searches for specified directories and compresses all the files in them; the search begins from the current working directory. When used with `gunzip`, compressed files of a specified directory are uncompressed.
`-v` *file-list*	For each compressed or decompressed file, displays its name and the percentage of its reduction in size.
-num	Determines the speed and size of the compression; the range is from –1 to –9. A lower number gives greater speed but less compression, resulting in a larger file that compresses and decompresses quickly. Thus –1 gives the quickest compression but with the largest size; –9 results in a very small file that takes longer to compress and decompress. The default is –6.

You can compress tar file members individually using the **tar z** option that invokes gzip. With the **z** option, tar invokes gzip to compress a file before placing it in an archive.Archives with members compressed with the **z** option, however, cannot be updated, nor is it possible to add to them. All members must be compressed, and all must be added at the same time.

The compress and uncompress Commands

You can also use the **compress** and **uncompress** commands to create compressed files.They generate a file that has a **.Z** extension and use a different compression format from gzip. The **compress** and **uncompress** commands are not that widely used, but you may run across **.Z** files occasionally. You can use the **uncompress** command to decompress a **.Z** file. The gzip utility is the standard GNU compression utility and should be used instead of **compress**.

Compressing with bzip2

Another popular compression utility is **bzip2**. It compresses files using the Burrows-Wheeler block-sorting text compression algorithm and Huffman coding. The command line options are similar to gzip by design, but they are not exactly the same. (See the bzip2 Man page for a complete listing.) You compress files using the **bzip2** command and decompress with **bunzip2**. The **bzip2** command creates files with the extension **.bz2**. You can use **bzcat** to output compressed data to the standard output. The **bzip2** command compresses files in blocks and enables you to specify their size. As when using gzip, you can use bzip2 to compress tar archive files. The following example compresses the **mydata** file into a bzip compressed file with the extension **.bz2**:

```
$ bzip2 mydata
$ ls mydata.bz2
```

To decompress, use the **bunzip2** command on a bzip file:

```
$ bunzip2 mydata.bz2
```

Using Zip

Zip is a compression and archive utility modeled on PKZIP**,** which was used originally on DOS systems. Zip is a cross-platform utility used on Windows, Mac, MS-DOS, OS/2, Unix, and Linux systems. Zip commands can work with archives created by PKZIP and can use Zip archives. You compress a file using the **zip** command. This creates a Zip file with the **.zip** extension. If no files are listed, **zip** outputs the compressed data to the standard output. You can also use the - argument to have **zip** read from the standard input. To compress a directory, you include the **-r** option. The first example archives and compresses a file:

```
$ zip mydata
```

$ **ls**

mydata.zip

The next example archives and compresses the **reports** directory:

$ zip -r reports

A full set of archive operations is supported. With the **-f** option, you can update a particular file in the Zip archive with a newer version. The **-u** option replaces or adds files, and the **-d** option deletes files from the Zip archive. Options also exist for encrypting files, making DOS-to-Unix end-of-line translations and including hidden files.

To decompress and extract the Zip file, you use the **unzip** command.

$ unzip mydata.zip

Review & Self Assessment Question:

Q1- What is Linux Files ?

Q2- What do you mean by File Structure ?

Q3-What is Pathname ?

Q4-Explain the commands for Creating and Deleting Directories ?

Further Readings

Linux Operating System Richard Petersen

Linux Operating System Paul S. Wang

Linux Operating System by David Maxwell and Andrew Bedford

Linux Operating System by Richard Blum and Christine Bresnahan

Linux Operating System by Bhatt P.C.P

UNIT: 7- NETWORKING, INTERNET, AND THE WEB

Contents

Introduction

Early packet-switched computer networking, involving a few research institutions and government agencies, started in the late 1960s and early 1970s.Today, it is hard to tell where the computer ends and the network begins. The view "The Network is the Computer" is more valid than ever. Most people cannot tolerate even a few minutes of Internet connection outage.

A computer network is a high-speed communications medium connecting many, possibly dissimilar, computers or hosts. A network is a combination of computer and telecommunication hardware and software. The purpose is to provide fast and reliable information exchange among the hosts. Typical services made possible by a network include

• Electronic mail

• On-line chatting and Internet phone calls

• File transfer

• Remote login

- Distributed databases
- Networked file systems
- Audio and video streaming
- Voice and telephone over a network
- World Wide Web, E-business, E-commerce, and social networks
- Remote procedure and object access

In addition to host computers, the network itself may involve dedicated computers that perform network functions: hubs, switches, bridges, routers, and gateways. A network extends greatly the powers of the connected hosts.

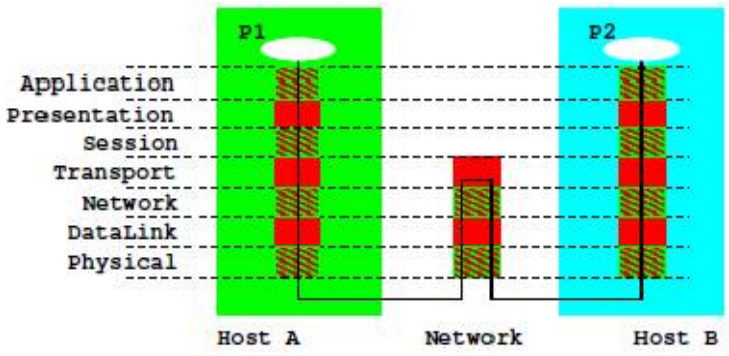
A good understanding of basic networking concepts, commands, information security, and how the Web works will be important for any Linux user/programmer.

Networking Protocols

For programs and computers from different vendors, under different operating systems, to communicate on a network, a detailed set of rules and conventions must be established for all parties to follow.Such rules are known as networking protocols.We use different networking services for different purposes; therefore, each network service follows its own specific protocols. Protocols govern such details as

- Address format of hosts and processes
- Data format
- Manner of data transmission
- Sequencing and addressing of messages
- Initiating and terminating connections
- Establishing services
- Accessing services
- Data integrity, privacy, and security

Thus, for a process on one host to communicate with another process on a different host, both processes must follow the same protocol. The Open System Interconnect (OSI) Reference Model provides a standard layered view of networking protocols and their interdependence. The corresponding layers on different hosts, and inside the network infrastructure, perform complementary tasks to make the connection between the communicating processes .



Among common networking protocols, the Internet Protocol Suite is the most widely used. The basic IP (Internet Protocol) is a network layer protocol.The TCP (Transport Control Protocol) and UDP (User Datagram Protocol) are at the transport layer. The Web is a service that uses an application layer protocol known as HTTP (the Hypertext Transfer Protocol).

Networking protocols are no mystery. Think about the protocol for making a telephone call. You (a client process) must pick up the phone, listen for the dial tone, dial a valid telephone number, and wait for the other side (the server process) to pick up the phone. Then you must say "hello," identify yourself, and so on. This is a protocol from which you cannot deviate if you want the call to be made successfully through the telephone network, and it is clear why such a protocol is needed. The same is true of a computer program attempting to talk to another computer program through a computer network. The design of efficient and effective networking protocols for different network services is an important area in computer science.

Chances are your Linux system is on a Local Area Network (LAN) which is connected to the Internet. This means you have the ability to reach, almost instantaneously, across great distances to obtain information, exchange messages, upload/download files, interact with others, do literature searches,and much more without leaving the seat in front of your workstation. If your computer is not directly connected to a network but has a telephone or cable modem, then you can reach the Internet through an Internet service provider (ISP).

The Internet

The Internet is a global network that connects computer networks using the Internet Protocol (IP). The linking of computer networks is called internetworking, hence the name Internet. The Internet links all kinds of organizations around the world: universities, government offices, corporations, libraries, supercomputer centers, research labs, and

individual homes. The number of connections on the Internet is large and growing rapidly.

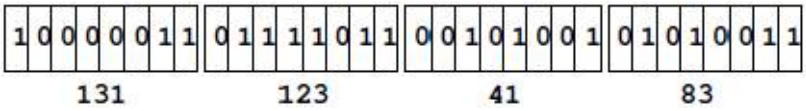
The Internet evolved from the ARPANET,1 a U.S. Department of Defense Advanced Research Projects Agency (DARPA) sponsored network that developed the IP as well as the higher level Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) networking protocols. The architecture and protocol were designed to support a reliable and flexible network that could endure wartime attacks.

The transition of ARPANET to the Internet took place in the late 1980s as NSFnet, the U.S. National Science Foundation's network of universities and it has virtually eliminated all historical rivals such as BITNET and DECnet.

The Internet Corporation for Assigned Names and Numbers (ICANN) is a nonprofit organization responsible for IP address space allocation, protocol parameter assignment, domain name system management, and maintaining root server system functions.

Network Addresses

An address to a host computer is like a phone number to a telephone. Every host on the Internet has its own network address that identifies the host for communication purposes. The addressing technique is an important part of a network and its protocol. An Internet address (IP address) is represented by 4 bytes in a 32-bit quantity. For example, monkey, a host at Kent State, has the IP address 131.123.41.83 . This dot notation (or quad notation)



gives the decimal value (0 to 255) of each byte.2 The IP address is similar to a telephone number in another way: the leading digits are like area codes, and the trailing digits are like local numbers.

Because of their numerical nature, the dot notation is easy on machines but hard on users. Therefore, each host may also have a domain name composed of words, rather like a postal address. For example, the domain name for monkey is monkey.cs.kent.edu (at the Department of Computer Science, Kent State University). The Linux command host displays the IP and domain name of any given host. For example,

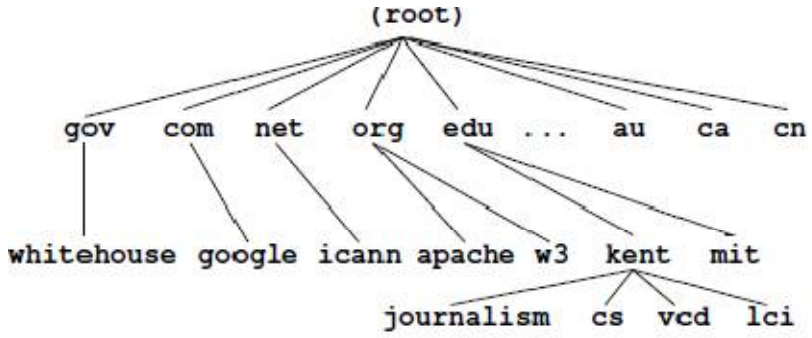
host monkey.cs.kent.edu

displays

monkey.cs.kent.edu is an alias for monkey.zodiac.cs.kent.edu.

monkey.zodiac.cs.kent.edu has address 131.123.41.83

With domain names, the entire Internet name space for hosts is recursively divided into disjoint domains in a hierarchical tree (Figure 7.3). The address for monkey puts it in the cs local domain, within the kent subdomain, which is under the edu top-level domain (TLD) for U.S. educational institutions.



Other TLDs include org (nonprofit organizations), gov (U.S. government offices), mil (U.S. military installations), com (commercial outfits), net (network service providers), uk (United Kingdom), cn (China), and so forth. Within a local domain (for example, cs.kent.edu), you can refer to machines by their hostname alone (for example, monkey, dragon, tiger), but the full address must be used for machines outside.

The ICANN accredits domain name registrars, which register domain names for clients so they stay distinct. All network applications accept a host address given either as a domain name or as an IP address. In fact, a domain name is first translated to a numerical IP address before being used.

Packet Switching

Data on the Internet are sent and received in packets. A packet envelops transmitted data with address information so the data can be routed through intermediate computers on the network. Because there are multiple routes from the source to the destination host, the Internet is very reliable and can operate even if parts of the network are down.

Client and Server

Most commonly, a network application involves a server and a client

• A server process provides a specific service on a host machine that offers such a service. Example services are email (SMTP), secure remote host access (SSH), secure file transfer (SFTP), and the World Wide Web (HTTP). Each Internet standard service has its own unique port number that is identical on all hosts. The port number together with the Internet address of a host indentifies a particular server program any where on the network. For example, SFTP has port number 115, SSH has 22, and HTTP has 80. On your Linux system, the file /etc/services lists the

standard and additional network services, indicating their protocols and port numbers.

• A client process on a host connects with a server on another host to obtain its service. Thus, a client program is the agent through which a particular network service can be obtained. Different agents are usually required for different services.

AWeb browser such as Firefox is an HTTP client. It runs on your computer to access Web servers on any Internet hosts. The Linux wget command is another useful client that can download files from the Internet using the HTTP or the FTP protocol.

The Domain Name System

As stated in previous Section, every host on the Internet has a unique IP address and a domain name. The network name space, the set of all domain names with their associated IP addresses, changes dynamically with time due to the addition and deletion of hosts, regrouping of local work groups, reconfiguration of subparts of the network, maintenance of systems and networks, and so on. Thus, new domain names, new IP addresses, and new domain to- IP associations can be introduced in the name space at any time without central control. The domain name system (DNS) is a network service that supports dynamic update and retrieval of information contained in the distributed name space. A network client program (for example, the Firefox browser) will normally use the DNS to obtain IP address information

for a target host before making contact with a server. The dynamic DNS also supplies a general mechanism for retrieving many kinds of information about hosts and individual users.

Here are points to note about the DNS name space:

• The DNS organizes the entire Internet name space into a big tree structure. Each node of the tree represents a domain and has a label and a list of resources.

• Labels are character strings (currently not case sensitive), and sibling labels must be distinct. The root is labeled by the empty string. Immediately below the root are the TLDs: edu, com, gov, net, org, info, and so on. TLDs also include country names such as at (Austria), ca (Canada), and cn (China). Under edu, for example, there are subdomains berkeley, kent, mit, uiuc, and so on .

• A full domain name of a node is a dot-separated list of labels leading from the node to the root.

• A relative domain name is a prefix of a full domain name, indicating a node relative to a domain of origin. Thus, cs.kent.edu is actually a name relative to the root.

• A label is the formal or canonical name of a domain. Alternative names, called aliases, are also allowed. For example, the main Web server host info has the alias www, so it is also known as www.cs.kent.edu. To move the Web server to a different host, a local system manager reassigns the alias to another host.

Networking in Nautilus

Linux systems are listed individually. Systems running other operating systems are grouped under different icons such as the Windows Network icon. Of course, you can browse only machines with permission. Normally, login will be required unless you have arranged a no-password login

For example, these Locations work:

• sftp

• ssh

• sftp

Accessing Samba Shared Files

Usually, you'll find Linux and MS Windows R systems on the same in-house network. Nautilus makes it easy to access shared files from MS Windows R. Just enter the Location

smb://host/share folder

to reach the target shared folder via the Common Internet File System protocol, the successor of Server Message Block (SMB). Linux systems use SaMBa, a free, open-source implementation of the CIFS file sharing protocol, to act as server and client to MS Windows R systems. Use an IP for the host to be sure. Here ae some Location examples on a home network.

smb://192.168.2.102/SharedDocs

smb://192.168.2.107/Public

Networking Commands

Linux offers many networking commands. Some common ones are described here to get you started. In earlier chapters, we mentioned briefly several networking commands. For example, we know that

hostiname

displays the domain name of the computer you are using. If given an argument, this command can also set the domain name (when run as root), but the domain name is usually only set at system boot time. To get the IP

address and other key information from the DNS about your computer or another host, you can use

host $(hostname) (for your computer)

host targetHost (for target host)

For example, host google.com produces

google.com has address 74.125.45.100
google.com has address 74.125.67.100
google.com has address 209.85.171.100
google.com mail is handled by 10 smtp4.google.com.
google.com mail is handled by 10 smtp1.google.com.
google.com mail is handled by 10 smtp2.google.com.
google.com mail is handled by 10 smtp3.google.com.

For any given host, its DNS data provide IP address, canonical domain name, alias domain names, DNS server hosts, and email handling hosts. Other commands that help you access the DNS data from the command line include nslookup and dig (DNS Information Groper).

For example, dig monkey.cs.kent.edu

gives

; <<>> DiG 9.5.0-P2 <<>> monkey.cs.kent.edu
;; QUESTION SECTION:
;monkey.cs.kent.edu. IN A
;; ANSWER SECTION:
monkey.cs.kent.edu.1800 IN CNAME monkey.zodiac.cs.kent.edu.
monkey.zodiac.cs.kent.edu. 43200 IN A 131.123.41.83
;; AUTHORITY SECTION:
zodiac.cs.kent.edu. 300 IN NS ns.cs.kent.edu.
zodiac.cs.kent.edu. 300 IN NS ns.math.kent.edu.
;; Query time: 152 msec
;; SERVER: 192.168.2.1#53(192.168.2.1)

The desired information (ANSWER section) together with the identity of the name server (SERVER) that provided the data is displayed.

The command dig is very handy for verifying the existence of hosts and finding the IP address or domain name aliases of hosts. Once the name of a host is known, you can also test if the host is up and running, as far as networking is concerned, with the ping command.

ping host

This sends a message to the given remote host requesting it to respond with an echo if it is alive and well.

To see if any remote host is up and running, you can use ping, which sends an echo Internet control message to the remote host. If the echo

comes back, you'll know that the host is up and connected to the Internet. You'll also get round-trip times and packet loss statistics. When successful, the ping commands continues to send echo packets. Type ctrl+c to quit.

SSH with X11 Forwarding

Networking allows you to conveniently access Linux systems remotely. Most Linux distributions come with OpenSSH installed. As mentioned in Chapter 1, Section 1.2, you can ssh to a remote Linux and use it from the command line. Furthermore, you can

ssh -X userid @remoteHostname

to log in to the given remote host with X11 forwarding/tunneling, which allows you to start any X applications, such as gedit or gnome-terminal, on the remote host and have the graphical display appear on your local desktop.

This works if your local host is a Linux/UNIX/MacOS system. It can also work from MS Windows R. Follow these steps:

1. Obtain and install an X11 server on Windows, such as the Xming or the heavier duty Cygwin.
2. Assuming you have downloaded and installed Xming, click the Xming icon to launch the X11 server. The X11 server displays an icon on your start panel so you know it is running.
3. Set up SSH or Putty on your Windows R system:

• Putty Settings—Go to Connection->SSH->X11 and check the Enable X11 forwarding box. Also set X display location to 127.0.0.1:0.0.

• SSH Settings—Check the Tunneling->Tunnel X11 Connections box. Also check the Authentication->Enable SSH2 connections box.

4. Use either Putty or SSH to connect to a remote Linux/Unix computer. Make sure your remote account login script, such as .bash profile, does not set the DISPLAY environment variable. It will be set for you to something like localhost:10.0 automatically when you connect via SSH.
5. Make sure your X11 server (Xming for example) is running. Now, if you start an X application on the remote Linux system, that graphical application will then SSH tunnel to your PC and use the X11 server on your PC to display a graphical user interface (GUI). For example, you can start gedit, nautilus --no-desktop, or even firefox.

No Password ssh, sftp, and scp

The commands ssh, sftp, and scp are for remote login, secure ftp, and secure remote cp, respectively. When using any of these you usually need to enter the password for the remote system interactively. When you need

to perform such tasks frequently, this can be a bother. Fortunately, you can easily avoid having to enter the password. Just follow these steps.

Most Linux systems come with Open SSH installed. This means you already have the SSH suite of commands. These enable you to securely communicate from one computer (as user1 on host1) to another (as user2 on host2). We will assume you are logged in as user1 on host1 (this is your local host), and you wish to arrange secure communication with your account user2 on host2, which we will refer to as the remote host.

SSH can use public-key encryption for data security and user authentication. If you have not done it yet, the first step in arranging for password-less login is to generate your own SSH keys. Issue the command

ssh-keygen

You'll be asked for a folder to save the keys and a passphrase to access them. In this case, don't provide any input in response to these questions from sshkeygen. Simply press the enter key in response to each question.

Key generation takes a little time. Then you'll see a message telling you that your identity (private key) is id_rsa and your public key is id_rsa.pub saved under the standard folder .ssh. in your home directory.

The second step is to copy your id_rsa.pub to your account on the desired remote-host. Issue the command

ssh-copy-id -i ~/.ssh/id_rsa.pub your userid @ remote-host

to append your public SSH key to the file ~userid/.ssh/authorized_key on the remote-host. Now you are all set. You can log in to remote-host without entering a password.

ssh userid @ remote-host

The same setup avoids a password when you use sftp or scp.

Remote File Synchronization

The rsync command makes it easy to keep files in sync between two hosts. It is very efficient because it uses a remote-update protocol to transfer just the differences between two sets of files across the network connection. No updating is performed for files with no difference. With the commands

rsync -az userid @ host:source destDir (remote to local sync)

rsync -az source userid @ host:destDir (local to remote sync)

the given source file/folder is used to update the same under the destination folder destDir. When source is a folder, the entire hierarchy rooted at the folder will be updated.

The -az option indicates the commonly used archive mode to preserve file types and modes and gzip data compression to save networking bandwidth. The rsync tool normally uses ssh for secure data

transfer and does not require a password if you have set up password-less SSH between the two hosts .

For example,

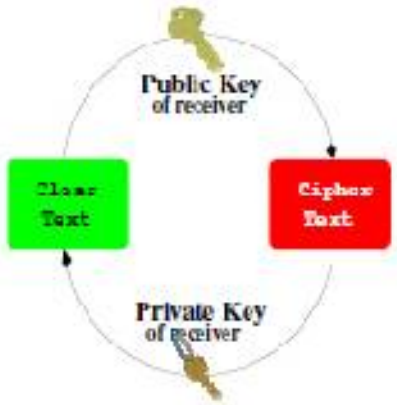
rsync -az pwang@monkey.cs.kent.edu:~/linux book ~/projects/

updates the local folder ~/projects/linux_book based on the remote folder ~/linux_book by logging in as pwang on the remote host monkey.cs.kent.edu. See the rsync man page for complete documentation.

Public-Key Cryptography and Digital Signature

Security is a big concern when it comes to networking. From the user's viewpoint, it is important to keep data and network transport secure and private. Public-key cryptography is an essential part of the modern network security infrastructure to provide privacy and security for many networking applications. Before the invention of public-key cryptography, the same secret key had to be used for both encryption and decryption of a message (symmetric-key cryptography). Symmetric-key is fine and efficient, and remains in widespread use today. However, a secret key is hard to arrange among strangers never in communication before; for example, parties on the Internet. The public-key cryptography breakthrough solves this key distribution problem elegantly.

GnuPG (GNU Privacy Guard), part of OpenGP, supports public-key cryptography.The Linux command for GnuPG is gpg or the largely equivalent gpg2. With gpg, you can generate a public key that you share with others and a private key you keep secret. You and others can use the public key to encrypt files and messages which only you can decrypt using the private key .



Using your private key, you can also attach a digital signature to any message/file. A receiver can verify the integrity (not altered) and

authenticity (really from the sender) of the the signed message. To do all that, make sure you first set up GnuPG and your personal keys.

If your Linux distribution does not already provide gpg, you can easily install the gnupg package (Section 8.24) with either of the following commands:

sudo apt-get install gnupg (Ubuntu/Debian)

sudo yum install gnupg (CentOS/Fedora)

If you like to use a GUI for gpg, install also the gpa package. However, the command-line interface is entirely adequate.

Setting Up GnuPG Keys

To use gpg, you first need to generate your public-private key pair.

gpg --gen-key

You'll be prompted to enter your choices for keytype (pick the default), keysize (pick 2048), and a passphrase (pick something you won't forget, but will be very hard for anyone to guess). The passphrase is required each time you access your private key, thus preventing others from using your private key.

You'll get a keyid displayed when your key pair is generated. Your keys and other info are stored by default in the folder $HOME/.gnupg. Use

gpg --list-public-keys

to display your public keys. For example,

pub 1024D/FCF2F84D 2009-07-25

uid Paul Wang (monkeykia) <pwang@cs.kent.edu>

sub 1024g/B02C4B40 2009-07-25

The pub line says that Paul's public master key (for signature) is a 1024-bit DSA key with id FCF2F84D and that his public subkey (for data encryption) is a 1024-bit ElGama key.

To enable others to encrypt information to be delivered for your eyes only, you should send your public keys to a public key server. The command

gpg --send-keys your keyid

sends your public key to a default gpg key server, such as

hkp://subkeys.pgp.net

Also, you can send your public keys to anyone by sending them an ASCII armored file generated by

gpg --armor --export your keyid > mykey.asc

The .asc suffix simply indicates that a file is an ASCII text file. The mykey.asc contains your key encoded using base64, a way to use 64 ASCII characters (AZ, az, 09 and +/) to encode non-ASCII files for easy communication over networks, especially via email. The Linux base64

command performs this encoding/decoding on any file. See man base64 for more information.

Such ASCII armored key files can be emailed to others or sent to another computer and imported to another GnuPG key ring with a command such as

gpg --import mykey.asc

Also, edit your $HOME/gnupg/gpg.conf file and append the line

default-key your keyid

Encryption/Decryption with GnuPG

To encrypt a file using a public key of uid,

gpg --encrypt -r uid

resulting in an encrypted file filename.gpg that can be sent to the target user who is the only one that can decrypt it.

Even if you are not going to send a file to anyone, you can still keep secrets in that file of yours protected in case someone gains unauthorized access to your computer account. You can

gpg --encrypt -r "your uid" filename

rm filename

generating the encrypted filename.gpg and removing the original filename. You can easily view the encrypted version with

nano < (gpg –decrypt filename.gpg)

Note that the Bash process expansion is handy here.

To make maintaining an encrypted file even easier, you may configure vi/vim to work transparently with gpg, allowing you to use vim to view and edit clear as well as gpg encrypted files. The VIM extension tGpg (yet another plug-in for encrypting files with gpg) is a good choice for this purpose.

Secure Email with Mutt and GnuPG

The Linux email client mutt works well with GnuPG to support s/mime (Secure/Multipurpose Internet Mail Extensions), allowing you to send and receive encrypted/signed email.

Assuming that you have arranged for your keys and sent your public keys to a key server as described in Section 7.7 and that your email correspondents are also set up with GnuPG or some other public-key system for their s/mime, you can easily use mutt to exchange emails securely with them.

Follow these steps to set up mutt.

1. Locate the file gpg.rc for mutt on your Linux.Usually, you’ll find it at /usr/share/doc/mutt-version/gpg.rc

2. Edit your mutt configuration file $HOME/.muttrc and add at the end a line to include the gpg.rc source /usr/share/doc/mutt-version/gpg.rc

3. Import your secure email correspondents' keys into your GnuPG key ring. Get your email correspondent to send you an ASCII armored key file or search for the key on the key server the --search-keys option:

gpg --import someKey.asc

gpg --search-keys targetEmailAddress

Now, you can encrypt/sign email after composing the email message by using the p key within mutt to select from the following options:

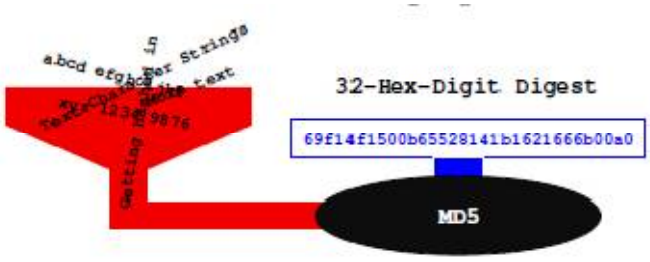
* encrypt
* sign
* both
* sign as

Receiving encrypted/signed email with mutt is just a matter of following on-screen instructions.

The popular email client Thunderbird also works with GnuPG if you install the Enigmail extension (via the tools->add-ons).

Message Digests

A message digest is a digital fingerprint of a message or file. Various algorithms have been devised to take a message (file) of any length and reduce it to a short fixed-length hash known as the digest of the original message or file.



These algorithms are designed to produce a different digest if any part of the message is altered. It is almost impossible to deduce the original message from knowledge of the digest. However, because there are an infinite number of possible messages but only a finite number of different digests, vastly different messages may produce the same digest.

Message digests are therefore useful in verifying the integrity (unalteredness) of files. When software is distributed online, a good practice is to display a fingerprint for the file, allowing you to check the integrity of the download and to avoid any Trojan horse code.

MD5 is a popular algorithm producing 128-bit message digests. An MD5 hash is usually displayed as a sequence of 32 hexadecimal

digits. On Linux, you can produce an MD5 digest with the md5sum command

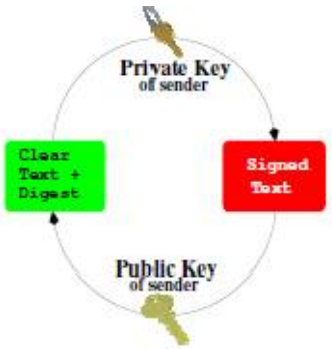
md5sum filename > digestFile

You'll get a digestFile file containing only the hash and the name filename. After downloading both filename and digestFile, a user can check file integrity with md5sum digestFile

Other digest algorithms in wide use include SHA-1 and others. The Linux command sha1sum is an alternative to md5sum.

Message Signing with GnuPG

To digitally sign a particular message, a message digest is created first. The message digest is then encrypted using your private key to produce a digital



signature which is attached to the message. Any receiver of a signed message can generate a message digest from the received message and check it against the digest obtained by decrypting the digital signature with the signer's public key.A match verifies the integrity and the authenticity of the received message.

gpg --sign file (produces signed binary file.gpg)

gpg --clearsign file (produces signed ASCII file.asc)

The --decrypt option automatically verifies any attached signature.

The Web

Out of all the networking applications, the Web is perhaps one of the most important and deserves our special attention.

There is no central control or administration of the Web. Anyone can potentially put material on the Web and retrieve information from it. The Web consists of a vast collection of documents that are located on computers throughout the world. These documents are created by academic, professional, government, and commercial organizations, as well as by individuals. The documents are prepared in special formats and delivered through Web servers, programs that return documents in

response to incoming requests. Linux systems are often used to run Web servers.

Primarily, Web documents are written in Hypertext Markup Language. Each HTML document can contain (potentially many) links to other documents served by different servers in other locations and therefore become part of a web that spans the entire globe. New materials are put on the Web continuously, and instant access to this collection of information can be enormously advantageous. As the Web grew, MIT (Massachusetts Institute of Technology, Cambridge, MA) and INRIA (the French National Institute for Research in Computer Science and Control) agreed to become joint hosts of the W3 Consortium, a standards body for the Web community.

A Web browser is a program that helps users obtain and display information from the Web. Given the location of a target document, a browser connects to the correct Web server and retrieves and displays the desired document. You can click links in a document to obtain other documents. Using a browser, you can retrieve information provided by Web servers anywhere on the Internet.

Typically, a Web browser, such as Firefox, supports the display of HTML files and images in standard formats. Helper applications or plug-ins can augment a browser to treat pages with multimedia content such as audio, video, animation, and mathematical formulas.

Hypertext Markup Language

A Web browser communicates with a Web server through an efficient HTTP designed to work with hypertext and hypermedia documents that may contain regular text, images, audio, and video. Native Web pages are written in the HTML and usually saved in files with the .html (or .htm) suffix.

HTML organizes Web page content (text, graphics, and other media data) and allows hyperlinks to other pages anywhere on the Web. Clicking such a link causes your Web browser to follow it and retrieve another page. The Web employs an open addressing scheme that allows links to objects and services provided by Web, email, file transfer, audio/video, and newsgroup servers. Thus, the Web space is a superset of many popular Internet services. Consequently, a Web browser provides the ability to access a wide variety of information and services on the Internet.

URLs

The Web uses Uniform Resource Locators (URLs) to identify (locate) resources (files and services) available on the Internet. A URL

may identify a host, a server port, and the target file stored on that host. URLs are used, for example, by browsers to retrieve information and by HTML to link to other resources.

A full URL usually has the form

scheme://server:port/pathname

The scheme part indicates the information service type and therefore the protocol to use. Common schemes include http (Web service), ftp (file transfer service), mailto (email service), file (local file system), https (secure Web service), and sftp (secure file transfer service).

For example,

sftp://pwang@monkey.cs.kent.edu/users/cs/faculty/pwang

gets you the directory list of /users/cs/faculty/pwang. This works on Firefox and on the Linux file browser nautilus, assuming that you have set up your SSH/SFTP.

For URLs in general, the server identifies a host and a server program.The optional port number is needed only if the server does not use the default port.The remainder of the URL, when given, is a file pathname. If this pathname has a trailing / character, it represents a directory rather than a data file. The suffix (.html, .txt, .jpg, etc.) of a data file indicates the file type. The pathname can also lead to an executable program that dynamically produces an HTML or other valid file to return.

Within an HTML document, you can link to another document served by the same Web server by giving only the pathname part of the URL. Such URLs are partially specified. A partial URL with a / prefix (for example, /file xyz.html) refers to a file under the server root, the top-level directory controlled by the Web server. A partial URL without a leading / points to a file relative to the location of the document that contains the URL in question.Thus, a simple file abc.html refers to that file in the same directory as the current document. When building a website, it is advisable to use a URL relative to the current page as much as possible, making it easy to move the entire website folder to another location on the local file system or to a different server host.

Accessing Information on the Web

You can directly access any Web document, directory, or service by giving its URL in the Location box of a browser. When given a URL that specifies a directory, a Web server usually returns an index file (typically, index.html) for that directory. Otherwise, it may return a list of the filenames in that directory.

You can use a search engine such as Google to quickly look for information on the Web.

Handling Different Content Types

On the Web, files of different media types can be placed and retrieved. The Web server and Web browser use standard content type designations to indicate the media type of files in order to process them correctly.

The Web borrowed the content type designations from the Internet email system and uses the same MIME (Multipurpose Internet Mail Extensions) defined content types. There are hundreds of content types in use today. Many popular types are associated with standard file extensions.

When a Web server returns a document to a browser, the content type is indicated. The content type information allows browsers to decide how to process the incoming content. Normally, HTML, text, and images are handled by the browser directly. Others types such as audio and video are usually handled by plug-ins or external helper programs.

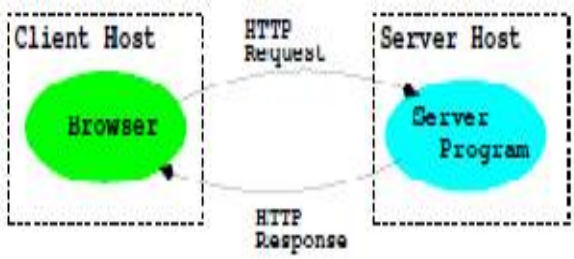
Putting Information on the Web

Now let's turn our attention to how information is supplied on the Web. The understanding sheds more light on how the Web works and what it takes to serve up information.

The Web puts the power of publishing in the hands of anyone with a computer connected to the Internet. All you need is to run a Web server on this machine and establish files for it to service.

Major computer vendors offer commercial Web servers with their computer systems. Apache is a widely used open-source Web server that is freely available from the Apache Software Foundation.

Linux systems are especially popular as Web hosting computers because Linux is free, robust, and secure. Also, there are many useful Web-related applications such as Apache, PHP (active Web page), MySQL (database server), and more available free of charge.



Once a Web server is up and running on your machine, all types of files can be served. On a typical Linux system, follow these simple steps to make your personal Web page.

1. Make a file directory in your home directory (~userid/public html) to contain your files for theWeb. This is your personal Web directory. Make this directory publicly accessible:

chmod a+x ~userid/public html

When in doubt, ask your system managers about the exact name to use for your personal Web directory.

2. In your Web directory, establish a home page, usually index.html, in HTML. The home page usually functions as an annotated table of contents. Make this file publicly readable:

chmod a+r ~userid/public html/index.html

3. Place files and directories containing desired information in your personal Web directory. Make each directory and each file accessible as before. Refer to these files with links in the home page and other pages.

4. Let people know the URL of your home page, which is typically

http://your-sever/~your-userid/

In a Web page, you can refer to another file of yours with a simple link containing a relative URL (<a href="filename">), where filename can be either a simple name or a pathname relative to the current document.

Among the Web file formats, hypertext is critical because it provides a means for a document to link to other documents.

What Is HTML?

HTML (the Hypertext Markup Language) is used to markup the content of a Web page to provide page structure for easy handling by Web clients on the receiving end. Since HTML 4.0, the language has become standardized. XHTML (XML compatible HTML) is the current stable version. However, a new standard HTML5 is fast approaching.

A document written in HTML contains ordinary text interspersed with markup tags and uses the .html filename extension. The tags mark portions of the text as title, section header, paragraph, reference to other documents, and so on. Thus, an HTML file consists of two kinds of information: contents and HTML tags. A browser follows the HTML tags to layout the page content for display. Because of this, line breaks and extra white space between words in the content are mostly ignored. In addition to structuring and formatting contents, HTML tags can also reference graphics images, link to other documents, mark reference points, generate forms or questionnaires, and invoke certain programs. Various visual editors or page makers are available that provide a GUI for creating and designing HTML documents. For substantial website creation projects, it will be helpful to use integrated development environments

such as Macromedia Dreamweaver . If you don't have ready access to such tools, a regular text editor can create or edit Web pages.

Marked As	HTML Tags
Entire document	<html>...</html>
Header part of document	<head>...</head>
Document title	<title>...</title>
Document content	<body>...</body>
Level *n* heading	<h*n*>...</h*n*>
Paragraph	...
Unnumbered list	...
Numbered list	...
List item	...
Comment	<!--...-->

An HTML tag takes the form <tag>. A begin tag such as <h1> (level-one section header) is paired with an end tag, </h1> in this case, to mark content in between.

The following is a sample HTML page (Ex: ex07/Fruits):

```
<html>
<head>
<title>A Basic Web Page</title>
</head>
<body>
<h1>Big on Fruits</h1>
<p>Fruits are good tasting and good for you ...</p>
<p> There are many varieties, ... and here is a short list: </p>
<ol>
<li> Apples </li>
<li> Bananas </li>
<li> Cherries </li>
</ol>
 </body></html>
```

Web Hosting

Web hosting is a service to store and serve ready-made files and programs so that they are accessible on the Web. Hence, publishing on the Web involves

1. Designing and constructing the pages and writing the programs for a website
2. Placing the completed site with a hosting service

Colleges and universities host personal and educational sites for students and faculty without charge. Web hosting companies provide the service for a fee.

Commercial Web hosting can provide secure data centers (buildings), fast and reliable Internet connections, specially tuned Web hosting computers (mostly Linux boxes), server programs and utilities, network and system security, daily backup, and technical support. Each hosting account provides an amount of disk space, a monthly network traffic allowance, email accounts, Web-based site management and maintenance tools, and other access such as FTP and SSH/SFTP.

To host a site under a given domain name, a hosting service associates that domain name to an IP number assigned to the hosted site. The domain to- IP association is made through DNS servers and Web server configurations managed by the hosting service.

Domain Registration

To obtain a domain name, you need the service of a domain name registrar.Most will be happy to register your new domain name for a very modest yearly fee. Once registered, the domain name is property that belongs to the registrant. No one else can register for that particular domain name as long as the current registrant keeps the registration in good order.

ICANN accredits commercial registrars for common TLDs, including .com, .net, .org, and .info. Additional TLDs include .biz, .pro, .aero, .name, and .museum.Restricted domains (for example, .edu, .gov, and .us) are handled by special registries (for example, net.educause.edu, nic.gov and respective countries.

Accessing Domain Registration Data

The registration record of a domain name is often publicly available. The standard Internet whois service allows easy access to this information. On Linux systems, easy access to whois is provided by the whois command

whois domain_name

which lists the domain registration record kept at a registrar.

For example, ‘

whois kent.edu

produces the following information

Registrant:

Kent State University
500 E. Main St.
Kent, OH 44242
UNITED STATES

Technical Contact:

Administrative Contact:

Bob Hart
Mgr., Network & Telecomm
Kent State University
120 Library Bldg
Kent, OH 44242
UNITED STATES
(330) 672-0385
pki-admin@kent.edu
Name Servers:
NS.NET.KENT.EDU 131.123.1.1
DHCP.NET.KENT.EDU 131.123.252.2
Domain record activated: 19-Feb-1987
Domain record last updated: 17-Mar-2009
Domain expires: 31-Jul-2009

On Linux systems, the whois command is sometimes called jwhois.

The DNS

DNS provides the ever-changing domain-to-IP mapping information on the Internet.We mentioned that DNS provides a distributed database service that supports dynamic retrieval of information contained in the name space. Web browsers and other Internet client applications will normally use the DNS to obtain the IP of a target host before making contact with a server over the Internet.

There are three elements to the DNS: the DNS name space, the DNS servers, and the DNS resolvers.

DNS Servers

Information in the distributed DNS is divided into zones, and each zone is supported by one or more name servers running on different hosts. A zone is associated with a node on the domain tree and covers all or part of the sub tree at that node. A name server that has complete information for a particular zone is said to be an authority for that zone. Authoritative information is automatically distributed to other name servers that provide redundant service for the same zone. A server relies on lower level servers for other information within its sub domain and on external servers for other zones in the domain tree. A server associated with the root node of the domain tree is a root server and can lead to information anywhere in the DNS. An authoritative server uses local files to store information, to locate key servers within and without its domain, and to cache query results from other servers. A boot file, usually /etc/named.boot, configures a name server and its data files.

The management of each zone is also free to designate the hosts that run the name servers and to make changes in its authoritative database. For example, the host ns.cs.kent.edu may run a name server for the domain cs.kent.edu.

A name server answers queries from resolvers and provides either definitive answers or referrals to other name servers. The DNS database is set up to handle network address, mail exchange, host configuration, and other types of queries, with some to be implemented in the future.

The ICANN and others maintain root name servers associated with the root node of the DNS tree. In fact, the VeriSign host a.root-servers.net runs a root name server. Actually, the letter a ranges up to m for a total of 13 root servers currently.

Domain name registrars, corporations, organizations, Web hosting companies, and other Internet service providers (ISPs) run name servers to associate IPs to domain names in their particular zones. All name servers on the Internet cooperate to perform domain-to-IP mappings on the fly.

DNS Resolvers

A DNS resolver is a program that sends queries to name servers and obtains replies from them. On Linux systems, a resolver usually takes the form of a C library function. A resolver can access at least one name server and use that name server's information to answer a query directly or pursue the query using referrals to other name servers.

Resolvers, in the form of networking library routines, are used to translate domain names into actual IP addresses. These library routines, in turn, ask prescribed name servers to resolve the domain names. The name servers to use for any particular host are normally specified in the file /etc/resolv.conf or /usr/etc/resolv.conf.

The DNS service provides not just the IP address and domain name information for hosts on the Internet.

Type	Description
A	Host's IP address
NS	Name servers of host or domain
CNAME	Host's canonical name, and an alias
PTR	Host's domain name, IP
HINFO	Host information
MX	Mail exchanger of host or domain
AXFR	Request for zone transfer
ANY	Request for all records

Dynamic Generation of Web Pages

Documents available on the Web are usually prepared and set in advance to supply some fixed content, either in HTML or in some other format such as plain text, PDF, or JPEG. These fixed documents are static. A

Web server can also generate documents on the fly that bring these and other advantages:

• Customizing a document depending on when, where, who, and what program is retrieving it

• Collecting user input (with HTML forms) and providing responses to the incoming information

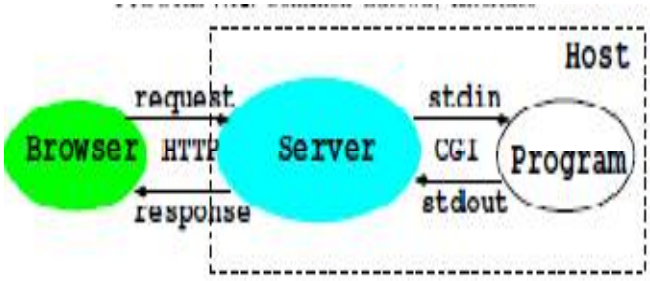
• Enforcing certain policies for outgoing documents

• Supplying contents such as game scores and stock quotes, which are changing by nature.

Dynamic Web pages are not magic. Instead of retrieving a fixed file, a Web server calls another program to compute the document to be returned. As you may have guessed, not every program can be used by a Web server in this manner. There are two ways to add server-side programming:

• Load programs directly into the Web server to be used whenever the need arises.

• Call an external program from the server, passing arguments to it (via the program's stdin and environment variables) and receiving the results (via the program's stdout) thus generated. Such a program must conform to the Common Gateway Interface (CGI) specifications governing how the Web server and the external program interact.



Dynamic Server Pages

The dynamic generation of pages is made simpler and more integrated with Web page design and construction by allowing a Web page to contain active parts that are treated by the Web server and transformed into desired content on the fly as the page is retrieved and returned to a client browser.

The active parts in a page are written in some kind of notation to distinguish them from the static parts of a page. The ASP (Active Server Pages, JSP (Java Server Pages), and the popular PHP (Hypertext Preprocessor) are examples.

Because active pages are treated by modules loaded into the Web server, the processing is faster and more efficient compared to CGI programs. Active page technologies such as PHP also provide form processing,

HTTP sessions, and easy access to databases. Therefore, they offer complete server-side support for dynamic Web pages.

Both CGI and server pages can be used to support HTML forms, the familiar fill-out forms you often see on the Web.

HTTP Briefly

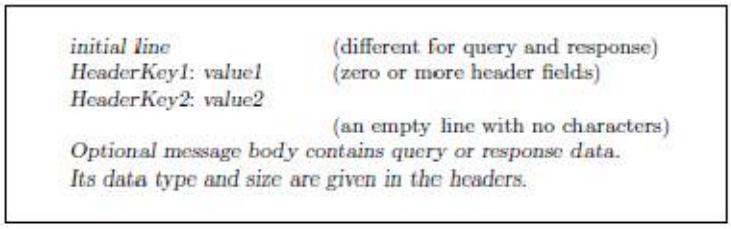
On the Web, browser-server communication follows HTTP. A basic understanding of HTTP is important for Linux programmers because Linux systems are very popular Web server hosts.

The start of HTTP traces back to the beginning of the Web in the early 1990s. HTTP/1.0 was standardized early in 1996. Improvements and new features have been introduced and HTTP/1.1 is now the stable version.

Here is an overview of an HTTP transaction:

1. Connection—A browser (client) opens a connection to a server.
2. Query—The client requests a resource controlled by the server.
3. Processing—The server receives and processes the request.
4. Response—The server sends the requested resource back to the client.
5. Termination—The transaction is finished, and the connection is closed unless another transaction takes place immediately between the client and server.

HTTP governs the format of the query and response messages.



The header part is textual, and each line in the header should end in return and newline, but it may end in just newline.

The initial line identifies the message as a query or a response.

• A query line has three parts separated by spaces: a query method name,a local path of the requested resource, and an HTTP version number.

For example,

GET /path/to/file/index.html HTTP/1.1

or

POST /path/script.cgi HTTP/1.1

The GET method requests the specified resource and does not allow a message body. A GET method can invoke a server-side program by specifying the CGI or active-page path, a question mark, and then a query string:

GET/cgi-bin/newaddr.cgi?name=value1&email=value2 HTTP/1.1

Host: monkey.cs.kent.edu

Unlike GET, the POST method allows a message body and is designed to work with HTML forms for collecting input from Web users.

• A response (or status) line also has three parts separated by spaces: an HTTP version number, a status code, and a textual description of the status. Typical status lines are

HTTP/1.1 200 OK

for a successful query or

HTTP/1.1 404 Not Found

when the requested resource cannot be found.

• The HTTP response sends the requested file together with its content type and length (optional) so the client will know how to process it.

A Real HTTP Experience

Let's manually send an HTTP request and get an HTTP response. To do that we will use the nc commnad. The command nc provides command-line (and scripting) access to the basic TCP and UDP and therefore allows you to make any TCP connections or send any UDP packets. Such abilities are usually reserved to programs at the C-language level that set up sockets for networking.

For example, the simple Bash pipeline (Ex: ex07/poorbr.sh)

```
echo $ ´GET /WEB/test.html HTTP/1.0\n ´|
nc monkey.cs.kent.edu 80
```

retrieves the Web page monkey.cs.kent.edu/WEB/test.html. In this example, we applied the Bash string expansion.

Note the HTTP get request asks for the file /WEB/test.html under the document root folder managed by the Web server on monkey. The request is terminated by an empty line, as required by the HTTP protocol.

Try this and you'll see the result display.

```
HTTP/1.1 200 OK
Date: Tue, 07 Apr 2009 19:45:03 GMT
Server: Apache/2.0.54 (Fedora)
X-Powered-By: PHP/5.0.4
Cache-Control: max-age=86400
Expires: Wed, 08 Apr 2009 19:45:03 GMT
Vary: Accept-Encoding
Content-Length: 360
Connection: close
Content-Type: text/html; charset=UTF-8
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 //EN">
```

<html> AND THE REST OF THE HTML PAGE
</html>

As you can see from the HTTP response, the Web server on monkey is Apache version 2 running under Fedora, a Linux system.

For downloading from the Web, you don't need to rely on our little pipeline. The wget command takes care of that need nicely. Wget supports HTTP, HTTPS, and FTP protocols and can download single files or follow links in HTML files and recursively download entire websites for offline viewing. The wget command can continue to work after you log out so you can download large amounts of data without waiting.

For More Information

• IPv6 is the next-generation Internet protocol. See www.ipv6.org/ for an overview.

• The official website for Gnu Privacy Guard is www.gnupg.org, and for OpenSSH, is www.openssh.com.

• Public-Key Cryptography Standards (PKCS) can be found at RSA Laboratories (www.rsa.com/rsalabs).

• HTML5 is the new and coming standard for HTML. See the specification at W3C.

• The DNS is basic to keeping services on the Internet and Web running. Find our more about DNS at www.dns.net/dnsrd/docs/.

• HTTP is basic to the Web. See RFC 1945 for HTTP 1.0 and RFC 2068 for HTTP 1.1.

Review & Self Assessment Question:

Q1- What is Computer Network?

Q2- Define the term Internet?

Q3- What do you mean by Domain Name System?

Q4-What do you mean by Remote File Synchronization?

Q5-What is Message Digests?

Q6- What is DNS Server? Explain it.

Further Readings

Linux Operating System Richard Petersen

Linux Operating System Paul S. Wang

Linux Operating System by David Maxwell and Andrew Bedford

Linux Operating System by Richard Blum and Christine Bresnahan

Linux Operating System by Bhatt P.C.P

UNIT: 8- PROGRAMMING IN LINUX

Contents

Introduction

With a basic understanding of commands, Shell usage and programming, structure of the file system, networking, and Web hosting.

The Linux system and many of its commands are written in the C language.C is a compact and efficient general-purpose programming language that has evolved together with UNIX and Linux. Thus, C is regarded as the native language for Linux. The portability of Linux is due, in large part, to the portability of C.

Because of its importance, C has been standardized by the American National Standards Institute (ANSI) and later by the International Organization for Standardization (ISO). The latest standard is known as ISO C99. The C99 standard specifies language constructs and a Standard C Library API (Application Programming Interface) for common operations, such as I/O (input/ output) and string handling. Code examples in this book are compatible with ISO C99.

On most Linux distributions, you'll find

• gcc (or g++)—The compiler from GNU that compiles C (or C++) programs. These include support for ISO C99 and ISO C++ code.

• glibc—The POSIX2-compliant C library from GNU. A library keeps common code in one place to be shared by many programs. The glibc library package contains the most important sets of shared libraries: the standard-compliant C library, the math library, as well as national language (locale) support.

On Linux, it is easy to write a C program, compile it with gcc, and run the resulting executable. For creating and editing short programs, such as examples in this book, simple text editors like gedit and nano are fine.More capable editors such as vim and emacs have C editing modes for easier coding. Integrated Development Environments (IDEs) for C/C++ on Linux, such as kdevelop, Anjula, and Borland C++, are also available to manage larger programming projects.

In this and the next two chapters, we will look at facilities for programming at the C-language level and write C code to perform important operating system tasks including I/O, file access, piping, process control, inter-process communications, and networking. The material presented will enable you to implement new commands in C, as well as control and utilize the Linux kernel through its C interface.

A collection of basic topics that relates to writing C code under Linux is explored in this chapter:

- Command-line argument conventions
- Actions of the C compiler
- Standard C Libraries
- Use and maintenance of program libraries
- Error handling and recovery
- Using the gdb debugger

Command-Line Arguments

Commands in Linux usually are written either as Shell scripts or as C programs. Arguments given to a command at the Shell level are passed as character strings to the main function of a C program. A main function expecting arguments is normally declared as follows:

```
int main(int argc, char *argv[])
```

The parameter argc is an integer. The notation

```
char *argv[ ]
```

declares the formal array parameter argv as having elements of type char * (character pointer). In other words, each of the array elements argv[0], argv[1], ..., argv[argc-1] points to a character string. The meanings of the formal arguments argc and argv are as follows:

argc—The number of command-line arguments, including the command name

argv[n]—A pointer to the nth command-line argument as a character string If the command name is cmd, and it is invoked as

```
cmd arg1 arg2
 then
argc            is 3
```

argv[0]	points to the command name cmd
argv[1]	points to the string arg1
argv[2]	points to the string arg2
argv[3]	is 0 (NULL)

The parameters for the function main can be omitted (int main( )) if they are not needed.

Now let's write a program that receives command-line arguments .To keep it simple, all the program does is echo the command line arguments to standard output.

```
/****** the echo command ******/
 #include <stdlib.h>
  #include <stdio.h>
 int main(int argc, char *argv[])
        {int i = 1; /* begins with 1 */
          while (i < argc)
        {printf("%s", argv[i++]); /* outputs string */
         printf(" "); /* outputs SPACE */
        }printf("\n"); /* terminates output line */
         return EXIT_SUCCESS; /* returns exit status */
        }
```

The program displays each entry of argv except argv[0], which is actually the command name itself. The string format %s of printf is used. To separate the strings, the program displays a space after each argv[i], and the last argument is followed by a newline.

Exit Status

Note that main is declared to return an int and the last statement in the preceding example returns a constant defined in <stdlib.h>

```
return EXIT_SUCCESS;
```

When a program terminates, an integer value, called an exit status, is returned to the invoking environment (a Shell, for example) of the program. The exit status indicates, to the invoker of the program, whether the program executed successfully and terminated normally. An exit status EXIT_SUCCESS (0 on Linux) is normal, while EXIT_FAILURE (1 on Linux), or any other small positive integer, indicates abnormal termination. At the Linux Shell level, for example, different actions can be taken depending on the exit status (value of $?) of a command. For a C program, the return value of main, or the argument to a call to exit, specifies the exit status. Thus, main should always return an integer exit status even though a program does not need the quantity for its own purposes.

Compile and Execute

To compile C programs, use gcc. For example,

gcc echo.c -o myecho

Here, the executable file produced is named myecho, which can be run with

myecho To be or not to be

producing the display

To be or not to be The argv[0] in this case is myecho.

The command gcc runs the GNU C Compiler (GCC).

Linux Command Argument Conventions

Generally speaking, Linux commands use the following convention for specifying arguments:

command [options] [files]

Options are given with a single or double hyphen (-) prefix.

-char

--word

where char is a single letter and word is a full word. For example, the ls command has the single-letter -F and the full-word --classify option. A command may take zero or more options. When giving more than one option, the single-letter options sometimes can be combined by preceding them with a single -.

For example,

ls -l -g -F

can be given alternatively as

ls –lgF

Some commands such as ps and tar use options, but do not require a leading hyphen.Other options may require additional characters or words to complete the specification. The -f (script file) option of the sed command is an example.

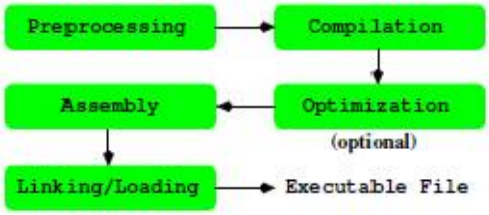
A file argument can be given in any one of the three valid filename forms: simple name, relative pathname, and full pathname. A program should not expect a restricted filename or make any assumptions about which form will be supplied by a user.

The GCC Compiler

To program in C, it is important to have a clear idea of what the C compiler does and how to use it. A compiler not only translates programs into machine code to run on a particular computer, it also takes care of arranging suitable run-time support for the program by providing I/O, file access, and other interfaces to the operating system. Therefore, a compiler

is not only computer hardware specific, but also operating system specific.

On Linux, the C compiler will likely be GCC, which is part of the GNU compiler collection. The C compiler breaks the entire compilation process into five phases



1. Preprocessing—The first phase is performed by the cpp (C preprocessor) program (or gcc -E). It handles constant definition, macro expansion,file inclusion, conditionals, and other preprocessor directives.

2. Compilation—Taking the output of the previous phase as input, the compilation phase performs syntax checking, parsing, and assembly code (.s file) generation.

3. Optimization—This optional phase specializes the code to the computer's hardware architecture and improves the efficiency of the generated code for speed and compactness.

4. Assembly—The assembler program as takes .s files and creates object (.o) files containing binary code and relocation information to be used by the linker/loader.

5. Linking—The collect2/ld program is the linker/loader which combines all object files and links in necessary library subroutines as well as runtime support routines to produce an executable program (a.out).

The gcc command can automatically execute all phases or perform only designated phases.

The gcc Command

Because of the close relationship between C and Linux, the gcc command is a key part of any Linux system. The gcc supports traditional as well as the standard ISO C99.

Typically, the gcc command takes C source files (.c and .h), assembly source files (.s), and object files (.o) and produces an executable file, named a.out by default. The compiling process will normally also produce a corresponding object file (but no assembly file) for each given source file.

Once compiled, a C program can be executed. The command name is simply the name of the executable file (if it is on the command search

PATH). For all practical purposes, an executable file is a Linux command.

Options for gcc

You can control the behavior of gcc by command-line options. A select subset of the available options is described here.

Please note that some options, such as -D and -I, have no space between the option and the value that follows it.

-E	Performs preprocessing only, outputs to stdout.
-S	Produces assembly code files (.s).
-c	Produces object (.o) files. No linking or a.out is done. This option is used for separate compilation of component modules in a program package.
g or -ggdb	Includes debugging information in object/executable code for gdb and other debuggers.
-o filename	Names the executable file filename instead of a.out.
-O,-O2,-O3	Activates the optimization phase and performs level 1, 2, or 3 optimization. The generated code will have increasingly improved speed and, most likely, also a smaller size. Optimization algorithms slow the compiler down considerably. Apply this option only after your code has been tested and debugged and the code is ready for production use
-llibname	Specifies libname as a library file to use when linking and loading the executable file. This option is passed by gcc to the linker/loader.
-Ldir	Adds dir to the library search path.
-std=standard	Uses the given standard for C such as ansi or c99.
-v	Displays the names and arguments of all subprocesses invoked in the different phases of gcc.
-Dname=str	Initializes the cpp macro name to the given string str. This command-line option is equivalent to inserting #define name str at the beginning of a source file. If =str is omitted, name is initialized to 1.
-Idir	Adds the directory dir to the directory list that gcc searches for #include files. The compiler searches first in the directory containing the source file, then in any directories specified by the -I option, and then in a list of standard system directories. Multiple -I options establish an ordered sequence of additional #include file directories.

-pg Prepares to generate an execution profile to be used with the Linux gprof utility.

The C Preprocessor

The C preprocessor (the cpp command) performs the first phase of the compilation process. The preprocessor provides important facilities that are especially important for writing system programs. Directives to the C preprocessor begin with the character # in column one. The directive

#include

is used to include other files into a source file before actual compilation begins. The included file usually contains constant, macro, and data structure definitions that usually are used in more than one source code file. The directive

#include "filename"

instructs cpp to include the entire contents of filename . If the filename is not given as a full pathname, then it is first sought in the directory where the source code containing the #include statement is located; if it is not found there, then some standard system directories are searched. If you have header files in non-standard places, use the -I option to add extra header search directories. The directive

#include <filename>

has the same effect, except the given filename is found in standard system directories. One such directory is /usr/include. For example, the standard header file for I/O is usually included by

#include <stdio.h>

at the beginning of each source code file. As you will see, an important part of writing a system program is including the correct header files supplied by Linux in the right order. The cpp directive #define is used to define constants and macros. For example, after the definitions

#define TRUE 1

#define FALSE 0

#define TABLE SIZE 1024

these names can be used in subsequent source code instead of the actual numbers. The general form is

#define identifier token . . .

The preprocessor will replace the identifier with the given tokens in the source code. If no tokens are given, identifier is defined to be 1. Macros with parameters also can be defined using the following form:

#define identifier(arg1, arg2, ...) token ...

For example,

#define MIN(x,y) ((x) > (y) ? (y) : (x))

defines the macro MIN, which takes two arguments. The macro call

MIN(a + b, c - d)

is expanded by the preprocessor into

((a+b) > (c-d) ? (c-d) : (a+b))

The right-hand side of a macro may involve symbolic constants or another macro. It is possible to remove a defined identifier and make it undefined by

#undef identifier

The preprocessor also handles conditional inclusion, where sections of source code can be included in or excluded from the compiling process, depending on certain conditions that the preprocessor can check. Conditional inclusion is specified in the general form

#if-condition
source code lines A
[#else
source code lines B]
#endif

If Condition	Meaning
`#if constant-expression`	True if the expression is non-zero
`#ifdef` *identifier*	True if *identifier* is `#defined`
`#ifndef` *identifier*	True if *identifier* is not `#defined`

If the condition is met, source code A is included; otherwise, source code B (if given) is included.

Conditional inclusion can be used to include debugging code with something like

#ifdef DEBUG
printf(...)
#endif

To activate such conditional debug statements, you can either add the line

#define DEBUG

at the beginning of the source code file or compile the source code file with

gcc -DDEBUG file ‘

Preventing Multiple Loading of Header Files

In larger C programs, it is common practice to have many source code and header files.The header files often have #include lines to include other headers.This situation often results in the likelihood of certain header files being read more than once during the preprocessing phase.This is not only wasteful,but can also introduce preprocessing errors.To avoid possible multiple inclusion, a header file can be written as a big conditional

inclusion construct.

```
/* A once only header file xyz.h */
        #ifndef __xyz_SEEN__
        #define __xyz_SEEN__
        /* the entire header file*/
         .
         .
         .
         #endif /* __xyz_SEEN__ */
```

The symbol __xyz_SEEN__ becomes defined once the file xyz.h is read by cpp (Ex: ex09/gcd.h). This fact prevents it from being read again due to the #ifndef mechanism. This macro uses the underscore prefix and suffix to minimize the chance of conflict with the other macros or constant name.

Compilation

The compiling phase takes the output of the preprocessing phase and performs parsing and code generation. If a -O option is given, then the code generation invokes code optimization routines to improve the efficiency of the generated code. The output of the compilation phase is assembly code.

Assembly

Assembly code is processed by the assembler as to produce relocatable object code (.o).

Linking and Loading

Linking/loading produces an executable program (the a.out file) by combining user-supplied object files with system-supplied object modules contained in libraries as well as initialization code needed. GCC uses collect2 to gather all initialization code from object code files and then calls the loader ld to do the actual linking/loading. The collect2/ld program treats its command-line arguments in the order given. If the argument is an object file, the object file is relocated and added to the end of the executable binary file under construction. The object file's symbol table is merged with that of the binary file. If the argument is the name of a library, then the library's symbol table is scanned in search of symbols that match undefined names in the binary file's symbol table. Any symbols found lead to object modules in the library to be loaded. Such library object /bin/bash: inking: command not found the same way. Therefore, it is important that a library argumentbe given after the names of object files that reference symbols defined in the library.

To form an executable, run-time support code (such as crt1.o, crti.o,crtbegin.o, crtend.o in /usr/lib/ or /usr/lib64/) and C library code (such as libgcc.a) must also be loaded. The correct call to collect2/ld is generated by gcc.

After all object and library arguments have been processed, the binary file's symbol table is sorted, looking for any remaining unresolved references. The final executable module is produced only if no unresolved references remain.

There are a number of options that collect2/ld takes. A few important ones are listed:

-lname Loads the library file libname.a, where name is a character string. The loader finds library files in standard system directories (normally /lib, /usr/lib, and /usr/local/lib) and additional directories specified by the -L option. The -l option can occur anywhere on the command line, but usually occurs at the end of a gcc or collect2/ld command. Other options must precede filename arguments.

-Ldir Adds the directory dir in front of the list of directories to find library files.

-s Removes the symbol table and relocation bits from the executable file to save space. This is used for code already debugged.

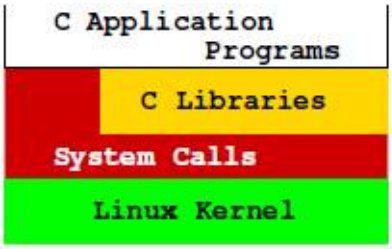
-o name Uses the given name for the executable file, instead of a.out.

The C Library

The C library provides useful functions for many common tasks such as I/O and stringn handling. In the given table lists frequently used POSIX-compliant libraries. However, library functions do depend on system calls to obtain operating system kernel services.

Functions	Header	Library File
I/O: **fopen**, **putc**, **fprintf**, **fscanf**, ...	<stdio.h>	*standard*
String: **strcpy**, **strcmp**, **strtok**, ...	<string.h>	*standard*
Character: **isupper**, **tolower**, ...	<ctype.h>	*standard*
Control: **exit**, **abort**, **malloc**, ...	<stdlib.h>	*standard*
ASCII conversion: **atoi**, **atol**, **atod**, ...	<stdlib.h>	*standard*
Error handling: **perror**, EDOM, errno, ...	<errno.h>	*standard*
Time/Date: **time**, **clock**, **ctime**, ...	<time.h>	*standard*
Mathematical: **sin**, **log**, **exp**, ...	<math.h>	-lm

An application program may call the library functions or invoke system calls directly to perform tasks. The above figure shows the relations among the Linux kernel, system calls, library calls, and application programs in C. By using standard library calls as much as possible, a C application program can achieve more system independence.



The program in next figure implements a command lowercase, which copies all characters from standard input to standard output while mapping (a one-to-one transformation) all uppercase characters to lowercase ones. The I/O routines getchar and putchar are used (Ex: ex09/lowercase.c).The C I/O library uses a FILE structure to represent I/O destinations referred to as C streams. A C stream contains information about the open file, such as the buffer location, the current character position in the buffer, the mode of access, and so on.

```
#include <stdlib.h>
#include <stdio.h>

int main()
{   int c;
    while ( (c = getchar()) != EOF )
          putchar( tolower(c) );
    return EXIT_SUCCESS;
}
```

As mentioned before, when a program is started under Linux, three I/O streams are opened automatically. In a C program, these are three standard C stream pointers stdin (for standard input from your keyboard), stdout (for standard output to your terminal window), and stderr (for standard error to your terminal window). The header file <stdio.h> contains definitions for the identifiers stdin, stdout, and stderr. Output to stdout is buffered until a line is terminated (by \n), but output to stderr is sent directly to the terminal window without buffering. Standard C streams may be redirected to files or pipes. For example,

putc(c, stderr)

writes a character to the standard error. The routines getchar and putchar can be defined as

```
#define getchar() getc(stdin)
#define putchar(c) putc(c, stdout)
```

I/O to Files

The I/O library routine fopen is used to open a file for subsequent I/O:

FILE *fopen(char *filename, char *access_mode)

This function prototype describes the arguments and return value of fopen. We will use the prototype notation to introduce C library and Linux system calls.

To open the file passed as the second command-line argument for reading, for example, you would use

```
FILE *fp = fopen(argv[2], "r");
```

The allowable access modes are listed in the next table The file is assumed to be a text file unless the mode letter b is given after the initial mode letter (r, w or a) to indicate a binary file. I/O with binary files can be very efficient for certain applications, as we will see in the next section. Now let's explain how to use the update modes.

Mode	Opens file for
"r"	reading
"w"	writing, discarding existing contents
"a"	appending at end
"r+"	*updating* (both reading and writing)
"w+"	updating, discarding existing contents
"a+"	updating, writing at end

Because the C stream provides its own buffering, sometimes there is a need to force any output data that remains in the I/O buffer to be sent out without delay. For this the function

```
int fflush(FILE *stream)
```

is used. This function is not intended to control input buffering.

File Updating

When the same file is opened for both reading and writing under one of the modes r+, w+, and a+, the file is being updated in place; namely, you are modifying the contents of the file. In performing both reading and writing under the update mode, care must be taken when switching from reading to writing and vice versa. Before switching either way, an fflush or a filepositioning function (fseek, for example) must be called on the stream. These remarks will become clear as we explain how the update modes work.

The r+ mode is most efficient for making one-for-one character substitutions in a file. Under the r+ mode, file contents stay the same if not explicitly modified. Modification is done by moving a file position indicator (similar to a cursor in a text editor) to the desired location in the file and writing the revised characters over the existing characters already there. A lowercase command based on file updating can be implemented by following the steps:

1. Open the given file with the r+ mode of fopen.
2. Read characters until an uppercase letter is encountered.
3. Overwrite the uppercase letter with the lowercase letter.

4. Repeat steps 2 and 3 until end-of-file is reached.

```
/******** lower.c ********/
#include <stdlib.h>
 #include <stdio.h>
#include <ctype.h>
#define SEEK_SET 0
 int main(int argc, char *argv[])
 {FILE *update;
int fpos; /* read or write position in file */
 char c;
 if ((update = fopen(argv[1], "r+")) == NULL)
 { fprintf(stderr, "%s: cannot open %s for updating\n",
  argv[0], argv[1]);
exit(EXIT_FAILURE);
 }
 while ((c = fgetc(update)) != EOF)
  { if ( isupper(c) )
{ ungetc(c, update); /* back up 1 char (a) */
fpos = ftell(update); /* get current pos (b) */
 fseek(update, fpos, SEEK_SET); /* pos for writing (c) */
 fputc(tolower(c), update);
 fpos = ftell(update);
fseek(update, fpos, SEEK_SET); /* pos for reading (d) */
 }
  } /* (e) */
fclose(update);
  return EXIT_SUCCESS;
}
```

After detecting an uppercase character, the file position is on the next character to read. Thus, we need to reposition the write indicator to the previous character in order to overwrite it. This is done here by backing up one character with ungetc (line a), recording the current position (line b), and setting the write position with fseek (line c) before putting out the lowercase character. Having done that, we can continue to process the rest of the file. However, we must set the read position with fseek (line d) before switching back to reading again.

The general form of the file position setting function fseek is

```
int fseek(FILE *stream, long offset, int origin)
```

The function normally returns 0, but returns -1 for error. After fseek, a subsequent read or write will access data beginning at the new position.

For a binary file, the position is set to offset bytes from the indicated origin, which can be one of the symbolic constants

SEEK_SET (usually 0) the beginning of the file
SEEK_CUR (usually 1) the current position
SEEK_END (usually 2) the end of the file

For a text stream, offset must be zero or a value returned by ftell, which gives the offset of the current position from the beginning of the file.

After end-of-file is reached, any subsequent output will be appended at the end of the file. Thus, if more output statements were given after (line e) in our example, the output would be appended to the file.

The w+ mode is used for more substantial modifications of a file. A file, opened under w+, is read into a memory buffer and then reduced to an empty file. Subsequent read operations read the buffer and write operations add to the empty file. The mode a+ also gives you the ability to read and write the file, but positions the write position initially at the end of the file.

I/O Redirection

The standard library function freopen

FILE *freopen(char *file, char *mode, FILE * stream)

connects an existing stream, such as stdin, stdout, or stderr, to the given file. Basically, this is done by opening the given file as usual but, instead of creating a new stream, assigning stream to it. The original file attached to stream is closed. For example, the statement

freopen("mydata", "r", stdin);

causes your C program to begin reading "mydata" as standard input. A successful freopen returns a FILE *.

For example, after the previous freopen, the code

char c = getc(stdin);

reads the next character from the file mydata instead of the keyboard.

A similar library function fdopen connects a file descriptor , rather than a stream, to a file in the same way.

A Linux system provides the Standard C Library, the X Window System library, the networking library, and more. The available library functions are all described in section 3 of the man pages.

Creating Libraries and Archives

We have mentioned that collect2/ld also links in libraries while constructing an executable binary file. Let’s take a look at how a library is created and maintained under the Linux system. Although our discussion is oriented toward the C language and C functions, libraries for other languages under Linux are very similar.

A subroutine library usually contains the object code versions of functions that are either of general interest or of importance for a specific project. The idea is to avoid reinventing the wheel and to gather code that has already been written, tested, and debugged in a program library, just like books in an actual library, for all to use. Normally, the library code is simply loaded together with other object files to form the final executable program.

On Linux, a library of object files is actually one form of an archive file, a collection of several independent files arranged into the archive file format. A magic number identifying the archive file format is followed by the constituent files, each preceded by a header. The header contains such information as filename, owner, group, access modes, last modified time, and so on. For an archive of object files (a library), there is also a table of contents in the beginning identifying what symbols are defined in which object files in the archive. The command ar is used to create and maintain libraries and archives.

The general form of the ar command is

ar key [position] archive-name file ...

Ar will create, modify, display, or extract information from the given archivename, depending on the key specified. The name of an archive file normally uses the .a suffix. Some more important keys are listed here.

r	To put the given files into the new or existing archive file, archivename. If a file is already in the archive, it is replaced. New files are appended at the end.
q	To quickly append the given files to the end of a new or existing archive file, archive-name, without checking whether a file is already in the archive. This is useful for creating a new archive and to add to a very large archive.
ru	Same as r, except existing files in the archive are only replaced if they are older than the corresponding files specified on the command line.
ri or ra	After ri or ra, a position argument must be supplied, which is the name of a file in the archive. These are the same as r, except new files are inserted before (ri) or after (ra) the position file in the archive.
t	To display the table of contents of the archive file.
x	To extract the named files in the archive into the current directory.This, of course, will result in one

or several independent files. If no list of names is given, all files will be extracted.

For example, the command

```
ar qcs libme.a file1.o file2.o file3.o
```

creates the new archive file libme.a by combining the given object files. The c modifier tells ar to create a new archive and the s modifier causes a table of contents (or index) to be included.

The command

```
ar tv libme.a
```

displays the table of contents of libme.a.

```
rw-rw-r-- 0/0 1240 Jul 9 16:18 2009 file1.o
rw-rw-r-- 0/0 1240 Jul 9 16:18 2009 file2.o
rw-rw-r-- 0/0 1240 Jul 9 16:18 2009 file3.o
```

If you do not wish or have permission to locate the libme.a file in a system library directory, you can put the library in your own directory and give the library name explicitly to gcc for loading. For example,

```
gcc -c myprog.c
gcc myprog.o libme.a
```

Note that myprog.c needs to include the header for libme.a, say, me.h, in order to compile successfully.

Error Handling in C Programs

An important aspect of system programming is foreseeing and handling errors that may occur during program execution. Many kinds of errors can occur at run time. For example, the program may be invoked with an incorrect number of arguments or unknown options. A program should guard against such errors and display appropriate error messages. Error messages to the user should be written to the stderr so that they appear on the terminal even if the stdout stream has been redirected. For example,

```
fprintf (stderr, "%s: cannot open %s\n", argv[0], argv[i]);
```

alerts the user that a file supplied on the command line cannot be opened. Note that it is customary to identify the name of the program displaying the error message. After displaying an error message, the program may continue to execute, return a particular value (for example, -1), or elect to abort. To terminate execution, the library routine exit(status);is used, where status is of type int. For normal termination, status should be zero. For abnormal terminal, such as an error, a positive integer status (usually 1) is used. The routine exit first invokes fclose on each open file before executing the system call exit, which causes immediate termination

without buffer flushing.A C program may use exit(status); directly if desired.

Errors from System and Library Calls

A possible source of error is failed system or library calls. A system call is an invocation of a routine in the Linux kernel. Linux provides many system calls, and understanding them is a part of learning Linux system programming. When a system or library call fails, the called routine will normally not terminate program execution. Instead, it will return an invalid value or set an external error flag. The error indication returned has to be consistent with the return value type declared for the function. At the same time, the error value must not be anything the function would ever return without failure. For library functions, the standard error values are

• EOF—The error value EOF, usually -1, is used by functions normally returning a non-negative number.

• NULL—The error value NULL, usually 0, is used by functions normally returning a valid pointer (non-zero).

• nonzero—A non-zero error value is used for a function that normally returns zero.

It is up to your program to check for such a returned value and take appropriate actions. The following idiom is in common use:

```
if ( (value = call(...)) == errvalue )
{ /* handle error here */
/* output any error message to stderr */
 }
```

Failed Linux system calls return similar standard errors -1, 0, and so on.

To properly handle system and library call errors, the header file <errno.h> should be included.

```
#include <errno.h>
```

This header file defines symbolic error numbers and their associated standard error messages.

No.	Name	Message	No.	Name	Message
1	EPERM	Not owner	27	EFBIG	File too large
2	ENOENT	No such file/dir	28	ENOSPC	No space on device
3	ESRCH	No such process	29	ESPIPE	Illegal seek
4	EINTR	Interrupted system call	30	EROFS	Read-only file system
5	EIO	I/O error	31	EMLINK	Too many links
6	ENXIO	No such device/addr	32	EPIPE	Broken pipe
7	E2BIG	Arg list too long			. . .

The external variable errno is set to one of these error numbers after a system or library call failure, but it is not cleared after a successful call. This variable is available for your program to examine. The system/library call

perror(const char *s)

can be used to display the standard error message. The call perror(str) outputs to standard error:

1. The argument string str
2. The colon (´: ´) character
3. The standard error message associated with the current value of errno
4. A newline (´\n ´) character

The string argument given to perror is usually argv[0] or that plus the function name detecting the error.

Sometimes it is desirable to display a variant of the standard error message. For this purpose, the error messages can be retrieved through the standard library function

char *strerrpr(int n) /* obtain error message string */

which returns a pointer to the error string associated with error n. Also, there are error and end-of-file indicators associated with each I/O stream. Standard I/O library functions set these indicators when error or end-of-file occurs. These status indicators can be tested or set explicitly in your program with the library functions

int ferror(FILE *s) returns true (non-zero) if error indicator is set
int feof(FILE *s) returns true if eof indicator is set
void clearerr(FILE *s) clears eof and error indicators

Error Indications from Mathematical Functions

The variable errno is also used by the standard mathematical functions to indicate domain and range errors. A domain error occurs if a function is passed an argument whose value is outside the valid interval for the particular function. For example, only positive arguments are valid for the log function. A range error occurs when the computed result is so large or small that it cannot be represented as a double.

When a domain error happens, errno is set to EDOM, a symbolic constant defined in <errno.h>, and the returned value is implementation dependent. On the other hand, when a range error takes place, errno is set to ERANGE, and either zero (underflow) or HUGE_VAL (overflow) is returned.

Error Recovery

A run-time error can be treated in one of three ways:

1. Exiting—Display an appropriate error message, and terminate the execution of the program.
2. Returning—Return to the calling function with a well-defined error value.

3. Recovery—Transfer control to a saved state of the program in order to continue execution. ‘

The first two methods are well understood. The third, error recovery, is typified by such programs as vi, which returns to its top level when errors occur. Such transfer of control is usually from a point in one function to a point much earlier in the program in a different function. Such non-local control transfer cannot be achieved with a goto statement which only works inside a function. The two standard library routines setjmp and longjmp are provided for non-local jumps. To use these routines, the header file setjmp.h must be included.

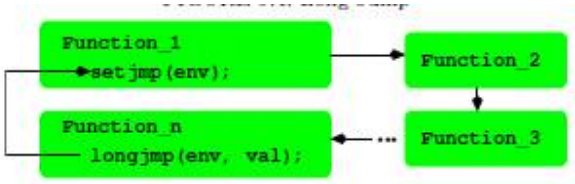
#include <setjmp.h>

The routine setjmp is declared as

int setjmp(jmp buf env) /* set up longjmp position */

which,when called, saves key data defining the current program state in the buffer env for possible later use by longjmp. The value returned by the initial call to setjmp is 0. The routine longjmp uses the saved env to throw control flow back to the setjmp statement.

void longjmp(jmp buf env, int val)



When called with a saved env and an integer val (must be nonzero), longjmp will restore the saved state env and cause execution to resume as if the original setjmp call has just returned the value val. For this backtracking to happen correctly, longjmp must be called from a function in a sequence of nested function calls leading from the function that invoked setjmp. In other words, setjmp establishes env as a non-local goto label, and longjmp is used to transfer control back to the point marked by env.

After the longjmp operation, all accessible global and local data have values as of the time when longjmp was called. The ANSI standard states that data values are not saved by the setjmp call.

Because of the way it works, setjmp can either stand alone or occur in the test condition part of if, switch, or while, and so on. The following is a simple example that shows how to use setjmp and longjmp.

```
#include <stdio.h>
#include <errno.h>
#include <setjmp.h>
jmp_buf env;
```

```
void recover(int n)
{ /* adjust values of variables if needed */
 longjmp(env, n);
}
  void func_2(int j)
  { /* normal processing */
recover(j);
}
void func_1(int i)
{ /* normal processing */
func_2( i * 2);
  }
 int main()
{ /* initialize and set up things here */
 /* then call setjmp */
 int err=0;
 if ( (err = setjmp(env)) != 0)
 { /* return spot for longjmp */
 /* put any adjustments after longjmp here */
  printf("Called longjmp\n");
  printf("Error No is %d\n", err);
 return err;
 }
 /* proceed with normal processing */
 printf("After initial setjmp()\n");
  printf("Calling func_1\n");
  func_1(19);
  }
```

In this example, the function main sets up the eventual longjmp called by the function recover. Note that recover never returns. It is possible to mark several places env1, env2,... with setjmp and use longjmp to transfer control to one of these marked places.

In addition to error recovery, a non-local jump can also be used to return a value directly from a deeply nested function call. This can be more efficient than a sequence of returns by all the intermediate functions. However, nonlocal control transfers tend to complicate program structure and should be used only sparingly.

Debugging with GDB

While the C compiler identifies problems at the syntax level, you still need a good tool for debugging at run time. GDB, the GNU debugger, is a

convenient utility for source-level debugging and controlled execution of programs. Your Linux distribution will usually have it installed. The command is gdb.

GDB can be used to debug programs written in many source languages such as C, C++, and f90, provided that the object files have been compiled to contain the appropriate symbol information for use by gdb. This means that you use the -g or better the -ggdb option of gcc.

Insight (sourceware.org/insight/) is a graphical user interface (GUI) front end for GDB. You can download and install it on your Linux if you prefer a window-menu–oriented environment for using gdb.

Other common debuggers include dbx and sdb. These are generally not as easy to use as gdb. We will describe how to use gdb to debug C programs. Once learned, gdb should be used as a routine tool for debugging programs. It is much more efficient than inserting fprintf lines in the source code. The tool can be used in the same way for many other programming languages.

Interactive Debugging

GDB provides an interactive debugging environment and correlates run-time activities to statements in the program source code. This is why it is called a source-level debugger. Debugging is performed by running the target program under the control of the gdb tool. The main features of gdb are listed below.

1. Source-level tracing—when a part of a program is traced, useful information will be displayed whenever that part is executed. If you trace a function, the name of the calling function, the value of the arguments passed, and the return value will be displayed each time the traced function is called. You can also trace specific lines of code and even individual variables. In the latter case, you'll be notified every time the variable value changes. ‘

2. Placing source-level breakpoints—a breakpoint in a program causes execution to suspend when that point is reached. At the breakpoint you can interact with gbx and use its full set of commands to investigate the situation before resuming execution.

3. Single source line stepping—when you are examining a section of code closely, you can have execution proceed one source line at a time. (Note that one line may consist of several machine instructions.)

4. Displaying source code—you can ask gbx to display any part of the program source from any file.

5. Examining values—Values, declarations, and other attributes of identifiers can also be displayed.

6. Object-level debugging—Machine instruction-level execution control and displaying of memory contents or register values are also provided.

To debug a C program using gdb, make sure each object file has been compiled and the final executable has been produced with gcc -ggdb. One simple way to achieve this is to compile all source code (.c) files at once using the gcc -ggdb source files command. This results in an executable a.out file suitable to run under the control of gdb. Thus, to use gdb on lowercase.c, you must first prepare it by gcc -g lowercase.c -o lowercase Then, to invoke gdb, you simply type gdb lowercase to debug the named executable file. If no file is given, a.out is assumed. When you see the prompt (gdb)the debugger is ready for an interactive session. When you are finished simply type the gdb command quit to exit from gdb.

A typical debugging session should follow these steps:

1. Invoke gdb on an executable file compiled with the -ggdb option.
2. Put in breakpoints.
3. Run the program under gdb.
4. Examine debugging output, and display program values at breakpoints.
5. Install new breakpoints to zero in on a bug, deleting old breakpoints as appropriate.
6. Resume or restart execution.
7. Repeat steps 4-7 until satisfied.

Having an idea of what gdb can do, we are now ready to look at the actual commands provided by gdb.

Basic gdb Commands

As a debugging tool, gdb provides a rich set of commands. The most commonly used commands are presented in this section. These should be sufficient for all but the most obscure bugs. The complete set of commands are listed in the gdb manual page.]

To begin execution of the target program within gdb, use

(gdb) run [args] [< file1] [> file2] (start execution in gdb)

where args are any command-line arguments needed by the binary file. It is also permitted to use > and < for I/O redirection. If lowercase is being debugged, then (gdb) run < input file > output file makes sense.

However, before running the program, you may wish to put in breakpoints first.

Command	Action
break *line*	Stops before execution of *line*
break *function*	Stops before call to *function*
break **address*	Stops at the *address*
display *expr*	Displays the C expression at a break

The break command can be abbreviated to br. Lines are specified by line numbers which can be displayed by these commands.

list	displays the next 10 lines.
list line1,line2	displays the range of lines.
list function	displays a few lines before and after function

When program execution under gdb reaches a breakpoint, the execution is stopped, and you get a (gdb) prompt so you can decide what to do and what values to examine. Commands useful at a breakpoint are in next table, where the command bt is short for backtrace which is the same as the command where. After reaching a breakpoint you may also single step source lines with step (execute the next source line) and next (execute up to the next source line). The difference between step and next is that if the line contains a call to a function, step will stop at the beginning of that function block but next will not.

As debugging progresses, breakpoints are put in and taken out in an attempt to localize the bug. Commands to put in breakpoints have been given. To disable or remove breakpoints, use

Command	Action
bt	Displays all call stack frames
bt *n*	Displays *n* innermost frames
bt `-`*n*	Displays *n* outermost frames
bt `full`	Displays all frames and local variable values
print *expr*	Displays the expression *expr*
whatis *name*	Displays the type of *name*
cont	Continues execution
kill	Aborts execution

disable number ... (disables the given breakpoints)

enable number ... (enables disabled breakpoints)

delete number ... (removes the given breakpoints)

Each breakpoint is identified by a sequence number.A sequence number is displayed by gdb after each break command. If you do not remember the numbers, enter info breakpoints to display all currently existing breakpoints.

If you use a set of gdb commands repeatedly, consider putting them in a file, say, mycmds, and run gdb this way

gdb -x mycmds a.out

A Sample Debugging Session with gdb

Let's show a complete debugging session using the source code low.c which is a version of lowercase.c that uses the Linux I/O system calls read and write to perform I/O.

```
#include <unistd.h>  '
#include <stdlib.h>
```

```
#include <stdio.h>
#include <ctype.h>
#define MYBUFSIZ 1024
int main(int argc, char* argv[ ])
{ char buffer[MYBUFSIZ];
void lower(char*, int);
int nc; /* number of characters */
while ((nc = read(STDIN_FILENO, buffer, MYBUFSIZ)) >
0)
{ lower(buffer,nc);
nc = write(STDOUT_FILENO, buffer, nc);
if (nc == -1) break;
}
if (nc == -1) /* read or write failed */
{ perror(argv[0]);
 exit(EXIT_FAILURE);
 }
 return EXIT_SUCCESS; /* normal termination */
}
  void lower(char *buf, int length)
 { while (length-- > 0)
{ if ( isupper( *buf ) )   *buf = tolower( *buf );
buf++;
 }
 }
```

We now show how gdb is used to control the execution of this program. User input is shown after the prompt (gdb). Output from gdb is indented.

We first compile lowercase.c for debugging and invoke gdb.

```
gcc -ggdb low.c -o low
gdb low
```

Now we can interact with gdb.

```
(gdb) list 10
int main(int argc, char* argv[])
{ char buffer[MYBUFSIZ];
void lower(char*, int);
int nc; /* number of characters */
while ((nc = read(0, buffer, MYBUFSIZ)) > 0)
{ lower(buffer,nc);
 nc = write(1, buffer, nc);
 if (nc == -1) break;
```

```
}
(gdb) br 10 (line containing system call read)
Breakpoint 1 at 0x400660: file low.c, line 10.
(gdb) br 12 (line containing system call write)
Breakpoint 2 at 0x400671: file low.c, line 12.
(gdb> br lower (function lower)
Breakpoint 3 at 0x4006ec: file low.c, line 23.
(gdb) run < file1 > file2 (run program)
Starting program: /home/pwang/ex/bug < file1 > file2
Breakpoint 1, main (argc=1, argv=0x7fff0f4ecfa8) at low.c:10
        10 while ((nc = read(0, buffer, MYBUFSIZ)) > 0)
        (gdb) whatis nc
        type = int
        (gdb) cont
        Continuing.
Breakpoint 3, lower (buf=0x7fff0f4ecab0 "It Is Time for  All Good
Men\n7", length=28) at low.c:23
        23 { while (length-- > 0)
         (gdb) bt
        #0 lower (buf=0x7fff0f4ecab0 "It Is Time for All
        Good Men\n7", length=28) at low.c:23
         #1  0x0000000000400671  in  main  (argc=1,
        argv=0x7fff0f4ecfa8)
         at low.c:11
         (gdb) whatis length
         type = int
        (gdb) cont
        Continuing.
Breakpoint 2, main (argc=1, argv=0x7fff0f4ecfa8) at low.c:12
        12 nc = write(1, buffer, nc);  \
         (gdb) bt
        #0 main (argc=1, argv=0x7fff0f4ecfa8) at low.c:12  \
         (gdb) cont
        Continuing.
         Program exited normally.
         (gdb) quit
```

GDB offers many commands and ways to debug.When in gdb, you can use the help command to obtain brief descriptions on commands. You can also look for gdb commands matching a regular expression with apropos command inside gdb. For example, you can type

(gdb) help break (displays info on break command)

(gdb) help (explains how to use help)

The GUI provided by insight can improve the debugging experience. For one thing, you don't need to memorize the commands because all the available controls at any given time are clearly displayed by the insight window.

Examining Core Dumps

In our preceding example (low.c), there were no errors. When your executable program encounters an error, a core dump file is usually produced. This file, named core.pid, is a copy of the memory image of your running program, with the process id pid, taken right after the error. Examining the core dump is like investigating the scene of a crime; the clues are all there if you can figure out what they mean. A core dump is also produced if a process receives certain signals. For example, you can cause a core dump by hitting the quit character (ctrl+\) on the keyboard.

The creation of a core file may also be controlled by limitations set in your Shell. Typing the Bash command

ulimit -c

will display any limits set for core dumps. A core dump bigger than the limit set will not be produced. In particular,

ulimit -c 0

prevents core dumps all together. To remove any limitation on core dumps use ulimit -c unlimited

You can use gdb to debug an executable with the aid of a core dump by simply giving the core file as an argument.

gdb executable corefile

Information provided by the given corefile is read in for you to examine. The executable that produced the corefile need not have been compiled with the -ggdb flag as long as the executable file passed to gdb was compiled with the flag.

Among other things, two pieces of important information are preserved in a core dump: the last line executed and the function call stack at the time of core dump. As it starts, gdb displays the call stack at the point of the core dump.

Summary

The C language is native to Linux and is used to write both application and system programs. Most Linux systems support C with the GCC compiler and the POSIX run-time libraries glibc from GNU.

The gcc compiler goes through five distinct phases to compile a program: preprocessing, compiling, optimizing (optional), assembly, and linking/

loading. GCC calls the preprocessor (cpp), the assembler (as), and the linker/loader (collect2/ld) at different phases and generates the final executable.

The Standard C Library is an ISO C99 API for headers and library routines.The GNU glibc contains Standard C Library implementations and other POSIX-compliant libraries. In addition, Linux provides many other useful libraries relating to networking, X Windows, etc.

A library is a type of archive file created and maintained using the ar command. You can create and maintain your own libraries with ar.

Standard header files provide access to system and library calls. Including the correct header files is important for C programs. Library functions, documented in section 3 of the Linux man pages, make application C programs easier to port to different platforms, whereas system calls, documented in section 2 of the man pages, access the Linux kernel directly.

Linux has well-established conventions for command-line arguments and for the reporting and handling of errors from system and library calls.The gdb debugger is a powerful tool for interactive run-time, source-level debugging and for analysis of a core dump. The insight tool provides a nice GUI for using gdb.

Review & Self Assessment Question:

Q1-What is Command Line Argument?

Q2-What is GCC Compiler?

Q3- Describe the term “The C Preprocessor”?

Q4-Explain Linking / Loading ?

Q5-Write the steps for Error Recovery ?

Further Readings

Linux Operating System Richard Petersen

Linux Operating System Paul S. Wang

Linux Operating System by David Maxwell and Andrew Bedford

Linux Operating System by Richard Blum and Christine Bresnahan

Linux Operating System by Bhatt P.C.P

UNIT: 9- I/O AND PROCESS CONTROL SYSTEM CALLS

Contents

Introduction

An operating system (OS) provides many tools and facilities to make a computer usable. However, the most basic and fundamental set of services is the system calls, specific routines in the operating system kernel that are directly accessible to application programs. There are over 300 system calls in Linux with a kernel-defined number starting from 1. Each system call also has a meaningful name and a symbolic constant in the form SYS_name for its number. With a few exceptions, a system call name corresponds to the routine sys_name in the Linux kernel source code.

A program under execution is called a process. When a process makes a system call at run time, a software-generated interrupt, often known as an operating system trap, triggers the process to switch from user mode to kernel mode and to transfer control to the entry point of the target kernel routine corresponding to the particular system call. A process running in kernel mode can execute instructions that are not available in user mode. Upon system call completion, the process switches back to user mode.

Higher level system facilities are built by writing library programs that use the system calls. Because Linux is implemented in C, its system calls are specified in C syntax and directly called from C programs.

Important Linux system calls are described here. These allow you to perform low-level input/output (I/O), manipulate files and directories, create and control multiple concurrent processes, and manage interrupts.

Examples show how system calls are used and how to combine different system calls to achieve specific goals.

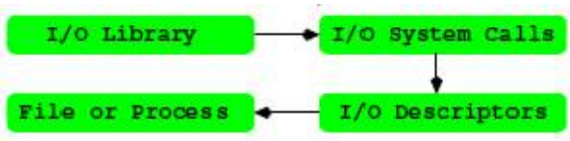
Just like library functions, a system call may need one or more associated header files. These header files are clearly indicated with each call described.

The set of system calls and their organization form the C-language interface to the Linux kernel, and this interface is nearly uniform across all major Linux distributions. The reason is because Linux systems closely follow POSIX (Portable Operating System Interface), an open operating system interface standard accepted worldwide. POSIX is produced by IEEE (Institute of Electrical and Electronics Engineers) and recognized by ISO (International Organization for Standardization) and ANSI (American National Standards Intitute). By following POSIX, software becomes easily portable to any POSIX-compliant OS.

Documentation for any system call name can be found with man 2 name in section 2 of the man pages.

System-Level I/O

High-level I/O routines such as putc and fopen, which are provided in the Standard C Library (Chapter 9), are adequate for most I/O needs in C programs. These library functions are built on top of low-level calls provided by the operating system. In Linux, the I/O stream of C is built on top of the I/O descriptor mechanism supported by system calls.



Getting to know the low-level I/O facilities will not only provide insight on how the library functions work, but will also allow you to use I/O in ways not supported by the Standard C Library.

Linux features a uniform interface for I/O to files and devices, such as a terminal window or an optical drive, by representing I/O hardware as special files. We shall discuss I/O to files, understanding they apply also to devices, which are nothing but special files. In addition to files, Linux supports I/O between processes (concurrently running programs) through abstract structures known as pipes and sockets. Although files, pipes, and sockets are different I/O objects, they are supported by many of the same low-level I/O calls explained here.

I/O Descriptors

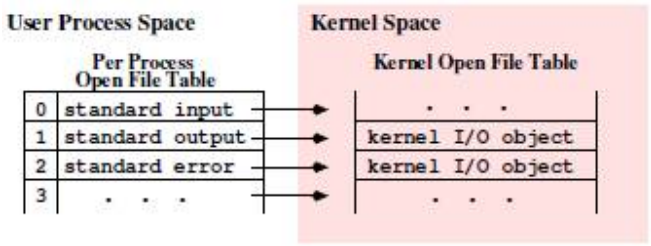
Before file I/O can take place, a program must first indicate its intention to Linux. This is done by the open system call declared as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *filename, int access [, mode_t mode])
```

Arguments to open are

filename character string for the pathname to the file
access an integer code for the intended access
mode the protection mode for creating a new file

The call opens filename, for reading and/or writing, as specified by access and returns an integer descriptor for that file. The filename can be given in any of the three valid forms: full pathname, relative pathname, or simple filename.The open command is also used to create a new file with the given name.Subsequent I/O operations will refer to this descriptor rather than to the filename.Other system calls return descriptors to I/O objects such as pipes and sockets.A descriptor is actually an index to a per-process open file table which contains necessary information for all open files and I/O objects of the process. The open call returns the lowest index to a currently unused table entry. Each table entry leads, in turn, to a kernel open file table entry. All processes share the same kernel open file table and it is possible for file descriptors from different processes to refer to the same kernel table entry.



For each process, three file descriptors, STDIN_FILENO(0), STDOUT_FILENO (1), and STDERR_FILENO (2), are automatically opened initially, allowing ready access to standard I/O. The access code is formed by the logical or (|) of header-supplied single-bit values including

O_RDONLY	to open file for reading only
O_WRONLY	to open file for writing only
O_RDWR	to open file for reading and writing
O_NDELAY	to prevent possible blocking
O_APPEND	to open file for appending
O_CREAT	to create file if it does not exist

O_TRUNC	to truncate size to 0
O_EXCL	to produce an error if the
O_CREAT	bit is on and file exists

Opening a file with O_APPEND instructs each write on the file to be appended to the end. If O_TRUNC is specified and the file exists, the file is truncated to length zero. If access is

(O_EXCL | O_CREAT)

and if the file already exists, open returns an error. The purpose is to avoid destroying an existing file.

The third and optional argument to open is a file creation mode in case the O_CREAT bit is on. The mode is a bit pattern (of type mode_t from <sys/types.h> with symbolic values from <sys/stat.h>) explained in detail in next Section , where the creat system call is described.

If the open call fails, a -1 is returned; otherwise, a descriptor is returned. A process may have no more than a maximum number of descriptors open simultaneously. This limit is large enough in Linux to be of no practical concern.

The following example (Ex: ex10/open.c) shows a typical usage of the open system call. The third argument to open is unused because it is not needed for the read-only (O_RDONLY) operation. In this case, any integer can be used as the third argument.

```
/******* open.c *******/
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
int main(int argc, char *argv[])
{ int fd; /* file descriptor */
/* open argv[1] for reading */
if ((fd = open(argv[1], O_RDONLY, 0)) == -1)
{ fprintf(stderr,"%s: cannot open %s\n", argv[0], argv[1]);
perror("open system call"); exit(EXIT_FAILURE); }
/* other code */
}
```

When a system or library call fails, you can use the code

perror (const char* msg) (displays system error)

to display the given message msg followed by a standard error message associated with the error.

When a descriptor fd is no longer needed in a program, it can be deleted from the per-process open file table using the call int close(int fd) (closes

descriptor) Otherwise, all open file descriptors will be closed when the program terminates.

Reading and Writing I/O Descriptors

Reading and writing are normally sequential. For each open descriptor, there is a current position which points to the next byte to be read or written. After k bytes are read or written, the current position, if movable, is advanced by k bytes. Whether the current position is movable depends on the I/O object. For example, it is movable for an actual file but not for stdin when connected to the keyboard.

The system calls read and write are declared as

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize t read(int fd, void *buffer, size t k) (reads input from fd)
ssize t write(int fd, void *buffer, size t k) (writes output to fd)
```

where fd is a descriptor to read from or write to, buffer points to an array to receive or supply the bytes, and k is the number of bytes to be read in or written out. Obviously, k must not exceed the length of buffer. Read will attempt to read k bytes from the I/O object represented by fd. It returns the number of bytes actually read and deposited in the buffer. The type size_t is usually unsigned int (non-negative) and ssize_t is usually int (can be negative). If read returns less than k bytes, it does not necessarily mean that end-of-file has been reached, but if zero is returned, then the end of the file has been reached.

The write call outputs k bytes from the buffer to fd and returns the actual number of bytes written out. Both read and write return a -1 if they fail. As an example, we can write a readline function with low-level read (Ex: ex10/readline.c).

```
int readline(char s[], int size)
{ char *tmp = s;
/* read one character at a time */
  while (0 < --size && read(0, tmp, 1) != 0
 && *tmp++ != '\n'); /* empty loop body */
*tmp = '\0'; /* string terminator */
return tmp-s; /* number of characters before terminator */
}
```

The while loop control is intricate and warrants careful study. The size argument is the capacity of the array s. The function returns the number of characters read, not counting the string terminator.

Moving the Current Position

When reading or writing an I/O object that is an actual file, the object can be viewed as a sequence of bytes. The current position is moved by the read and write operations in a sequential manner. As an alternative to this, the system call lseek provides a way to move the current position to any location and therefore allows random access to bytes of the file. The standard library function fseek is built on top of lseek. The call

```
#include <sys/types.h>
#include <unistd.h>
off t lseek(int fd,
        off t offset, int origin) (moves read/write position)
```

moves the current position associated with the descriptor fd to a byte position defined by (origin + offset). Table 10.1 shows the three possible origins.

Origin	Position
SEEK_SET	The beginning of a file
SEEK_CUR	The current position
SEEK_END	The end of a file

The offset can be positive or negative. The call lseek returns the current position as an integer position measured from the beginning of the file. It returns -1 upon failure.

Call	Meaning
lseek(fd, (off_t)0, SEEK_SET)	Puts current pos at first byte.
lseek(fd, (off_t)0, SEEK_END)	Moves current pos to end of the file.
lseek(fd, (off_t)-1, SEEK_END)	Puts current pos at last byte.
lseek(fd, (off_t)-10, SEEK_CUR)	Backs up current pos by 10 bytes.

It is possible to lseek beyond the end of file and then write. This creates a hole in the file which does not occupy file space. Reading a byte in such a hole returns zero.

In some applications, holes are left in the file on purpose to allow easy insertion of additional data later. It is an error to lseek a non-movable descriptor such as the STDIN_FILENO. See the example code package (Ex: ex10/lowerseek.c) for an implementation of the lowercase program using lseek and O_RDWR.

Operations on Files

System calls are provided for creating and deleting files, accessing file status information, obtaining and modifying protection modes, and other attributes of a file. These will be described in the following subsections.

Creating and Deleting a File

For creating a new file, the open system call explained in the previous section can be used. Alternatively, the system call

```
int creat(char *filename, int mode) (creates a new file)
```

can also be used. If the named file already exists, it is truncated to zero length, and ready to be rewritten. If it does not exist, then a new directory entry is made for it, and creat returns a file descriptor for writing this new file. It is equivalent to

open(filename, (O CREAT|O WRONLY|O TRUNC), mode)

The lower 9 bits of mode (for access protection) are modified by the file creation mask umask of the process using the formula

(~umask) & mode

The mode is the logical or of any of the basic modes shown in Table.

The initial umask value of a process is inherited from the parent process of a running program.We have seen how to set umask using the Bash umask command.The default umask is usually 0022, which clears the write permission bits for group and other.

A program can set umask with the system call

```
#include <sys/types.h>
#include <sys/stat.h>
  mode_t umask(mode_t mask);
```

The returned value is the old umask.

For example,

```
umask(0077);
```

will force file modes for newly created files to allow file access only for the owner. The value of umask is inherited by child processes. After a file is created, it can be read/written with the read, write calls.

Linking and Renaming Files

For an existing file, alternative names can also be given. The call link

```
#include <unistd.h>
int link(const char *file, const char *name) (a hard link)
 int symlink(const char *file, const char *name) (a symbolic link)
```

establishes another name (directory entry) for the existing file.The new name is a hard link and can be anywhere within the same file system. To remove a link, the call

int unlink(const char *name) (deletes file link)

is used. When the link removed is the last directory entry pointing to this file, then the file is deleted.

Use a symbolic link (the symlink system call) for a directory or a file in a different filesystem.

At the Shell level, renaming a file is done with the mv command. At the system call level, use

#include <stdio.h>

int rename(const char* old name, const char* new name)

Both filenames must be within the same filesystem. When renaming a directory, the new name must not be under old name.

Accessing File Status

```
struct stat
{ dev_t     st_dev;     /* ID containing file      */
  ino_t     st_ino;     /* i-number                */
  mode_t    st_mode;    /* file mode               */
  nlink_t   st_nlink;   /* number of hard links    */
  uid_t     st_uid;     /* user ID of owner        */
  gid_t     st_gid;     /* group ID of owner       */
  dev_t     st_rdev;    /* special file ID         */
  off_t     st_size;    /* total bytes             */
  blksize_t st_blksize; /* filesystem  blocksize   */
  blkcnt_t  st_blocks;  /* No. of blocks allocated */
  time_t    st_atime;   /* last access time        */
  time_t    st_mtime;   /* last modification time  */
  time_t    st_ctime;   /* last status change time */
};
```

For each file, Linux maintains a set of status information such as file type, protection modes, time when last modified and so on. The status information is kept in the i-node of a file. To access file status information from a C program, the following system calls can be used.

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

int stat(const char *file, struct stat *buf) (of file)

int fstat(int fd, struct stat *buf) (of descriptor fd)

int lstat(const char *link, struct stat *buf) (of the symbolic link)

Note that fstat is the same as stat, except it takes a file descriptor that has been opened already. This parallel exists for many other system calls. The lstat is the same as stat, except the former does not follow symbolic links. The status information for the given file is retrieved and placed in buf. Accessing status information does not require read, write, or execute permission for the file, but all directories listed in the pathname leading to the file (for stat) must be reachable.

The stat structure has many members. The next Tables list the symbolic constants for interpreting the value of the stat member st_mode.

Octal Bit Pattern	Symbol	Meaning
00400, 00200, 00100	S_IRUSR, S_IWUSR, S_IXUSR	r, w, or x by u
00040, 00020, 00010	S_IRGRP, S_IWGRP, S_IXGRP	r, w, or x by g
00004, 00002, 00001	S_IROTH, S_IWOTH, S_IXOTH	r, w, or x by o
00700, 00070, 00007	S_IRWXU, S_IRWXG, S_IRWXO	rwx by u, g, or o

There are three timestamps kept for each file:

• st atime (last access time)—The time when file was last read or modified. It is affected by the system calls mknod, utimes, read, and write. For reasons of efficiency, st_atime is not set when a directory is searched.

• st mtime (last modify time)—The time when file was last modified. It is not affected by changes of owner, group, link count, or mode. It is changed by : mknod, utimes, and write.

• st ctime (last status change time)—The time when file status was last changed. It is set both by writing the file and by changing the information contained in the i-node. It is affected by chmod, chown, link, mknod, unlink, utimes, and write.

The timestamps are stored as integers, and a larger integer value represents a more recent time. Usually, Linux uses GMT (Greenwich Mean Time). The integer timestamps, however, represent the number of seconds since a fixed point in the past, known as the POSIX epoch which is UTC 00:00:00, January 1, 1970. The library routine ctime converts such an integer into an ASCII string representing date and time.

The mask S_IFMT is useful for determining the file type. For example, if ((buf.st_mode & S_IFMT) == S_IFDIR)

Symbol	Bit Pattern	Meaning
S_IFMT	0170000	File type bit mask
S_IFSOCK	0140000	Socket
S_IFLNK	0120000	Symbolic link
S_IFREG	0100000	Regular file
S_IFBLK	0060000	Block device
S_IFDIR	0040000	Directory
S_IFCHR	0020000	Character device
S_IFIFO	0010000	FIFO
S_ISUID	0004000	Set-UID bit
S_ISGID	0002000	Set-group-ID bit
S_ISVTX	0001000	Sticky bit

determines whether the file is a directory.

Determining Allowable File Access

It is possible to determine whether an intended read, write or execute access to a file is permissible before initiating such an access. The access system call is defined as

```
#include <unistd.h>
int access(const char *file, int a mode) (checks access to file)
```

The access call checks the permission bits of file to see if the intended access given by a mode is allowable. The intended access mode is a logical or of the bits R_OK, W_OK, and X_OK defined by

```
#define R_OK 4 /* test for read permission */
```

```
#define W_OK 2 /* test for write permission */
#define X_OK 1 /* test for execute (search) permission */
#define F_OK 0 /* test for presence of file */
```

If the specified access is allowable, the call returns 0; otherwise, it returns -1. Specifying a mode as F_OK tests whether the directories leading to the file can be searched and whether the file exists.

Operations on Directories

Creating and Removing a Directory

In addition to files, it is also possible to establish and remove directories with Linux system calls. The system call mkdir creates a new directory.

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *dir, mode t mode) (makes a new folder)
```

It creates a new directory with the name dir. The mode works the same way as in the open system call. The new directory's owner ID is set to the effective user ID of the process. If the parent directory containing dir has the set-group-ID bit on, or if the filesystem is mounted with BSD (Berkeley UNIX) group semantics, the new directory dir will inherit the group ID from its parent folder. Otherwise, it will get the effective group ID of the process.

```
The system call rmdir
#include <unistd.h>
int rmdir(const char *dir) (removes a folder)
```

remove the given directory dir. The directory must be empty (having no entries other than. and ..). For both mkdir and rmdir, a 0 returned value indicates success, and a -1 indicates an error. The content of a directory consists mainly of file names (strings) and i-node numbers (i-number). The length limit of a simple fime name depends on the filesystem. Typically, simple file names are limited to a length of 255 characters.

The system call getdents can be used to read the contents of a directory file into a character array in a system-independent format. However, a more convenient way to access directory information is to use the directory library functions discussed in the next section.

Directory Access

In the Linux file system, a directory contains the names and i-numbers of files stored in it. Library functions are available for accessing directories. To use any of them, be sure to include these header files:

```
#include <sys/types.h>
#include <dirent.h>
```

To open a directory, use either

DIR *opendir(const char *dir) (opens directory stream)

or

DIR *fdopendir(int fd) (opens directory stream)

to obtain a directory stream pointer (DIR *) for use in subsequent operations.If the named directory cannot be accessed, or if there is not enough memory to hold the contents of the directory, a NULL (invalid pointer) is returned.

Once a directory stream is opened, the library function readdir is used to sequentially access its entries. The function

```
#include <sys/types.h>
#include <dirent.h>
```

struct dirent *readdir(DIR *dp) (returns next dir entry from dp)

returns a pointer to the next directory entry. The pointer value becomes NULL on error or reaching the end of the directory.

The directory entry structure struct dirent records information for any single file in a directory.

```
struct dirent
{ ino_t d_ino; /* i-node number of file */
  off_t d_off; /* offset to the next dirent */
  unsigned short d_reclen; /* length of this record */
 unsigned char d_type; /* file type */
  char d_name[256]; /* filename */
};
```

Each file in a filesystem also has a unique i-node number . The NAME_MAX constant, usually 255, gives the maxima length of a directory entry name. The data structure returned by readdir can be overwritten by a subsequent call to readdir.

The function

closedir(DIR *dp) (closes directory stream)

closes the directory stream dp and frees the structure associated with the DIR pointer.

To illustrate the use of these library functions, let’s look at a function searchdir (Figure 10.4) which searches dir for a given file and returns 1 or 0 depending on whether the file is found or not (Ex: ex10/searchdir.c). Note that the example uses knowledge of the dirent structure. Enumeration constants FOUND and NOT_FOUND are used. The for loop goes through each entry in dir to find file. Note the logical not (!) in front of strcmp.

Current Working Directory

The library routine

char *get current dir name(void); (obtains current directory)
returns the full pathname of the current working directory. The system call

int chdir(const char *dir) (changes directory)

is used to change the current working directory to the named directory. A value 0 is returned if chdir is successful; otherwise, a -1 is returned. Because the current directory is a per-process attribute, you will return to the original directory after the program exits.

An Example: ccp

It is perhaps appropriate to look at a complete example of a Linux command written in C. The command we shall discuss is ccp (conditional copy), which is used to copy files from one directory to another (Ex: ex10/ccp.c). A particular file is copied or not depending on whether updating is necessary.

The ccp command copies files from a source folder source to a destination folder dest. The usage is

ccp source dest [file . . .]

The named files or all files (but not directories) are copied from source to dest subject to the following conditions:

1. If the file is not in dest, copy the file.
2. If the file is already in dest but the file in source is more recent, copy the file.
3. If the file is already in dest and the file in source is not more recent, do not copy the file.

To check if a file is a directory, we call the isDir function (line 1). To compare the recency of two files (line 2), we use the function newer presented in next section.

```
/******** ccp : the conditional copy command ********/
 #include <sys/param.h>
 #include <stdio.h>
#include <stdlib.h>
 #include <dirent.h>
 #include <unistd.h>
 #include <string.h>
 #include <sys/stat.h>
 #include "newer.h"
  int isDir(const char *file)
 { struct stat stb;
 if (stat(file, &stb) < 0) /* result returned in stb */
 return -1; /* stat failed */
 return ((stb.st_mode & S_IFMT) == S_IFDIR);
```

```
}
void ccp(const char* name, const char* d1, const char* d2)
  { char f1[MAXPATHLEN+1], f2[MAXPATHLEN+1];
  strcpy(f1,d1); strcpy(f2,d2); strcat(f1,"/");
 strcat(f2,"/"); strcat(f1,name); strcat(f2,name);
 if ( isDir(f1)==0 ) /* (1) */
 if ( access(f2,F_OK) == -1 || newer(f1,f2) ) /* (2) */
  printf("copy(%s,%s)\n", f1, f2);
else
printf("no need to copy(%s,%s)\n", f1, f2);
}
  int main(int argc,char* argv[])
 { DIR *dirp1;
 struct dirent *dp;
if (argc < 3) /* need at least two args */
 {  fprintf(stderr, "%s: wrong number of arguments",
argv[0]);
 exit(EXIT_FAILURE);
 }
  else if (argc > 3) /* files specified */
 { int i;
 for (i = 3; i < argc; i++)
 ccp(argv[i],argv[1],argv[2]) ; /* (3) */
 return EXIT_SUCCESS;
}
 /* now exactly two args */
  if ((dirp1 = opendir(argv[1])) == NULL)
 { fprintf(stderr, "%s: can not open %s", argv[0], argv[1]);
  exit(EXIT_FAILURE);   }
for (dp = readdir(dirp1); dp != NULL;
 dp = readdir(dirp1)) /* (4) */
 if (strncmp(dp->d_name,".", 1))
  ccp(dp->d_name,argv[1],argv[2]);
 return EXIT_SUCCESS;
 }
```

If files are given on the command line, we call the function ccp on those files (line 3). Otherwise, we go through all files whose names do not begin with a period (line 4). To compile we use

```
gcc ccp.c newer.c -o ccp
```

Shell-Level Commands from C Programs

In the ccp.c example, we have not performed any actual file copying. We simply used printf to indicate the copying actions needed. To carry out the file copying, it is most convenient to invoke a Shell-level cp command from within a C program. Allowing execution of Shell-level commands from within C programs is very useful. With this ability, you can, for example, simply issue a cp command to copy a file from a C program rather than writing your own routines. The Linux library call system is used for this purpose.

```
#include <stdlib.h>
int system(const char *cmd str) /* issues Shell command */
```

The system call starts a new Sh process to execute the given string cmd_str. The Shell terminates after executing the given command and system returns. The returned value represents the exit status of the given command. Thus, to copy file1 to file2, you can use

```
char cmd_string[80];
sprintf(cmd_string, "cp %s %s\n", file1, file2);
system(cmd_string);
```

The string is, of course, interpreted by the Shell before the command is invoked.Any substitution and filename expansion will be done. Also, the Shell locates the executable file (for example, /bin/cp) on the command search path for you. Use the full pathname of the command if you do not wish to depend on the PATH setting. The system call waits until the command is finished before returning.

One shortcoming of the system function is that it does not allow you to receive the results produced by the command or to provide input to it. This is remedied by the library function popen.

Process Control

A key operating system kernel service is process control. A process is a program under execution, and in a multiprogramming system like Linux, there will be multiple processes running concurrently at any given time.

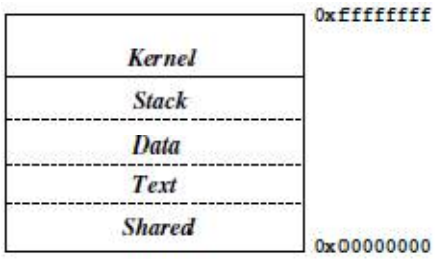
We will look at process address space, states, control structures, creation and termination, executable loading, and inter-process communication here and in later sections.

Virtual Address Space

When created, each individual process has, among other resources, memory space allocated for its exclusive use. This memory space is often referred to as the virtual address space (or simply address space) of a process. The address space consists of a kernel space which is the Linux kernel shared by all processes and a user space which is off limits to other processes.A process executing in user mode has no access to the kernel

space except through system calls provided by the kernel. Upon a system call, control is transferred to a kernel address through a special signal (Chapter 10, Section 10.16) and the process switches to kernel mode. While in kernel mode, the process has access to both user space and kernel space. The process switches back to user mode upon return of the system call.

The process user space is organized into shared, text, data, and stack regions.



• Stack—A last-in-first-out data structure used to manage function calls, returns, parameter passing, and returned values. The memory used for the stack will grow and shrink with the depth of nesting of function calls.

• Data—The values of variables, arrays, and structures. Objects allocated at compile time will occupy fixed memory locations in the data area. Room for dynamically allocated space (through malloc) is also included in the data area.

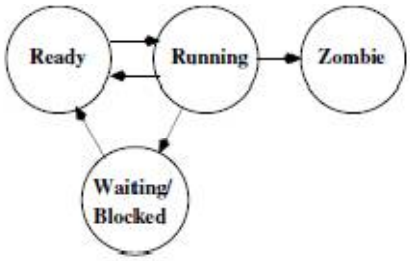
• Text—The machine instructions that represent the procedures or functions in the program. This part of a process will generally stay unchanged over the lifetime of the process.

• Shared—Code from libraries that is not duplicated when shared with other processes.

In addition to the address space, each process is also assigned system resources necessary for the kernel to manage the process.

Process Life Cycle

Each process is represented by an entry in the process table which is manipulated by the kernel to manage all processes. The kernel schedules the CPU (Central Processing Unit) and switches it from running one process to the next in rapid succession. Thus, the processes appear to make progress concurrently. On a computer with multiple CPUs, a number of processes can actually run simultaneously or in parallel. A process usually goes through a number of states before running to completion.



The process states are

• Running—The process is executing.

• Waiting/Blocked—A process in this state is waiting for an event to occur. Such an event could be an I/O completion by a peripheral device, the termination of another process, the availability of data or space in a buffer, the freeing of a system resource, and so on. When a running process has to wait for such an event, it is blocked and waiting to be unblocked so it can continue to execute. A process blocking creates an opportunity for a context switch, shifting the CPU to another process. Later, when the event a blocked process is waiting for occurs, it awakens and becomes ready to run.

• Ready—A process in this state is then scheduled for CPU service

• Zombie—After termination of execution, a process goes into the zombie state. The process no longer exists. The data structure left behind contains its exit status and any timing statistics collected. This is always the last state of a process.

A process may go through the intermediate states many times before it is finished.

From a programming point of view, a Linux process is the entity created by the fork system call. In the beginning, when Linux is booted there is only one process (process 0) which uses the fork system call to create process 1, known as the init process. The init process is the ancestor of all other processes, including your login Shell. Process 0 then becomes the virtual memory swapper.

The Process Table

A system-wide process table is maintained in the Linux kernel to control all processes. There is one table entry for each existing process. Each process entry contains all key information needed to manage the process, such as PID (a unique integer process ID), UID (real and effective owner and group ID's of user executing this process), process status, and generally information displayed by the ps command. Linux provides a directory under /proc/ for each existing process, making it easy to access information on individual processes from the Shell level.

The ps Command

You can also obtain various kinds of information on processes with the command ps (displays process status)

Because Linux is a multi-user system and because there are many system processes that perform various chores to keep Linux functioning, there are always multiple processes running at any given time. The ps command attempts to display a reasonable set of processes that are likely to be of interest to you, and you can give options to control what subset of processes are displayed.

The ps command displays information only for your processes. Give the option -a display all interesting processes. Also, ps displays in short form unless given the option -f to see a full-format listing. For example,

ps -af

displays, in full format, all interesting processes. Use the option -e (or -A) to display all current processes, including daemon processes (those without a control terminal such as the cron process). See the ps man page for quite a few other options.

Information provided for each process includes

PID—The process ID in integer form

PPID—The parent process ID in integer form

S—The single-letter state code from the ps man page

STIME or START—The process start time

TIME—CPU time (in seconds) used by the process

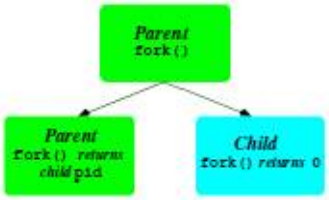
TT—Control terminal of the process

COMMAND—The user command which started this process

When you are looking for a particular process, the pipe

ps -e | grep string

can be handy.



Process Creation: fork

The fork system call is used inside a C program to create another process.

#include <sys/types.h>

#include <unistd.h>

pid t fork();

The process which calls fork is referred to as the parent process, and the newly created process is known as the child process. After the fork call, the child and the parent run concurrently.

The child process created is a copy of the parent process except for the following:

• The child process has a unique PID.

• The child process has a different PPID (PID of its parent).

The fork is called by the parent, but returns in both the parent and the child. In the parent, it returns the PID of the child process, whereas in the child, it returns 0. If fork fails, no child process is created, and it returns -1. Here is a template for using fork.

```
pit_t pid;
if ((pid = fork()) == 0)
 {
/* put code for child here */
 }
 if (pid < 0)
 {
 /* fork failed, put error handling here */
 }
/* fork successful, put remaining code for parent here */
```

The following simple program (Ex: ex10/simplefork.c) serves to illustrate process creation, concurrent execution, and the relationships between the child and the parent across the fork call.

```
/******** simplefork.c ********/
#include <sys/types.h>
 #include <unistd.h>
#include <stdlib.h>
 #include <stdio.h>
  int main()
 { pid_t child_id;
  child_id = fork();      /* process creation (1) */
  if ( child_id == 0 )     /* child code begin (2) */
{ printf("Child: My pid = %d and my parent pid = %d\n",
        getpid(), getppid());
        _exit(EXIT_SUCCESS); /* child terminates (3) */
  }
        /* child code end */
 if ( child_id < 0 ) /* remaining parent code */
```

```
{
fprintf(stderr, "fork failed\n");
exit(EXIT_FAILURE);
}
 printf("Parent: My pid = %d, spawned child pid = %d\n",
 getpid(), child_id);
return EXIT_SUCCESS;
}
```

After calling fork (line 1), you suddenly have two processes, the parent and the child, executing the same program starting at the point where fork returns.

The child and parent execute different code sections in our example because of the way the program is written. The child only executes the part under if (child_id==0) (line 2). At the end of the child code (line 3), it must terminate execution. Otherwise, the child would continue into the code meant only for the parent. The exit system call is slightly different from library function exit and is explained in Section 10.14. Note also that a process can use the system calls getpid() and getppid() to obtain the process ID of itself and its parent, respectively.

Program Execution: exec Routines

A process can load and execute another program by overlaying itself with an executable file. The target executable file is read in on top of the address space of the very process that is executing, overwriting it in memory, and execution continues at the entry point defined in the file. The result is that the process begins to execute a new program under the same execution environment as the old program, which is now replaced.

This program overlay can be initiated by any one of the exec library functions, including execl, execv, execve, and several others, each a variation of the basic execv library function.

```
#include <unistd.h>
extern char **environ;
int execv(const char *filename, char *const argv[]);
```

where filename is the full pathname of an executable file, and argv is the command-line arguments, with argv[0] being command name.

This execv call overlays the calling process with a new executable program. If execv returns, an error has occurred. In this case the value returned is -1. The argument argv is an array of character pointers to null-terminated character strings. These strings constitute the argument list to be made available to the new process. By convention, at least one argument must be present in this array, and the first element of this array

should be the name of the executed program (i.e., the last component of filename). To the calling program, a successful execv never returns.

Other exec functions may take different arguments but will work the same way as execv. To avoid confusion, we will refer to all of them as an exec call.

An exec call is often combined with fork to produce a new process which runs another program.

1. Process A (the parent process) calls fork to produce a child process B.
2. Process B immediately makes an exec call to run a new program.

An exec call transforms the calling process to run a new program. The new program is loaded from the given filename which must be an executable file. An executable file is either a binary a.out. or an executable text file containing commands for an interpreter. An executable text file begins with a line of the form

#! interpreter

When the named file is an executable text file, the system runs the specified interpreter, giving it the named file as the first argument followed by the rest of the original arguments. For example, a Bash script may begin with the line #!/bin/bash

and an Sh script with

#!/bin/sh

As for an executable binary, Linux has adopted the standard ELF (Executable and Linking Format) which basically provides better support for the linking and dynamical loading of shared libraries as compared to the old UNIX a.out format. The command

readelf -h a.out

displays the header section of the executable a.out. Do a

man 5 elf

to read more about the ELF file format.

- Process ID, parent process ID, and process group ID
- Process owner ID, unless for a set-userid program
- Access groups, unless for a set-groupid program
- Working directory and root directory
- Session ID and control terminal
- Resource usages
- Interval timers
- Resource limits
- File mode mask (umask)
- Signal mask
- Environment variable values

Furthermore, descriptors which are open in the calling process usually remain open in the new process. Ignored signals remain ignored across an exec, but signals that are caught are reset to their default values.

Synchronization of Parent and Child Processes

After creating a child process by fork, the parent process may run independently or elect to wait for the child process to terminate before proceeding further. The system call

```
#include <sys/types.h>
#include <sys/wait.h>
pid t wait(int *t status);
```

searches for a terminated child (in zombie state) of the calling process. It performs the following steps:

1. If there are no child processes, wait returns right away with the value -1 (an error).
2. If one or more child processes are in the zombie state (terminated) already, wait selects an arbitrary zombie child, frees its process table slot for reuse, stores its termination status in *t_status if t_status is not NULL, and returns its process ID.
3. Otherwise, wait sleeps until one of the child processes terminates and then goes to step 2.

When wait returns after the termination of a child, the variable (*t status) is set, and it contains information about how the process terminated (normal, error, signal, etc.) You can examine the value of *t_status with predefined macros such as

WIFEXITED(*t_status) (returns true if child exited normally) WEXITSTATUS(*t_status) (returns the exit status of child)

See man 2 wait for other macros and for additional forms of wait.

A parent process can control the execution of a child process much more closely by using the ptrace (process trace) system call.This system call is primarily used for interactive breakpoint debugging such as that supported by the gdb command.When the child process is traced by its parent, the waitpid system call is used, which returns when the specific child is stopped (suspended temporarily).

Process Termination

Every running program eventually comes to an end. A process may terminate execution in three different ways:

1. The program runs to completion and the function main returns.
2. The program calls the library routine exit or the system call exit.

3. The program encounters an execution error or receives an interrupt signal, causing its premature termination.

The argument to exit/exit is the process exit status and is part of the termination status of the process. Conventionally, a zero exit status indicates normal termination and non-zero indicates abnormal termination. The system call

void exit(int status)

terminates the calling process with the following consequences:

1. All of the open I/O descriptors in the process are now closed.

2. If the parent process of the terminating process is executing a wait, then it is notified of the termination and provided with the child termination status.

3. If the terminating process has child processes yet unfinished, the PPIDs of all existing children are set to 1 (the init process). Thus, the new orphan processes are adopted by the init process.

Most C programs call the library routine exit which performs clean-up actions on I/O buffers before calling exit. The exit system call is used by a child process to avoid possible interference with I/O buffers shared by parent and child processes.

The User Environment of a Process

The parameters argc and argv of a C program reference the explicit arguments given on the command line. Every time a process begins, another array of strings, representing the user environment, called the environment list, is also passed to the process. This provides another way through which to control the behavior of a process. If the function main is declared as

int main(int argc, char* argv[], char* arge[])

then arge receives additional values for the environment list which is always available for a process in the global array environ:

extern char **environ

Each environment string is in the form

name=value

Although direct access to environ is possible in a C program, it is simpler to access environment values in a C program with the library routine getenv:

#include <stdlib.h>

char* getenv(const char* name)

This routine searches the environment list for a string, of the form name=value, that matches the given name and returns a pointer to value or NULL if no match for name is found.

With getenv we can write a simple test program (Ex: ex10/envtest.c).

```
/******** envtest.c ********/
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char* argv[], char* arge[])
 { char *s;
 s = getenv("PATH");
 printf("PATH=%s\n", s);
return EXIT_SUCCESS;
}
```

You can set environment values at the Shell level.Environment variables and their values are contained in the environment list. Frequently used environment variables include PATH, HOME, TERM, USER, SHELL, DISPLAY, and so on .

In Bash, we can also pass additional environmental values to any single command by simply listing them before the command. For example,

```
gcc envtest.c -o envtest
foo=bar ... ./envtest
```

At the C level, the execl and execv library calls pass to the invoked program their current environment. The system call

```
#include <unistd.h>
int execve(const char *file, char *const argv[], char *const
envp[]);
```

can be used to pass an environment array envp containing additional environmental values to the new program (Ex: ex10/execve.c).

```
/* passing environment with execve */
#include <unistd.h>
#include <stdlib.h>
char* envp[3];
int main(int argc, char* argv[])
{ envp[0]="first=foo";
 envp[1]="second=bar";
envp[2]=NULL;
execve("target-program", argv, envp);
exit(EXIT_FAILURE); /* execve failed */
 }
```

Interrupts and Signals

Basic Concepts

We already know that a program executes as an independent process. Yet, events outside a process can affect its execution. The moment when such

an event would occur is not predictable. Thus, they are called asynchronous events. Examples of such events include I/O blocking, I/O ready, keyboard and mouse events, expiration of a time slice, as well as interrupts issued interactively by the user. Asynchronous events are treated in Linux using the signal mechanism. Linux sends a certain signal to a process to signify the occurrence of a particular event. After receiving a signal, a process will react to it in a well-defined manner. This action is referred to as the signal disposition. For example, the process may be terminated or suspended for later resumption.There is a system-defined default disposition associated with each signal.A process normally reacts to a signal by following the default action. However, a program also has the ability to redefine its disposition to any signal by specifying its own handling routine for the signal.

Symbol	Default action	Meaning
SIGHUP	Terminate	Hangup (for example, terminal window closed)
SIGINT	Terminate	Interrupt (for example, CTRL+C from keyboard)
SIGQUIT	Core dump	Quit (for example, CTRL+\ from keyboard)
SIGILL	Core dump	Illegal instruction
SIGTRAP	Core dump	Trace/breakpoint trap
SIGABRT	Terminate	Abort (**abort**())
SIGBUS	Core dump	Memory bus error
SIGFPE	Core dump	Floating point exception
SIGKILL	Terminate	Force terminate
SIGSEGV	Core dump	Invalid memory reference
SIGALRM	Terminate	Time signal (**alarm**())
SIGPROF	Terminate	Profiling timer alarm
SIGSYS	Core dump	Bad argument to system call
SIGCONT	Resume	Continue if stopped
SIGSTOP	Suspend	Suspends process
SIGTSTP	Suspend	Stop (for example, CTRL+Z) from terminal

There are many different signals. For instance, typing ctrl+\ on the keyboard usually generates a signal known as quit. Sending the quit signal to a process makes it terminate and produces a core image file for debugging.Each kind of signal has a unique integer number, a symbolic name, and a default action defined by Linux.A complete list of all signals can be found with man 7 signal.

Sending Signals

You may send signals to processes connected to your terminal window by typing certain control characters such as ctrl+\, ctrl+c, and ctrl+z typed at the Shell level. These signals and their effects are summarized below.

ctrl+c SIGINT terminates execution of foreground process

ctrl+\ SIGQUIT terminates foreground process and dumps core

ctrl+z SIGTSTP suspends foreground process for later resumption

In addition to these special characters, you can use the Shell-level command kill to send a specific signal to a given process. The general form of the kill command is

kill [−sig no] process

where process is a process number (or Shell jobid). The optional argument specifies a signal number sig no.If no signal is specified, SIGTERM is assumed which causes the target process to terminate.

In a C program, the standard library function

int raise(int sig_no) (sends sig_no to the process itself)

is used by a process to send the signal sig_no to itself, and the system call

int kill(pid_t pid, int sig_no) (sends sig_no to process pid)

is used to send a specified signal to a process identified by the given numerical pid.

Signal Delivery and Processing

When a signal is sent to a process, the signal is added to a set of signals pending delivery to that process. Signals are delivered to a process in a manner similar to hardware interrupts. If the signal is not currently blocked (temporarily ignored) by the process, it is delivered to the process by the following steps:

1. Block further occurrences of the same signal during the delivery and handling of this occurrence.
2. Temporarily suspend the execution of the process and call the handler function associated with this signal.
3. If the handler function returns, then unblock the signal and resume normal execution of the process from the point of interrupt.

There is a default handler function for each signal. The default action is usually exiting or core dump. A process can replace a signal handler with a handler function of its own. This allows the process to trap a signal and deal with it in its own way. The SIGKILL and SIGSTOP signals, however, cannot be trapped.

Signal Trapping

After receiving a signal, a process normally (by the default signal handling function) either exits (terminated) or stops (suspended). In some situations, it is desirable to react to specific signals differently. For instance, a process may ignore the signal, delete temporary files before terminating, or handle the situation with a longjmp.

The system call sigaction is used to trap or catch signals

```
#include <signal.h>
int sigaction(int signum,
const struct sigaction *new,
 struct sigaction *old);
```

where signum is the number or name of a signal to trap. The new (old) structure contains the new (old) handler function and other settings. The

handling action for signum is now specified by new, and the old action is placed in old, if it is not NULL, for possible later reinstatment.

The struct sigaction can be found with man 2 sigaction, but you basically can use it in the following way:

```
struct sigaction new;
 new.sa handler=handler function;
 new.sa_flags=0;
```

The handler function can be a routine you write or one that is defined by the system. If handler function is SIG_IGN, the signal is subsequently ignored. If it is SIG_DFL, then the default action is restored. The new handler normally remains until changed by another call to sigaction.The sa_flags control the behavior of the signal handling. For example, sa_flags=SA_RESETHAND automatically resets to the default handler after the new signal handler is called once.

We now give a simple example that uses the sigaction system call to trap the SIGINT (interrupt from terminal) signal and adds one to a counter for each such signal received (Ex: ex10/sigcountaction.c). To terminate the program type ctrl+\ or use kill -9.

```
 #include <signal.h>
 #include <stdio.h>  ‘
 void cnt(int sig)
{
static int count=0;
printf("Interrupt=%d, count=%d\n", sig, ++count);
}
 int main()
{
struct sigaction new;
struct sigaction old;
new.sa_handler=cnt;
new.sa_flags=0;
sigaction(SIGINT, &new, &old);
printf("Begin counting INTERRUPTs\n");
 for(;;); /* infinite loop */
}
```

If the signal handler function, such as cnt here, is defined to take an int argument (for example, sig), then it will automatically be called with the signal number that caused a trap to this function. Of course, counting the number of signals received is of limited application.A more practical

example, cleanup.c, has to do with closing and deleting a temporary file used by a process before terminating due to a user interrupt.

Review & Self Assessment Question:

Q1- What do you mean by system Level I/O?
Q2- What are the operations on files?
Q3- What do you mean by Directory Access?
Q4- What is Virtual Address Space?
Q5- Explain process life cycle?
Q6- What do you mean by Process Termination?
Q7- What is Signal Trapping?

Further Readings

Linux Operating System Richard Petersen
Linux Operating System Paul S. Wang
Linux Operating System by David Maxwell and Andrew Bedford
Linux Operating System by Richard Blum and Christine Bresnahan
Linux Operating System by Bhatt P.C.P

BIBLIOGRAPHY

Linux Operating System by Richard Petersen
Linux Operating System by Paul S. Wang
Linux Operating System by Love
Linux Operating System by Isaak Seel
Linux Operating System by Karim Yaghmour and Jon Masters
Linux Operating System by Jason Cannon