Getting Started with Yellow Dog Linux

Second Edition

Getting Started with Yellow Dog Linux

First Edition

Written by John Worsley, Andrew Brookins, and Kai Staats
Cover Art by Jake Fedie, © Terra Soft Solutions, Inc. 2000
Published by OpenDocs, LLC, Portland, Oregon

Second Edition

Revised and Reset by Kai Staats
Cover Art by Jake Fedie, © Terra Soft Solutions, Inc. 2003
Published by Codra, Greeley, Colorado

Dedicated to the Terra Soft Team
"My greatest appreciation to the most dedicated, motivated, and enjoyable staff I could ask for. It has been a pleasure working with each of you--Amanda, Dan, David, Jake, Shauna, and Troy. Looking forward to the next four years!" --kai

Introduction

Welcome to Getting Started with Yellow Dog Linux.

This book covers everything you will need to know in order to get your PowerPC computer up and running with Yellow Dog Linux™ from Terra Soft Solutions®, Inc.

Please note this book assumes you have access to the Yellow Dog Linux website for the **Guide to Installation** *as it is available only online*. Following this Introduction and installation, this book provides an explanation for how the computer "powers-on" and "boots-up". You are then given an introduction to and a basic set of tools to work with the graphical user interface (GUI).

If you are interested (and we suggest you at least give it a go), the second half of this book looks at the "command line interface" (shell) as it relates to basic configuration through an introduction to systems administration. *You may be surprised by the power of the shell!*

Additionally, this book offers a quick introduction to popular applications, general troubleshooting, a glossary of terms, and appendices with additional information.

Who should read this book?

Getting Started with Yellow Dog Linux is for any individual interested in running Yellow Dog Linux on a PowerPC computer, with emphasis on Apple computers. This book will introduce you to Yellow Dog Linux, regardless of your previous familiarity with Linux (if any), and describe how to quickly become comfortable with its use.

This book is not for the complete novice, and assumes some knowledge of the basic workings of other operating systems, such as Apple's Mac OS™ or Microsoft Windows™ , drawing parallels between the Mac OS™ and Windows™ GUIs (Graphic User Interfaces) and the "KDE" and "Gnome" Linux Desktop Environments for Yellow Dog Linux. While the underlying systems between Mac OS™ , Windows™ , and Linux differ, what is most important to one learning how to *use* a new system is the *User Interface*.

Overview

Part I, Getting Started

Chapter 1, Preparing for Installation, discusses some things you should consider before installing Yellow Dog Linux. This is a good chapter to read, even if you have worked with Linux extensively before, as sometimes the simplest assumption can cause the greatest headache.

The Guide to Installation is updated often, based on receipt of ongoing feedback from Yellow Dog Linux users, and is therefore not included in this book. It is *imperative* that you *download the latest Guide to Installation* to accompany this book:

www.yellowdoglinux.com/support/installation/

Chapter 2, Booting Up!, provides an educational overview as to what occurs internal to your computer when it powers-on, and how to login into Yellow Dog Linux through BootX or "yaboot".

Part II, The Graphical User Interface

Chapters 3. Introduction to the Graphical User Interface provides some background on new choices regarding your Linux graphical user interface. Then you will dive right into the characteristics of the default GUI, some familiar, others new. This chapter draws some correlations between Mac OS™ , Windows™ , and Yellow Dog Linux to help you feel right at home.

Chapter 4, Customizing the Graphical User Interface leads you through the process of customizing your Linux desktop to match the familiarity of another operating system or to create your own environment.

Chapter 5, Graphical File Management takes you to the next level of using Yellow Dog Linux with an introduction to Konqueror, the default Linux graphical file manager with a discussion of file and data management, organization, and searching.

Chapter 6, Key Applications & Utilities quickly introduces you to the common applications such as word processors, spread sheets, image manipulation; CDs, MP3s, and RealAudio, as well as configuring your printer, network, and modem.

Part III, The Linux Command Line
This section addresses advanced topics, including Linux-specific techniques and command line applications intended for the user who wants to go beyond the graphical interface. It is divided into two chapters.

Chapter 7, The Power of the Shell introduces the basics of the Linux command line interface. It introduces Bash (the GNU "Bourne-Again Shell") and covers the essentials of how to navigate and manipulate the Linux filesystem from the command line.

Chapter 8. Basic Systems Administration introduces you to topics such as user management, permissions configuration, and system-level service management.

Part IV, Troubleshooting

This section contains *Chapter 9, Common Configuration Problems* and *Chapter 10, Where to Find Help*.

Terminology

Technical terminology can sometimes confound the user and impede progress over an otherwise easily understandable topic. To try to minimize these semantic issues, the following sections describe several potential points of confusion and give a foundational as to what you should expect from this book in reference to them.

A Folder by Any Other Name...

Over the years, many conventions and naming schemes have been used to describe the hierarchy of a modern computer's filesystem. With Mac OS™ or Windows you are familiar with *folders*, an icon within which you may store documents, aliases, applications, even more folders.

Linux is functionally no different in this regard, but the naming conventions themselves vary. In the graphical environment, Linux directories are sometimes referred to as folders as with Mac OS™ and Windows.

However, you may find that to share a common vocabulary and understanding with other Linux users, or if you choose to dive into the Linux command line interface (*shell*), the most common equivalent of *folder* is *directory*. In addition, a *sub-directory* is a directory within a directory and is often referred to as a *nested directory*. Finally, a *path* is a complete set of *nested directories* found in a particular location, where each *nested directory* is separated from the previous by a "/". Throughout this book, each of these terms will be applied when it is most relevant to the context. Typically, *directory* will be used when discussing a specific directory

on the system outside of a graphical context. *path* will be used when describing a full set of *nested directories*.

Links and Aliases
As *directories* are to *folders*, so are *links* to *aliases* in Linux. Chapter 5 covers the creation and management of links from a graphical interface while Chapter 7 covers the creation and management of links from the shell.

Programs, Executables, Applications
Another element common to all operating systems is the "program", which is referred to in Linux as an *application*, *executable*, or *binary,* with some differentiation between the three.

The only notable difference is that an *application* typically refers to a program in its entirety (including any associated library files, input files, and preferences files that may be required to run it), whereas *binary* and *executable* are specific references to the *single* file that you actually execute, or *run* --the file that you choose from a Start menu or double-click on the Desktop.

While is it important to note the term *application* will, for the most part, be used in this book as to refer to both binaries and executables, as "program" is often a common, singular reference in Mac OS™ and Windows™.

More than One Mouse Button?

Before you get very far with this guide, or Yellow Dog Linux, the authors of the book and of the software urge you to immediatley *consider investing in a two or three-button mouse*. While Mac OS™ has always been built with expectation of a single-button, Linux's roots have evolved it's graphical components in such a fashion as to all but require more than a single mouse-button in order to achieve efficient usage from the system.

The second and third buttons take only minutes to become familiar and offer a recognizable increase in performance and efficiency.

If you insist on keeping a single-button mouse, however, YDL by default **does** recognize the F11 and F12 keys as a middle-click, and right-click, respectively, and can emulate the functionality of a three-button mouse.

How Does Linux Work?

Let's start with how non-UNIX operating systems function. Inherent to their architecture is a *bundled* approach whereby the *kernel* (at the center, the 'heart' of

an OS) intimately incorporates the fundamental *device drivers* (enabling software to motivate hardware to do something--such as make the mouse pointer move) and *windowing server* (what provides the graphical user environment). The draw-back is that if one layer crashes, it is possible the entire system will lock-up (as we have all experienced).

Typical non-UNIX architecture:

user (you)

apps (Microsoft Office™ , Adobe Photoshop™)

server/window manager

3rd party device drivers (printers, scanners, PCI cards, etc.)

kernel (contains basic device drivers)

firmware (a software program stored in a chip on a computer's motherboard that boots the system into an operating system; “bios” on x86 computers)

hardware (your computer)

UNIX systems, Linux included, differ in that each layer is for the most part independent from the others. So if the graphical server (XFree86) dies unexpectedly, the kernel and associated device drivers just keep on going, reporting an error and perhaps restarting the service that died.

In addition the kernel can include only a very basic set of drivers built-in, reducing the complexity and size of the kernel for faster, more reliable performance on unique hardware. If you are going operate Linux on a cell phone or hand-held game station, it is not necessary to use a kernel that includes modem or ethernet drivers. So don't! *The flexibility of Linux*.

Typical UNIX architecture:

user (you)

apps (OpenOffice, Mozilla, Evolution, XMMS)

window manager (KDE, GNOME)

servers (XFree86, Apache, Sendmail)

shells (for command line access)

device drivers (printers, scanners, PCI cards, etc.)

kernel (may contain basic device drivers built-in)

firmware (a tiny OS that sits in ROM--gets the machine booted up; referred to as “bios” on x86 PCs)

hardware (your computer)

Do I Have to Use the Command Line?

No, you do not. The Linux graphical Interface has matured to the level that you would expect from any other operating system. However, you may find that instead of fearing the command line interface (*shell*), you will find it be a powerful tool that is not as complicated nor as confusing as you once thought. *You may even find you can’t live without it!*

Feeling Lost? Overwhelmed?

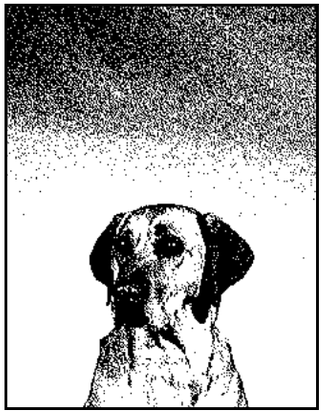
It is both important and helpful to note that in working with Linux, whether through a graphical environment or from the command line, is literally editing a text file. Configuration files and Preferences alike are simple text files that store these settings.

So if you are configuring a server and get lost, take a deep breath and remember you need only locate and then edit the correct configuration file, restart a server if required, and you are up and running.

A Request for Feedback

If you have any feedback on the Yellow Dog Linux product, you can contact Terra Soft, the developers of Yellow Dog Linux, at:

suggest@yellowdoglinux.com



III. The Linux Command Line

The Power of the Shell

A Brief Introduction to the Shell

For new users, interacting with Linux via the *shell* (command line interface) may be confusing, perhaps a bit stressful. With a basic understanding and some knowledge of fundamental commands, however, you may find the command line to be simpler, more efficient, and more powerful than expected. The command line offers rapid movement through and management of the Linux operating system. In addition the command line offers a powerful suite of commands, scripts, editing and file query features. As you will learn in the following pages, you can even launch multiple images or files into a graphical application from the command line *with a few, simple keystrokes*.

Underlying the graphical KDE components of your Yellow Dog Linux is the traditional power of UNIX, which historically has always been more command line oriented than graphical. The “command line” is a text-based *prompt* where you can navigate mounted *volumes* (drives) and *execute* (launch) applications. This prompt is typically called a “console” or a “shell.”

KDE and Linux have both matured to the degree where you can achieve the great majority of your daily user tasks without having to use the shell. However, a basic familiarity with the shell can be very useful in circumstances that demand a more hands-on approach. Additionally, some very useful applications exist which do not have a graphical counterpart as the scope of that particular application is narrower than one requiring a graphical user interface.

Understanding the Prompt

The traditional UNIX prompt is intended to concisely tell you the following: what machine you are on, who you are logged in as, and what directory you are presently working in. The current directory is typically surrounded by brackets, and it's followed by a dollar sign and a space. The cursor then blinks as it patiently waits for your input. An example of such a prompt would be as follows:

```
[user@host root]$
```

The above prompt offers you this information: you are logged in as the user “root,” you are on a machine called “host,” and you are working in a directory called “root.”

<warning>
Know Your Relative Paths!

It's important to note that, in the interest of readability, the present working directory is both relative and context-specific. Thus, the directory named “root”

could signify either “/root,” or “/usr/local/example/root.” The UNIX command “pwd,” which stands for “present working directory,” displays the full path that you are working in.

The Nature of Command Line Input

Though the prompt and input grammar of the UNIX shell may at first seem somewhat arcane, you will at some point find that there is a simple and consistent rhyme and reason to it. A simple input command to the shell consists of two major parts: the *command*, and its *arguments*.

The two are always separated by “white space,” which is simply a space, or series of spaces; the first word is always the command. It follows, then, that the series of words and characters following the command are the arguments, which are modifiers that instruct the command how to behave.

Arguments vary widely from application to application. They can potentially include names of output files, input files, instructions on the amount of output to provide, or even a simple request to display what arguments the command can understand or require. If an argument does not represent a file or literal string of characters, it is usually preceded by one or two dashes and is called a “flag” (e.g. -K, --help, --version).

A flag can be either a simple toggle to switch a mode on or off (e.g. --verbose), or it can require another argument immediately after it that is separated by white-space, or sometimes a mathematical equals-sign (e.g. -o my_output_file.txt, --width=50). Again, this is a general overview of the nature of inputting commands; the flags and arguments available to you vary from program to program.

Thus, an example input command to the shell might read as follows:

```
[root@host root]$ ls -la --color=auto
```

Breaking this input down, the preceding *command* is ls (the UNIX program that tells the system to list contents), and its *arguments* consist of -l, -a and --color=auto. In this context, these flags instruct “ls” to display a long output, show all files (even if they are hidden files), and automatically display color when appropriate. These flags are, of course, specific only to “ls” and will have variable effects upon other programs (if they have any effect at all). Most UNIX applications support the --help flag, which looks similar to this output:

```
[root@host root]$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuSUX nor --sort.

-a, --all                  do not hide entries starting with .
```

```
-A, --almost-al          do not list implied . and ..
-b, --escape             print octal escapes for non-graphic characters
      --block-           size=SIZE use SIZE-byte blocks
-B, --ignore-backups     do not list implied entries ending with ~
-c                       with -lt: sort by, and show, ctime (time of last modification
                         of file status information) with -l: show ctime and sort by
                         name otherwise: sort by ctime
-C                       list entries by columns
      --color[=WHEN]     control whether color is used to distinguish file types.
                         WHEN may be 'never', `always', or `auto'
-d, --directory          list directory entries instead of contents
-D, --dired              generate output designed for Emacs' dired mode
-f                       do not sort, enable -aU, disable -lst
-F, --classify           append indicator (one of */=@|) to entries
...
```

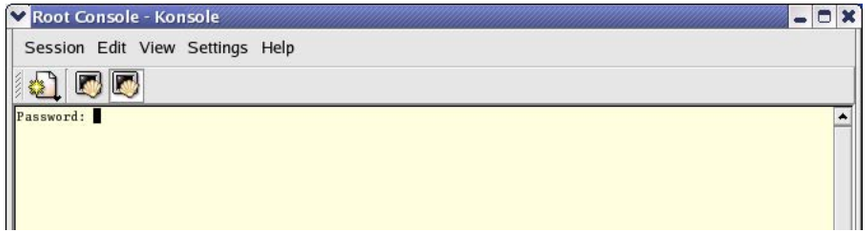
Konsole - Accessing the Shell

KDE comes bundled with its own advanced implementation of the *shell* interface, called “Konsole.” Konsole differs from many other shells in that it allows for more advanced visualization features (such as transparency, and movable components), as well as multiple “virtual shells” within a single window. The ability to have numerous shells in a single window allows related shell sessions to be easily accessed from one another.

Starting Konsole

Start the Konsole application is to click on the Konsole Icon that is located on the KPanel, next to the K Menu icon. It resembles a small computer monitor. When launched, you will be greeted with black text on a white background (the default color scheme, which can be modified to your liking via the Settings options), as follows.

Congratulations! You are now *in the shell*.



Virtual Shells (Multiple Sessions)

To create a new shell within a single Konsole window, click the “New” button, usually in the lower left-hand corner of the window. Note that the bottom of the Konsole window has its own mini-taskbar which displays all open shells within that instance of Konsole.

Holding the left-mouse button over the “New” button presents a few options unavailable to you in the standard shell.

All modular customization options are available to you in Konsole by simply right-clicking on the screen. This opens a context-menu from which you can change the color scheme and visualization options beneath “Schema,” change menu modes, move or hide the scrollbar, and change the font family and size. Experiment as much as you like, personalizing your own Konsole session, but remember that the settings will not be saved unless you right-click on the Konsole window and choose “Save Options” at the bottom of the context menu. (This context menu also exists in the “Options” menu at the top of the window.)

<tip>
Toggling between Shells

If you have a few shells open in a single Konsole window that you find yourself switching between frequently, it can be helpful to know that by holding down the shift button on the keyboard, you can use the left and right arrow keys to toggle back and forth between open shells in the same window.

Exploring the Shell: BASH Essentials

Many users consider the command prompt the most intimidating part of any operating system. However, the command prompt, or *shell*, is also the environment that gives Linux and UNIX much of their power and flexibility.

The depth and breadth of the command prompt's scope are quite impressive. It may be used for more tasks than can be comprehensively and succinctly summarized. To begin with, it is most notably used to execute programs, manage the filesystem, modify users and groups, and to manage shell *scripts* that create the Linux system environment.

Essential Shell Commands

The remaining sections of this chapter outline and describe some *must have* commands for users wanting to spend time at the command prompt. You will first learn your way around the shell in the following sub-section, Navigating the Shell.

Once you know your way around a bit, we will proceed into the section on intermediate shell techniques. This section covers how to suspend and activate “jobs” (running processes) in and out of the “foreground” and “background,” how to kill run-away processes, create aliases and links for common routines, and use the built-in BASH history features.

Finally, if you have not satisfied your appetite for the more obscure UNIX knowledge, you may want to delve carefully into some advanced techniques, involving powerful tricks available to BASH shell gurus that can increase your efficiency and productivity at the command line by re-directing input and output between processes, stringing commands together with *pipes* and *backticks*, and automating tedious routines with special built-in bash commands.

<note>
A Note on Shell Conventions

Within BASH there are many characters that have special, non-literal meanings when used in a command. Covering all of these is outside the scope of this book, but some basic ones that we will make reference to throughout the rest of the remaining chapters are:

- The *tilde* (±) represents the currently logged in user's home directory. For most users, this is /home/[username] where "[username]" is the name of the user logged in. This is not always the case, however (e.g., the root user's home directory is typically /root).
- The *single dot* (.) represents the present working directory of the user initiating the command.
- The *double dot* (..) represents the working directory one directory below the present working directory (e.g., from /home/user the double dots represent /home
- The *asterisk* (*) represents a wildcard symbol. When passed to a BASH command without any other characters, it will represent all files in that working directory. By attaching other characters to the asterisk, you can use the wildcard to address several files by their common filename components (e.g., "*.txt" refers to any file ending in ".txt" and "song*mp3" will refer to any filename beginning with "song" and ending with "mp3").
- The *double quote* (") and *single quote* (') symbols are extremely important when working within the shell. Shell commands separate their arguments or parameters from one another by *white space* (which is usually a *single* space).

However, sometimes a single argument may *require* white space (e.g., any filename with a space in it, such as "My Files"). The shell allows you to group words and symbols together as a literal argument by surrounding the argument in either single-quotes or double-quotes (e.g., ls "My Files" or ls 'My Files').

We will cover other special characters in BASH as appropriate, but knowing the above will aid in getting a solid understanding of the frequently cryptic-appearing shell commands.

Navigating the Shell

To acclimate to the environment of the shell, it is vital that you get in the habit of being able to regularly "get your bearings," so to speak. Before you learn this skill, it is easy get lost in what often seems like a maze of obscure filenames and paths. The idea is that once you have a sense of where you *are*, you can begin to focus more on where you are *going*.

The goal of this section is to not only give you these skills, but also to instruct you on how to fluently speak the language of the shell. By doing so, you will learn to quickly, efficiently and confidently navigate the environment through the power and elegance of the command prompt.

We will spend this section introducing you to the most integral commands toward this goal: ls, pwd, cd, pushd (and popd), file, stat, more (and less), grep, and find.

To begin with, we will review the ls command. In the following sections, we explain the basics of working with these commands at the shell. When applicable, we also include basic usage of the command, along with command-line flags and arguments.

The List Command: ls

The ls command stands, as you might have guessed, for *list*. Put simply, you use the ls command to list the contents of a directory. To illustrate, let's take the following directory structure from a home directory. If you enter the shell you should see something similar to the following.

```
[admin@imac admin]$
```

This is the shell prompt, which we introduced in Chapter 6 within a brief introduction to the shell. To review, the above line tells you succinctly that you are the "admin" user located at ("@") the machine named "imac." As a demonstration, enter the following command into your own shell:

Using ls

```
[admin@imac admin]$ ls /
```

Once you have typed ls, you must press ENTER to execute the command.

The ls command is very powerful, and can be used for a number of functions. Obviously, the most popular of these is the listing of files within a directory. However, ls has a host of features that are also quite useful. You can use "flags" (special arguments) to enable ls to provide more information than its basic output.

For example, if you type:

Using ls to see Hidden Files

```
ls -a
```

The -a flag is short for -all, or show all entries. If you examine the resulting output you will see that there are now several additional files displayed. They look something like this:

```
./  ../  .bash_logout  .bash_profile  .bashrc  .mailcap  .screenrc
```

Notice the periods (".") which begin the filenames. Linux and Unix systems use this leading period to signify that a file that is *hidden*. The files that begin with a period are typically configuration files or configuration directories.

Using the -l flag displays the long listing, which provides extended details about the file or files you are viewing.

Using ls with Long Output

```
[admin@imac admin]$ ls -l
total 4
drwx------    2 admin    admin    4096 May  4 23:25 tmp
```

The output is very clearly separated into a series of columns. The most important of these columns are the first, third, fifth, and sixth. Respectively, these columns display the *permissions* on the file, the *owner* of the file, the *group* of the file, the *date* that file was last modified, and *which file* you are viewing.

In our case, we are viewing the tmp/ file in the admin's home path. The tmp/ file is a directory (everything within Linux is technically treated as a file, even if it is a directory or a running process!) that has the permissions set to owner-only read, write and execute. The owner of the file is "admin," the group is "admin" and the last time the file was modified was May 4th (of the current year) at 23:25.

<note>
On Permissions

For more detailed information on explaining and decoding the obscure-looking "drwx------" permissions string in the previous example, see Chapter 8's section on permissions, Managing Ownership and Permissions.

It is commonly possible in Linux to "group" flags together. This is the case with ls. You can use the the -a and -l flags together with the ls command to display the long listing of all contents in the directory, including the hidden files mentioned earlier.

Combining Flags with ls

```
ls -al
```

The l s -al command will display something similar to the following:

```
drwx------    4 admin    admin    4096   May  4 23:25 ./
drwxr-xr-x    5 root     root     4096   May  4 23:25 ../
-rw-r--r--    1 admin    admin    24     May  4 23:25 .bash_logout
-rw-r--r--    1 admin    admin    122    May  4 23:25 .bash_profile
-rw-r--r--    1 admin    admin    85     May  4 23:25 .bashrc
-rw-r--r--    1 admin    admin    141    May  4 23:25 .mailcap
-rw-r--r--    1 admin    admin    3471   May  4 23:25 .screenrc
drwx------    2 admin    root     4096   May  4 23:25 .xauth
drwx------    2 admin    admin    4096   May  4 23:25 tmp
```

Directory Commands: pwd, cd, pushd, and popd

If ls can be thought of as your eyes, showing you what's directly in front of you, the directory commands exist to fulfill the need of your legs. That is, they serve to move you around the system. Using this metaphor, the command pwd, which stands for *present working directory*, is useful to get your bearings and determine where in the filesystem you are currently standing. Sometimes the location of your present working directory can be a bit ambiguous or confusing; it is these times when the pwd command can become quite useful.

With a working knowledge of the general structure of how the filesystem is composed, you can use the idea of *where* you are in the filesystem to move up and down through directories with the cd command, which stands for "change directory."

Finally, there are a pair of more advanced tools called pushd and popd that allow an experienced user to more easily navigate commonly accessed paths.

Get your Bearings: pwd

At the Linux command line, it's a good idea to always have an idea of what directory you are in. While the command prompt itself is usually configured to show you the name of the top-most directory of your present working path, this can sometimes be misleading if you are not careful. For example, examine the prompt:

```
[user@imac local]$
```

You can see that the name of the top-most directory is "local." However, it is unclear as to whether or not this means that your present working path is /usr/local, /var/local, or any other number of paths ending in /local. The prompt shows "local"

only as a small reminder. This can be confusing, particularly when you have many shells open performing their own separate operations.

For this reason, you have the pwd command at your disposal. In the above example, if you had left a shell open and forgotten just where you had left yourself, you could "get your bearings" by typing pwd and pressing ENTER with no further arguments.

Using pwd

```
[user@imac local]$ pwd
/usr/local
[user@imac local]$
```

As you can see, we have successfully used pwd above to determine that the "local" was, in fact, a reminder for the complete path /usr/local.

Changing Paths

To actually *change* the *present working directory* at the command line, you need to take advantage of the cd command.

The syntax for cd is simply cd [path], where "[path]" is the directory you wish to *change into*. This path can be an *absolute* path (e.g., /usr/local/src), or a *relative* path (e.g., local/src, while already working in the /usr path).

Changing directories with cd

```
[user@imac user]$ cd /usr/local/src
[user@imac src]$ pwd
/usr/local/src
[user@imac src]$
```

This example illustrates a user named "user" changing from their home directory (/home/user) to /usr/local/src by invoking cd with an absolute path. It also shows the usage of pwd to verify the function of cd.
The cd command also recognizes the two special single period (".") and double period ("..") paths. The single period represents the present working directory, and can be explicitly attached to a path to be sure that you are changing from the present working path. The pair of dots represents the directory *one below*> the present working directory. For example, if you are working in the path /usr/local/src, the single period will represent the complete, current path (/usr/local/src), while using double periods will represent the path just below your working path (/usr/local).

Special paths may be used in conjunction with real paths in order to create complex navigational uses of the cd command.

Changing directories with cd and special paths

```
[user@imac user]$ cd /usr/local/src
[user@imac src]$ pwd
/usr/local/src
[user@imac src]$ cd ../../src
[user@imac src]$ pwd
/usr/src
[user@imac src]$
```

The above example begins with the user named "user" changing from their home directory to /usr/local/src with an absolute path. The user then appears to change their mind, and wishes to change to the /usr/src directory. This is achieved above by stringing together two double period paths, signifying to go down two levels in the filesystem (to /usr), and then up one to src (/usr/src).

Advanced Path Actions: pushd and popd

If you have to spend any large period of time at the command line, you will probably ask yourself if there is a simple way to return to a recently visited path. This will become particularly apparent if you find yourself having to re-type long and easily mis-typed paths, such as /etc/sysconfig/network-scripts/.

The good news is that there are a number of ways to do this in Linux. The most prevalent feature built into the BASH command-line lies in the use of the pushd and popd directory stacks.

The pushd command behaves on the surface identically to cd. It moves your present working directory to the path which you pass to it. However, in addition to changing your present working directory, it *pushes your* previous working directory onto a stack of previously visited paths. By typing popd, and pressing enter, you "pop" your present working directory, and return to the last visited directory on the stack. You may thus use popd as a means to continually return to a common path or set of paths.

Using pushd and popd

```
[user@imac user]$
[user@imac user]$ pushd /etc/sysconfig/network-scripts/
/etc/sysconfig/network-scripts/ ~
[user@imac ethernet]$ pushd /root
/root /etc/sysconfig/network-scripts/ ~
[user@imac /etc]$ pwd
/root
[user@imac /etc]$ popd
```

```
/etc/sysconfig/network-scripts/ ~
[user@imac ethernet]$ pwd
/etc/sysconfig/network-scripts/
[user@imac ethernet]$ popd
~
[user@imac user]$
```

This example illustrates our friend "user" using pushd to change from his home directory (/home/user) to a rather long working path, /etc/sysconfig/network-scripts/. From this path, user decides that he needs to make a brief visitation to /root and does so again with the pushd command.

After verifying the path change with pwd, the user then changes his mind and decides to return to the most recent working directory on the stack, and by issuing the popd command, he finds himself back in /etc/sysconfig/network-scripts/. Finally, "user" decides that the whole trip was apparently fruitless, and returns to his home path (/home/user, abbreviated by the tilde symbol) with a final execution of popd.

<note>
A Note on the Directory Stack

Remember that when you change directories, the stack is not maintained automatically. It must be populated by the pushd command, and only then can you utilize the popd command to fall back on directories in the stack.

If invoked without a defined directory stack, you might see a message like this:

```
[user@imac user]$ popd
bash: popd: directory stack empty
[user@imac user]$
```

Investigative Commands: file, stat, more or less and grep

Between ls and the family of cd-related commands, you have the necessary tools to get around the filesystem and look at things on a surface level. That is, you are able to view files by their *filenames*, but not for their contents.

In order to be able to investigate beyond just the file's name, size and location, you will require a new set of investigative tools. These include file and stat, for general characteristics, more or less for examining the contents of text files, and grep for searching for text patterns in one or more files.

General Investigation: file and stat

The file Command

You can use the file command to determine a file's type. This can be a very useful command if you need to know the type of document that a particular file is, but lack the knowledge to determine this by the filename, extension, or contents. When you type file [filename], where "[filename]" is the file you wish to report on, the program performs three sets of tests to determine what kind of file is being looked at. These tests are generally quite accurate in returning the type of file that is found, and frequently return extra information about the file in question once its file type is derived.

Using file

```
[user@imac user]$ ls
Desktop  letter.txt  mysong.mp3  personal  song.mp3
[user@imac user]$ file letter.txt
letter.txt: ASCII text
[user@imac user]$ file song.mp3
song.mp3: MP3, 128 kBits, 44.1 kHz, JStereo
[user@imac user]$
```

This example illustrates once again our friend "user" and his adventures at the command line. Here, he appears to have forgotten what his files letter.txt and song.mp3 consisted of, and he expertly uses the file command to refresh his memory. As you might expect, letter.txt is reported to be an ASCII text file, while song.mp3 is found to be an MP3 audio file.

Since file recognizes the MP3 audio format, it able to report that song.mp3 was encoded at 128kBits, with a sampling rate of 44.1kHz, in joint-stereo.

The stat Command

Another great investigative tool within BASH is stat, which is used to view the status and file characteristics of a particular file. The syntax of stat is, as you might have guessed by now, stat [filename], where "[filename]" is the name of the file that you wish to retrieve status on.

The information returned by stat is much more technical than that of file. It returns the complete size of the file, including the number of physical blocks that the file is occupying on the hard disk. Additionally, it shows you the *type* of file being examined (e.g., Directory, Regular File, Symbolic Link, etc), the permissions, and the system IDs of the user and group owners.

Finally, perhaps most usefully, stat shows a series of date/time stamps indicating when the file being examined was last *accessed* by the system (this includes reads, as well as writes), when its contents were last *modified*, and when any characteristic was *changed*.

<note>
Modified versus Changed

The difference between “last modified” and “last changed” lies in the context of the change that was recorded: if the actual contents of the file were altered, it is recorded as a “modification.” If an external characteristic of the file was changed, such as its file name, ownership, or permissions, it is recorded as a “change.”

Specific Investigation: more or less, and grep

Files on your Linux filesystem can generally be grouped into two categories: ASCII and binary. The technical differences are just that: quite technical. In short, though, a *binary* is programming code that has been *compiled* into *machine language*. A binary file's contents are therefore unreadable without the use of special tools.

Conversely, an ASCII file consists only of text characters, and therefore are easy to read once accessed. From the command line you have a variety of text editing tools at your disposal, but more often than not you will find yourself wanting to be able to peer into a file's contents without the relatively cumbersome complexity of even a lightweight text editor. For this reason, bash provides two valuable commands to allow you to view the contents of an ASCII text file: more and less.

The more and less commands are largely identical in function. The less command has the distinction of being newer, and less “primitive” in some of the more advanced terminal emulation features. For most operations, you probably will not be able to tell the difference, but we will rely on less for our examples.

The syntax for less is less [filename], where “[filename]” is the name of the file that you wish to view. If the file is too large to fit in a single shell, it will *page* and wait for you to either press ENTER to scroll down line by line, or hit the SPACEBAR in order to scroll a full page down. Other common commands to use within less are “b” to scroll backwards and “q” to quit.

On large documents, you may wish to pass the -m flag, which causes less to show the percentage of the file read on each page of output.

Using less

```
[user@imac user]$
[user@imac user]$ less -m letter.txt
August 30, 2000.
From the desk of Test P. User.

To whom it may concern,

This is a letter that I am writing to you with the intent to clarify
the purpose for my writing this letter. Without being too redundant,
```



```
I would like to begin by stating that this is a letter that I am writing to
you with the intent to clarify the purpose for my writing this letter.

Furthermore, this is a just letter that I am writing to you with the intent
to clarify the purpose for my writing this letter. Without being too
redundant, I would like to begin by stating that this is a letter that I am
writing to you with the intent to clarify the purpose for my writing this
letter.

Similarly, this is a also letter that I am writing to you with the intent
to clarify the purpose for my writing this letter. Without being too
redundant, I would like to begin by stating that this is a letter that I am
writing to you with the intent to clarify the purpose for my writing this
letter.

The best way to explain this is to state that this is a letter that I am
80%
```

As you can see, by passing the -m flag, less shows that 80% of the document has been displayed, rather than simply prompting you to continue.

The less interface also provides some powerful search facilities to search up and down a file. If you wish to search forward in a file for a pattern, hit the “/” key, and type the desired pattern immediately after the slash. When you press ENTER, less will search the entire file for the pattern, highlighting each found instance. To repeat the search, just hit the “n” key as many times as you wish to repeat the search.

To search backwards in less, just use the “?” key in the same manner that you would use the “/” to search. Similarly, to repeat the reverse search, hit the “n” key as many times as you wish to repeat the search.

<note>
Piping Output to less

One of the most common ways to use the less command is via a pipe. Without being too specific on the nature of pipes, by attaching the argument | less after any command, you “funnel” that program's output directly into the less program. The pipe symbol (“|”) is created by holding SHIFT and pressing the backslash (“\”) key.

This can be a very useful trick if you are running a program whose output spans more than a single shell's height, when it is not possible or convenient to scroll backwards.

<note>
Using head and tail

For extremely large files, the head and tail commands can be very useful if you are only interested in a few lines worth of their contents.

You can view the first and last 10 lines of a file by using the syntax head [filename] and tail [filename], respectively, where "[filename]" is the name of the file you wish to view a portion of.

If the search facilities in less are not wide enough in scope, you will probably be interested in the remarkably versatile grep command. grep allows you to search for a given search pattern in numerous files.

The syntax for grep is grep [pattern] [source], where "[pattern]" is the word or phrase that you wish to search for, and "[source]" is the file (or files) that you wish to search within for that pattern. This use of grep will output each line found in the source which matches the pattern. If grep finds no matching lines, it will not issue any output.

Using grep on a single file

```
[user@imac user]$ grep whom letter.txt
To whom it may concern,
[user@imac user]$
```

This example shows the use of grep by "user" to display each line in "letter.txt" that matches the "whom" search pattern.

If you pass multiple files to grep for analysis, grep will precede each matching line with the name of the file in which a match was found. In this way, you can use grep to find which files (of many possible files) include what you are searching for.

Using grep on multiple files

```
[user@imac user]$ grep whom letter.txt song.mp3
letter.txt:To whom it may concern,
[user@imac user]$
```

This example checks both the "letter.txt" file used in the previous example, as well as the "song.mp3." Despite the fact that it's obviously absurd for our friend "user" to be checking his MP3 audio file for a piece of text, this example illustrates that when searching multiple sources, grep outputs not just the line found, but the name of the file in which the pattern matched.

A more real-world example of using grep might involve checking multiple Perl ".pl" files for a single piece of code that you might have lost, or searching through multiple mailbox files for an important e-mail that had gotten lost.

Common flags that you might want to use in conjunction with grep are -i, to search for "[pattern]" in a case-insensitive fashion, and -v to search for any line which does NOT match "[pattern]".

Using grep with optional flags

```
[user@imac user]$ grep "similarly" letter.txt
[user@imac user]$ grep -i "similarly" letter.txt
Similarly, this is a also letter that I am writing to you with the intent
[user@imac user]$
```

This example demonstrates that passing the -i flag to grep causes the search to be case-insensitive. Without passing -i in the first grep attempt, no lines are found, as the only line which contains that word begins with a capital "S."

For advanced users, you may also take advantage of the power of *regular expressions* by passing the -E flag to grep. Regular expressions are a powerful standard that allow you use special characters such as the open bracket ("["), close bracket ("]"), pipe ("|") and more to apply search criteria within the search pattern itself. Regular expressions are outside the scope of this document, but you may read more about them by typing man grep to review the complete manual page on grep.

Using regular expressions with grep

```
[user@imac user]$ grep -E "[Ss]imilarly|concern" letter.txt
To whom it may concern,
Similarly, this is a also letter that I am writing to you with the intent
[user@imac user]$
```

The -E flag invokes the capabilities within grep to search a regular expression-style search pattern rather than a literal pattern. As a simple example, the above search pattern ("[Ss]imilarly|concern") searches for any instance of the words "Similarly," "similarly," or "concern" in the file letter.txt

File Management

While the last few years have seen maturation by leaps and bounds in Linux's graphical file management systems (such as KDE's Konqueror), it certainly doesn't hurt to have the command line skills to be able to achieve your file management needs. There may be some cases where you have lost the ability to start the GUI, or you may need to do some file manipulation remotely over a network connection without the benefits of a window'd environment.

This section covers the basics of how to manage files and directories, and how to get an idea of the "bigger picture" in terms of knowing where disk space is being used, and where it is not.

Managing Files and Directories

The fundamental skills in managing a filesystem begin with knowing how to manipulate files and directories in the most basic ways. These include how to copy, create, remove and move files, directories, and their sub-directories. The commands involved with these functions in Linux are cp, mkdir, rm (and rmdir), and mv, respectively.

Copying Files with cp

The command to copy a file from one location to another is cp, short for copy. In its simplest form, it requires two arguments: the source file, and the location of the destination file to be created.

Simple Copy

Here is a simple example of copying a file from one directory to another, using cp:

```
[user@imac user]$ cp /etc/motd .
```

This examples uses cp to copy a file called "motd" (the "Message of the Day" file) to the present working directory (recall that a single period in Linux refers to the present working directory).

You can also supply a filename (rather than a directory) as the second argument if you wish create a copy with a different name than the original file.

Copy and Rename

Here is a slightly more involved example of copying a file from one directory to another, and renaming it in the process:

```
[user@imac user]$ cp /etc/motd /home/user/.old_motd
```

This example uses cp to copy the file called "motd" from the /etc path to a file in the user's home directory called ".old_motd." (Recall that the leading dot signifies that this file will be a *hidden* file, invisible to ls, unless invoked with the -a flag.)

You can also copy multiple source files to a single destination directory by supplying each file that you wish to copy as the leading arguments, and ending the list of arguments with a relative or absolute path which you wish to copy the files into.

Copying Multiple Files

Here is an example of copying several files to a new directory location:

```
[user@imac user]$ cp /etc/m* mstuff/
```

This example uses cp to copy any file beginning with "m" (recall that the "*" symbol represents a wildcard) in the /etc directory to a directory within the present working directory called "mstuff."

Removing Files With rm

The command to permanently remove a file from the filesystem is rm, short for "remove." In its most basic form, it can be used to delete a single file by typing rm [filename] where [filename] is the name of the file which you wish to remove from the system. Here is a basic example:

Removing a File

```
[user@imac user]$ ls
Desktop  motd  personal
[user@imac user]$ rm motd
[user@imac user]$ ls
Desktop  personal
[user@imac user]$
```

As you can see, passing rm the argument "motd" removed the file with that name in the present working directory. rm can just as easily accept an absolute filename (e.g., /home/user/motd would function identically), or a series of file names, separated by spaces.

<caution>
Exercising Caution with rm

When using Linux from the command line, you do not have the safety net of a "Trash" icon. Using rm will *permanently* remove the file or files passed to it, so always exercise extreme caution when using this command!

Useful Flags for rm

The most common flags that you should be familiar with for rm are the interactive -i flag, the recursive -r flag, and the force -f flag. The interactive flag is a safeguard, forcing you to confirm deletion before the file is actually removed from the system, and it can be a useful "sanity check" when removing many files at a time to include -i, just in case.

The -r and -f flags are *extremely dangerous*, particularly when wielded by a root user, and therefore extra caution must be taken with them! The recursive -r flag

causes rm to attempt to delete all files within a given sub-directory, including all subsequent sub-directories, and their files. The -f flag causes rm to suppress any warning that it might otherwise present before completing the operation (such as verifying that you do not want to delete files otherwise set to "read-only").

For most users, the damage that can be done by combining these flags with rm -rf is limited. This is because they are restricted to being able to delete only files that they own, or have been assigned ownership to. As stated earlier, though, a user with root access can completely wipe out their system with a single misplaced rm -rf command. It is for this reason that these flags are recommended only for users that are completely confident in their skills at the command line (and even then, only with the utmost care).

Making New Directories

The command to create a new directory is mkdir, short for "make directory." Any user may use the mkdir command. However, where that user is allowed to create the directory is dependent on the *permissions* in the location which the user wishes to create the new directory in. (For more on permissions, see the next section on Managing Ownership and Permissions.)

The argument passed to mkdir is either a relative or absolute path. As an example, in your home directory, you might want to create a "personal" directory to hold all of your personal files.

Creating a Directory

Here is an example of creating a relative directory:

```
[user@imac user]$ mkdir personal
[user@imac user]$ ls
Desktop  personal
[user@imac user]$
```

As you can see, the command has created a directory called "personal" within the relative path that the user was in. The complete path of this new directory is /home/user/personal, even though the argument passed to mkdir was only the relative argument "personal."

If the user had changed their current working directory to a different path (e.g., /usr/src), the same effect could have been achieved with the command:

```
[user@imac src]$ mkdir /home/user/personal
[user@imac src]$ ls /home/user
Desktop  personal
[user@imac src]$
```


This use of the absolute path illustrates the flexibility of using mkdir from the command line. To further illustrate, you may also pass multiple arguments to mkdir, separated by spaces, in a single command line statement, in order to create several directories at once.

```
[user@imac src]$ mkdir ~/personal/music ~/personal/writing
[user@imac src]$ ls ~/personal/
music  writing
[user@imac src]$
```

(Recall that the ~ or tilde, represents the current user's home directory.)

Removing Directories

The inverse of the mkdir command is logically called rmdir, short for "remove directory." The function of rmdir is very similar to that of mkdir. You need only pass it the directory, or directories, that you wish to remove as command line arguments.

If you pass rmdir a directory that is not empty, the command will not be able to complete. The rmdir command will only successfully remove directories whose contents have already been deleted, as a precaution against accidentally deleting out important areas of the filesystem. (To delete out the entire contents of an existing directory, and its sub-directories and files, see the earlier section on the rm command and its -rf options.

Here is an example, removing one of the paths we just created in the last section:

```
[user@imac src]$ rmdir ~/personal/music
[user@imac src]$ ls ~/personal
writing
[user@imac src]$
```

Just like mkdir, you may pass either relative or absolute paths to rmdir, as well as multiple paths, by passing several directories, separated by spaces, to rmdir at the command line.

Moving Files and Directories with mv

Recall that in Linux, despite their obvious practical differences, files and directories are each treated as "files" as far as the filesystem is concerned. For this reason, files and directories are moved via the same command: mv, short for "move."

The mv command functions very similarly to cp, in that it accepts as its first argument the name of the source file which you wish to move, and accepts as the second argument the destination, or location, of the moved file. Here is a simple

example of moving a file from one location to another:

Moving a File with mv

```
[user@imac user]$ mv /tmp/oldfile .
```

This example uses mv to move a file called “oldfile” that is in the “tmp” directory to the present working directory, signified by the single dot. This operation, if successful, will behave just like cp, however the file will no longer exist in the /tmp path as well, as it would if it had been copied.

This command will only be successful if the user initiating the move has both the rights to read and write the source file, as the operation removes the file from its original location. For more on read and write permissions, see the next section on Managing Ownership and Permissions.

You might think of mv as being the command used to rename files, as this function is often the net effect of its usage. Here is an example of renaming a file with the mv command:

Renaming a File with mv

```
[user@imac user]$ mv oldfile newfile
```

This example uses mv to “move” the file called “oldfile” to a new file called “newfile,” effectively renaming the file within the same sub-directory. You may pass complete paths to mv in this fashion as well, if you wish to both move and rename a file (e.g., mv oldfile /tmp/newfile).

Managing Disk Space

Another fundamental skill in system administration is managing disk space. The first step in being able to manage your disk space is knowing how it is allocated, and how much is being used. The two commands discussed in this section regarding this are df and du.

Reporting Filesystem Usage with df

The Linux command for reporting filesystem disk space usage is df. Typing df without any parameters will display each filesystem that your system has mounted (e.g., /dev/hda10), how much space that filesystem has total, how much is used, how much is available, and where that /home).

Here is an example of a couple common ways to take advantage of df:

Viewing Disk Space with df

```
[user@imac user]$ df
Filesystem   1k-blocks  Used Available Use% Mounted on
/dev/hda11     5436916   1481464   3679272  29% /
[user@imac user]$
```

As you can see, by default, df reports space in units of 1k (kilobyte) blocks. To make this more readable, you may wish to pass the human-readable -h flag.

```
[user@imac user]$ df -h
Filesystem    Size  Used Avail Use% Mounted on
/dev/hda11    5.2G  1.5G  3.5G  29% /
[user@imac user]$
```

While this is a less precise examination of the disk space, it can frequently be more readable at a glance to get an idea of what the current disk usage looks like on your system.

Reporting Disk Usage with du

While df gives you the “big picture” of what's going on in your system, you may at times want to know how much disk space is being used within a directory on a single partition. For this kind of analysis, du is available.

The du command summarizes disk usage within a passed directory or directories. Simply pass each directory that you wish to recursively investigate to du . Here is an example of using du to analyze the disk space used within the /usr/local directory:

Analyzing Disk Space with du

```
[user@imac user]$ du /usr/local
4    /usr/local/bin
4    /usr/local/doc
4    /usr/local/etc
4    /usr/local/games
4    /usr/local/info
4    /usr/local/lib
4    /usr/local/man/man1
4    /usr/local/man/man2
4    /usr/local/man/man3
4    /usr/local/man/man4
4    /usr/local/man/man5
4    /usr/local/man/man6
4    /usr/local/man/man7
4    /usr/local/man/man8
4    /usr/local/man/man9
4    /usr/local/man/mann
44   /usr/local/man
4    /usr/local/sbin
```

```
4     /usr/local/src
2756          /usr/local/RealPlayer8/Codecs
3312          /usr/local/RealPlayer8/Common
36   /usr/local/RealPlayer8/Plugins/ExtResources
10760         /usr/local/RealPlayer8/Plugins
684           /usr/local/RealPlayer8/Help/realplay/pics
1020          /usr/local/RealPlayer8/Help/realplay
1032          /usr/local/RealPlayer8/Help
20068         /usr/local/RealPlayer8
20148         /usr/local
[user@imac user]$
```

As you can see, this example reports the amount of space being used in each sub-directory within the /usr/local path, and totals all of this space in the final line for /usr/local. If you wish to omit the sub-directory analysis and focus only on the total, you may pass du the summarize -s flag:

```
[user@imac user]$ du -s /usr/local
20148   /usr/local
[user@imac user]$
```

If you wish to analyze several sub-directories, you may pass them all to du on a single command line, separated by spaces. You may wish to pass the total -c flag if you do so, to combine each passed directory's space into a grand total summary line:

```
[user@imac user]$ du -sc /usr/local /opt /var/www
20148   /usr/local
4   /opt
628 /var/www
20780   total
```

As evidenced in the grouping of the summarize -s and total -c flags above, you can group flags together with du, as you might expect with most GNU tools.

Intermediate Shell Techniques

Once you are more familiar with using the shell for basic purposes such as navigating and managing the filesystem, and searching the contents of existing files, you may wish to further educate yourself in the use of the BASH environment. This section covers somewhat more advanced shell techniques, including how to increase your efficiency at the command line through the use of BASH's dynamic history, the creation of shortcuts in the form of aliases and symbolic links, the re-direction of program output to and from files, and manually managing running processes.

Increasing Efficiency at the Command Line

Using the command line should not be a laborious or overly involved task. Much of the power of the BASH command line lies in its myriad of built-in tools that greatly increase efficiency when applied by an experienced user. The command line is admittedly arcane to initiates, but it needn't be tedious.

Using the BASH History

BASH, by default, keeps a running history of the commands that you type in a special file called .bash_history, in your home directory. You may quickly and easily access this file through a number of keyboard shortcuts, as well as the history command itself.

The simplest way to get acquainted with the BASH history is to type history and press ENTER, with no further arguments. What you will see depends on what you've been doing recently at the command line, but it will be a line by line list of the most recent commands, preceded by a number which increments with each command.

The history Command

```
[user@imac user]$ history
1  ls
2  exit
3  ls -la
4  chmod a+rx .
5  ls -la
6  ls -l song.mp3
7  ls -l letter.txt
8  chmod 740 letter.txt
9  ls -l letter.txt
10  man chmod
11  history
[user@imac user]$
```

As you can see in the above example, the BASH history facilities log each of the commands that you have recently run. The last command you see in this list will invariably be the history command itself, as it is logged before the output finishes reaching the screen.

Of course, active users don't generally want to type history in this fashion, as hundreds or thousands of lines may fill the shell. Thus, this command is often *piped* into the grep command via the “|” symbol. This allows you to use grep to search for a pattern within the output of history, limiting the output received.

Piping history through grep

```
[user@imac user]$ history |grep ls
1  ls
3  ls -la
5  ls -la
6  ls -l song.mp3
7  ls -l letter.txt
9  ls -l letter.txt
[user@imac user]$
```

This example illustrates piping the output of history into the grep command, with the search pattern “ls.” This limits the output from the command to only show any command containing the requested search pattern (“ls”).

See the section on Using Pipes and Backticks for a more involved explanation on the use of *pipes*.

The above example demonstrates the ability to search through an extended history log for specific command fragments. However, suppose that you want to simply repeat a previously issued command, or make a slight correction to a misspelled command. BASH allows for this by setting certain keys on the keyboard to interact dynamically with the history file as special control keys.

Special Control Keys within the BASH History

You can scroll back through the most recent commands in the BASH history by hitting the “up” arrow key. Each time you press “up” it will insert the next most recent command preceding the current command line buffer. Conversely, you can press the “down” arrow key to move forward in the history from any other point. In addition, if your terminal emulation allows it, you may press “page up” and “page down” to jump from the very beginning to the very end of your complete history. Trying to move the history further than it can go in either direction will result in the shell beeping and remaining on the current buffer.

Moving through the command history in this fashion inserts each found command, exactly as it resides in the history file, into the current command line buffer. Bear in mind that you can alter a command to suit your needs by moving the cursor using the left and right arrow keys, and making whatever modifications are necessary. Note also that pressing ENTER at any place in the command will execute the complete command as it sits in the buffer without omitting any fragments, regardless of where the cursor sits when pressed.

<note>
Shortcut Controls

If your terminal emulation does not allow for you to press HOME and END to move the cursor from the far left and far right of the current command buffer, respectively, the BASH command line interprets CTRL+A and CTRL+E as replacement shortcuts. Press the CTRL and A keys simultaneously to move the

cursor to the beginning of the line, and the CTRL and E keys to move to the end of the buffer.

Another powerful control mechanism for looking up previously typed commands is called <emphasis>reverse-i-search</emphasis>. This functions similarly to how modern web browsers, such as Konqueror, will commonly guess what domain name you wish to visit by filling in the remainder of the web address as you type a recently visited URL.

To use reverse-i-search, press the CTRL and R keys simultaneously. The prompt will be replaced with a reverse-i-search prompt.

```
[user@imac user]$
(reverse-i-search)`':
```

Once at the prompt, begin typing a command, or fragment of a command, that you believe to be somewhere in your history of commands. As you type, the reverse-i-search prompt will begin to guess commands that match the input dynamically to the right of the search pattern.

Using reverse-i-search

```
[user@imac user]$
(reverse-i-search)`grep ': grep -E “[Ss]imilarlyIconcern” letter.txt
```

If upon typing your complete search pattern the correct command has not appeared, you may at that point press CTRL+R again to search further through your BASH history for the same pattern. The search results will first show the most recent commands, searching backwards in the history for the currently entered pattern upon each subsequent pressing of CTRL+R.

Once the command you are searching for appears in the reverse-i-search buffer, press ENTER to execute the command. If you wish to make modifications to the command in the buffer, press CTRL+E to move the cursor to the end of the command and close the search console.

If you are unable to find the desired command, you may press CTRL+C at any time to cancel the search and return the shell to a blank command line.

Using Tab Completion

BASH also supports another very useful feature called *tab completion*. Tab completion allows you to use the TAB key to ask the shell to guess what you are trying to type at any point in the command line.

If you press TAB in the middle of typing a command, BASH will attempt to locate a command starting with the fragment which you typed. If it can find only one

possibility, it will insert that command for you (e.g., pressing the TAB key after typing rmd). BASH will assume you are trying to complete the command rmdir, as it is the only command installed by default which begins with the letters "rmd".

If the command fragment typed is ambiguous at the point when TAB is pressed, the shell will guess as much of the command that it can and then beep to indicate that several options are available after the existing command. Pressing TAB twice sequentially with an ambiguous command fragment will cause the shell to display some reasonable guesses without changing your current command line buffer.

```
[user@imac user]$ mk
mk_cmds  mkdirhier    mkindex  mkofm    mktexlsr
mk_modmap    mkfifo   mkinodedb    mkpasswd mktexmf
mkcfm    mkfontdesc   mkmanifest   mkservdb mktexpk
mkdep    mkfontdir    mknod    mkstub   mktextfm
mkdir    mkhtmlindex  mkocp    mktemp   mkxauth
[user@imac user]$ mkd
mkdep  mkdir  mkdirhier
[user@imac user]$ mkd
```

The first use of tab completion above shows what happens when the user "user" types mk and then presses TAB twice sequentially (presumably in the hopes of finding the command he is looking for, which he knows starts with "mk"). This is a rather vague fragment, and thus returns 25 results. By adding the "d" and again pressing TAB twice, the list is narrowed down to only three commands.

<note>
Tab Completion Sanity Checking

In the event of an extremely ambiguous attempt at tab completion, the system will prompt you to confirm before displaying too large a list of possibilities. For example, typing just "m" into the command line and pressing TAB twice will result in the following:

```
[user@imac user]$ m
Display all 157 possibilities? (y or n)
```

Tab completion also applies to arguments passed to typed commands. The shell will attempt to guess filenames if you press TAB after having entered any command name. If you do not specify a path in the argument fragment, it will search within the present working directory for a filename beginning with the argument fragment. For example, you might know that you wish to change into a directory titled "Desktop" in your home directory.

Using Tab Completion with Arguments

```
[user@imac user]$ ls -l
total 3300
```



```
drwxr-xr-x    4 user user     4096 May 28 05:59 Desktop
-rwxrwxrwx    1 user user     1301 Jun 30 00:53 letter.txt
lrwxrwxrwx    1 user user    10 Jun 23 04:41 mysong.mp3 -> ./song.mp3
drwxrwxr-x    3 user user     4096 Jun 23 01:11 personal
-rwxrwxr-x    1 user user  3359554 Jun 29 22:56 song.mp3
[user@imac user]$
```

Since only one directory can be found that begins with a capital "D," you can use tab completion to save time by typing simply cd D, and pressing TAB.

```
[user@imac user]$ ls -l
total 3300
drwxr-xr-x    4 user user     4096 May 28 05:59 Desktop
-rwxrwxrwx    1 user user     1301 Jun 30 00:53 letter.txt
lrwxrwxrwx    1 user user    10 Jun 23 04:41 mysong.mp3 -> ./song.mp3
drwxrwxr-x    3 user user     4096 Jun 23 01:11 personal
-rwxrwxr-x    1 user user  3359554 Jun 29 22:56 song.mp3
[user@imac user]$ cd Desktop/
[user@imac Desktop]$
```

You need only press TAB once in this case, as the tab completion requested is non-ambiguous. BASH assumes that you wish to enter the Desktop/ directory, and upon pressing TAB, you will find the command completed for you.

While this example seems quite simple, bear in mind that this is a highly circumstantial technique! If there were a directory called "Documents" in the same user's home directory, for example, cd D would be ambiguous, and hitting TAB after typing this fragment will result in the shell beeping at you rather than finishing your command. Pressing TAB one more time will, as you might expect, cause BASH to provide you with some reasonable guesses based on what is possible with the given command fragment.

```
[user@imac user]$ ls -l
total 3304
drwxr-xr-x    4 user user     4096 Jun 30 04:50 Desktop
drwxrwxr-x    2 user user     4096 Jun 30 04:50 Documents
-rwxrwxrwx    1 user user     1301 Jun 30 00:53 letter.txt
lrwxrwxrwx    1 user user    10 Jun 23 04:41 mysong.mp3 -> ./song.mp3
drwxrwxr-x    3 user user     4096 Jun 23 01:11 personal
-rwxrwxr-x    1 user user  3359554 Jun 29 22:56 song.mp3
[user@imac user]$ cd D
Desktop    Documents
[user@imac user]$ cd D
```

The solution to being able to tab complete the above scenario is quite simple; you need only type cd De, or cd Do, and press TAB to have BASH finish your command without requesting further clarification.

While tab completion may seem a bit esoteric or overly-complicated as a means to

increase efficiency, once you begin to experiment with it, it is simple to learn and you will soon find it an irreplaceable tool that vastly decreases the time spent in typing out long commands, paths or filenames.

Output/Input Re-direction

In working at the command line, there will often be circumstances where you wish to place the output from a command-line program into a file for some other use. Conversely, there will be other times when you wish to use the contents of a file as the *input* for a program which expects data to be typed directly into the program when run.

This kind of general action is called *re-direction*, and BASH allows for this kind of activity via the special entity representations of less-than ("<"), greater-than ">" and double-greater-than (">>") symbols. By following a valid BASH command with the greater than (">") symbol and a filename, the shell is instructed to re-direct the *standard output* (sometimes called "stdout") of that command (if any) to the specified file. If the file is found, it will be overwritten. Similarly, by following a valid command with the double-greater-than (">>") symbol and a filename, the shell is instructed to *append* the re-directed output to the passed filename.

Re-directing Output with BASH

Consider that you may have a directory whose contents you wish to log in a file. This might be useful for documenting changes in a directory over time. A simple syntax for this would be ls -l > [filename], where "[filename]" is the name of the file that you wish to store the results in.

```
[user@imac user]$ ls -l > mydirectory.txt
[user@imac user]$ more mydirectory.txt
total 3304
drwxr-xr-x    4 user user     4096 Jun 30 04:50 Desktop
drwxrwxr-x    2 user user     4096 Jun 30 04:50 Documents
-rwxrwxrwx    1 user user     1301 Jun 30 00:53 letter.txt
-rw-rw-r--    1 user user    0 Jun 30 05:51 mydirectory.txt
lrwxrwxrwx    1 user user   10 Jun 23 04:41 mysong.mp3 -> ./song.mp3
drwxrwxr-x    3 user user     4096 Jun 23 01:11 personal
-rwxrwxr-x    1 user user  3359554 Jun 29 22:56 song.mp3
```

The above example demonstrates that using the ">" symbol after a valid filename causes the output not to show up on the screen, but is instead re-directed. Then, examining the filename following the ">" symbol with the more (or less) command, you can easily see that the output from ls -l is now stored in the file "mydirectory.txt."

<note>
Output Re-direction Permissions

You must, of course, be allowed to create a file in the path which you try to re-direct the output to! If your user does not have write permissions in the directory specified, or if your user does not have write permissions to overwrite an existing file, BASH will deny the re-direction.

```
[user@imac user]$
[user@imac user]$ ls -l > /root/badfilename.txt
bash: /root/badfilename.txt: Permission denied
[user@imac user]$
```

Re-directing Input with BASH

In contrast to the last example, consider that you have a program which accepts *standard input* (sometimes called "stdin") from the command line when run; Sendmail is an example of such a program. The Sendmail binary, sendmail, takes a command line argument indicating who to send an e-mail to, and then accepts input from the command line until a single "." is found on a new line.

```
[user@imac user]$ /usr/lib/sendmail admin@localhost
Hello, systems administrator.

I'm your user.

It's nice to meet you!

Sincerely,
Test P. User.
.
[user@imac user]$
```

In this case, it's not necessarily convenient to type your e-mail by hand while the sendmail binary sits running, waiting for you to complete your e-mail. It's also quite obviously not the most ideal way to enter this sort of information; in case you wish to change part of the output, you have no ability to change the data, as it is being entered manually. You could copy and paste the data from a text editor, but that too is more trouble than is necessary in this case if you have the the file locally on the machine with sendmail.

By using file input re-direction with the "<" symbol, you can instruct BASH to use a *file* in place of input from the shell. In this case, the syntax would read:

```
[user@imac user]$ /usr/lib/sendmail admin@localhost < letter.txt
[user@imac user]$
```

Passing the phrase "< letter.txt" to BASH at the end of the program indicated that the file called letter.txt should be drawn from rather than the "standard input," which defaults to the console.

Additionally, since the input stream is terminated after the file is re-directed to sendmail, you are not even required to end the file source with a period, despite the fact that sendmail would require it when accepting input from the console directly.

I/O (Input/Output) re-direction is a fundamental concept on any UNIX-like operating system, and Linux is no exception. As you become more familiar with Linux, you will find many circumstances in which it is extremely convenient to be able redirect I/O to files in place of the default "standard input" and "standard output."

Symbolic Links, Environment Variables and Aliases

Even with all of the built-in methods for increasing efficiency at the command line, some commands and system paths will remain more cumbersome than they ought to. Fortunately, we've only scratched the surface of the capabilities of the Linux command line.

This section covers the creation of links and symlinks (symbolic links), as well as the creation and management of *aliases*, and *environment variables*. These tools will expand your ability to customize your working environment, and simplify your daily tasks by helping you to more succinctly address complex commands and filenames.

Creating Hard Links

A link in Linux is a filesystem reference to a file on the hard drive. By default, each file has a single link within the filesystem, though unlike other operating systems, a single file may have several *links*. By creating a *hard link* to a file with the ln command, another reference to the same file is created, and becomes visible by all system processes. The syntax to create a *hard link* is ln [filename] [linkname], where "[filename]" is the name of the existing file that you wish to create another link for, and "[linkname]" is the name of the new link to be created.

```
[user@imac user]$ ls -l *mp3
-rwxrwxr-x    1 user user  3359576 Jun 30 07:42 song.mp3
[user@imac user]$ ln song.mp3 linkedsong.mp3
[user@imac user]$ ls -l *mp3
-rwxrwxr-x    2 user user  3359576 Jun 30 07:42 linkedsong.mp3
-rwxrwxr-x    2 user user  3359576 Jun 30 07:42 song.mp3
[user@imac user]$
```

At first glance, it might appear that linking linkedsong.mp3 from song.mp3 merely created a copy of the latter. Upon closer examination, however, you will notice a few oddities. For example, performing an analysis with du will show that while there are two files listed by ls now consisting of 3.3MB each, there is only a total of 3.5MB worth of data in the entire directory.


```
[user@imac user]$ ls -l *mp3
-rwxrwxr-x    2 user user  3359576 Jun 30 07:42 linkedsong.mp3
-rwxrwxr-x    2 user user  3359576 Jun 30 07:42 song.mp3
[user@imac user]$ du -s .
3508    .
[user@imac user]$
```

Probing further, it will become apparent that making any modification to either linkedsong.mp3 or song.mp3 will show up in both files.

```
[user@imac user]$ ls -l *mp3
-rwxrwxr-x    2 user user  3359582 Jun 30 08:03 linkedsong.mp3
-rwxrwxr-x    2 user user  3359582 Jun 30 08:03 song.mp3
[user@imac user]$
```

The only exception to this relationship between the original file and the hard-link is involved in *removing* either the link, or the original file. You may freely use the rm command to "remove" one of the files, even though in actuality you are simply removing a link, and not deleting the file or link's data from the system.

Creating Symbolic Links

In addition to being able to create *hard links*, ln has the capability to create a more flexible kind of link called a *symlink*. A *symlink* is similar to an *alias* in Mac OS™ , or a *shortcut* in Windows™ . It acts as a "forwarding address" to an actual file. The syntax to create a *symlink* is ln -s [filename] [linkname].

Creating a "symlink" with ln

```
[user@imac user]$ ls -l *mp3
-rwxrwxr-x    2 user user  3359582 Jun 30 08:03 linkedsong.mp3
-rwxrwxr-x    2 user user  3359582 Jun 30 08:03 song.mp3
[user@imac user]$ ln -s song.mp3 mysong.mp3
[user@imac user]$ ls -l *mp3
-rwxrwxr-x    2 user user  3359582 Jun 30 08:03 linkedsong.mp3
lrwxrwxrwx    1 user user     8 Jun 30 10:01 mysong.mp3 -> song.mp3
-rwxrwxr-x    2 user user  3359582 Jun 30 08:03 song.mp3
[user@imac user]$
```

By using ls -l, some differences between hard-linking and symbolic-linking are obvious. The *symlink* ("mysong.mp3") has a size of only 8k, and is shown to point to the file called song.mp3.

The most notable difference between a *hard link* and a *symlink* is that a *hard link* **must** point to an existing file. In contrast, a *symlink* may point to a completely non-existent file. The reason for this is that the *symlink* is just a "pointer" - it has no genuine knowledge of what it is pointing to, even though for most system functions

it will be treated as if it were literally the file it points to. Once again, the exception to this is deletion; using rm on a symbolic link will perform the function of deleting the link itself, not the file which the link points to.

For this reason *symlinks* are both more flexible, and more dangerous, depending on what you are doing with them! If you create a symbolic link to a commonly used library, for example, and the library name changes when it is upgraded, your link will be broken. Broken links are easy to spot, because they will be highlighted. In fact, if your terminal emulation supports color, the link itself will be white on a red background, blinking off and on, just to make sure it has your attention!

You will frequently find symbolic links used in library directories for a single library. This is done to resolve linking for a variety of version conventions. For example, in the /usr/lib/ you will frequently find symbolic links to extremely similarly named files, where only the version numbers vary in their specificity.

Examining Existing Library Symbolic Links

```
[user@imac lib]$ ls libkdeui*
-rwxr-xr-x    1 root root  931 May  3 21:18 libkdeui.la
lrwxrwxrwx    1 root root   17 May 21 06:11 libkdeui.so -> libkdeui.so.3.0.0
lrwxrwxrwx    1 root root   17 May 21 06:11 libkdeui.so.3 -> libkdeui.so.3.0.0
-rwxr-xr-x    1 root root  2824845 May  3 21:18 libkdeui.so.3.0.0
[user@imac lib]$
```

Creating and Using Aliases

Like any good command line-based environment, BASH also has support for *aliases*. An alias is essentially a stored routine in memory. It has an associated *name* by which it is run, and a *procedure* that executes when its name is invoked. The name can be any name that you wish, although you generally want to be careful not to name it after an existing system program, as the alias will take precedence!

The command to create an alias is alias. The syntax to create a new alias is alias [name]="[procedure]" where "[name]" is the name that you wish to use to invoke the alias, and "[procedure]" is the procedure that you wish to be executed. If an alias already exists with that name, it will be overwritten without confirmation.

Creating a Simple Alias with alias

```
[user@imac user]$ alias mp3s="ls *mp3"
[user@imac user]$ mp3s
linkedsong.mp3  mysong.mp3  song.mp3
[user@imac user]$
```

The above example illustrates the creation of an alias named "mp3s" that, when

run, performs the command ls *mp3.

The Life Cycle of an Alias

The lifetime of an alias extends only as long as your shell session unless you explicitly take steps to make sure the alias is re-initialized each time you login to your account.

There is a special file called .bashrc which every user has a copy of in their home directory.

```
[user@imac user]$ more .bashrc
# .bashrc

# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then    . /etc/bashrc
fi
[user@imac user]$
```

This file is processed each time you login to the system, and it is a handy place to put common aliases that you wish to be set *each time you login*. Just open .bashrc in your text editor of choice and add as many commands as you like. Be sure that each command is on its own line. Note that there is also an easier way to add aliases to this file, explained below.

Adding Aliases to .bashrc

```
[user@imac user]$ echo "alias mp3s='ls -l *mp3'" >> .bashrc
[user@imac user]$ more .bashrc
# .bashrc

# User specific aliases and functions

# Source global definitions
if [ -f /etc/bashrc ]; then    . /etc/bashrc
fi
alias mp3s='ls -l *mp3'
[user@imac user]$
```

The example above illustrates an easy shortcut to appending a line to the .bashrc file. It uses the echo command to echo the phrase "alias mp3s='ls -l *mp3' to the .bashrc file. Ordinarily echo would simply repeat the output back to the screen; however, attaching the ">>" re-direction symbol causes BASH to append the output to th .bashrc file.

As you can see, the command alias mp3s='ls -l *mp3' has been successfully appended. This means that the next time "user" logs into the system, the "mp3s"

alias will be automatically set.

Listing Current Aliases

Many aliases are set by default on most Linux systems. These can include customized versions of ls that automatically pass certain flags to the real ls command, special aliases that correspond to directory commands, and so on. They are rarely constant from system to system; on your Yellow Dog Linux machine, however, you will likely have only a few aliases configured for special ls modes, and one for which.

In order to list all currently configured aliases, just type alias and press ENTER with no further arguments.

Listing Current Aliases

```
[user@imac user]$ alias
alias l.='ls -d .[a-zA-Z]* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias mp3s='ls *mp3'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot --show-tilde'
[user@imac user]$
```

Removing Aliases

Due to the extremely circumstantial nature of aliases, it is not uncommon to want to *unset* an alias after having set it in memory. The command to perform this function is unalias, and it accepts only one argument: the alias that you wish to remove from memory. This can be an especially important command if you happen to accidentally set an alias over a command you need to regain access to. (You can, of course, always log out, and log back in to clear the aliases set in memory, but this is not always convenient in the Shell environment.)

Removing an Alias with unalias

```
[user@imac user]$ alias ls=""
[user@imac user]$ ls
[user@imac user]$ ls
[user@imac user]$ unalias ls
[user@imac user]$ ls
Desktop  Documents  letter.txt  linkedsong.mp3  mysong.mp3  personal  song.
mp3
[user@imac user]$
```

This example demonstrates our trusty “user” getting himself into yet more trouble. In the above scenario, he has accidentally set an alias over the command name ls.

As you can see, this causes the ls command to be inaccessible until the unalias command is used to release the empty alias for ls.

<note>
Aliases and You

Remember that, since Linux is a multi-user environment, your aliases will not affect any other user that is logged into the system, and vice-versa. Do not be afraid of causing problems for other users by experimenting with your shell environment!

Environment Variables

BASH, as a shell environment, is much more than just a command line interpreter. It is actually a complete programming environment, allowing you to write customized scripts that perform programmatic operations, all with built-in BASH commands and variables.

While programming with BASH is outside the scope of this document, it is important to understand (and be able to use) an element of BASH programming known as *environment variables*. Environment variables are similar to aliases, in that they consist of a pair of associated values in memory.

Environment variables, however, are more general in application than aliases. While an alias has a name which is used to invoke a particular command or set of commands, an environment variable can contain any value. It can also be interpreted by BASH directly into any existing command at any point. There are many ways to create environment variables within BASH. For our examples, we will use the export command.

Semantically, an environment variable can therefore be set with the export, command, and can be used within any other BASH command by typing the *name* of the environment variable preceded by the special “$” symbol.

Creating Environment Variables

Using export, the syntax to create a new environment variable is export [name]=”[value]” where “[name]” is the name by which you will address the variable, and “[value]” is the value that will be initially assigned to it.

```
[user@imac user]$ export anymp3=”*mp3”
[user@imac user]$
```

This example sets an environment variable in memory with the name “anymp3” and the value “*mp3.” To verify the creation of the environment variable, you can simply type export and press ENTER with no further arguments. This will display each of the currently set environment variables in memory (shown with the declare -x syntax, another means by which you may set up environment variables in BASH).

Since this is usually a very large list, you may want to pipe the output from the export command to grep in order to search for the variable you just set.

```
[user@imac user]$ export |grep anymp3
declare -x anymp3=”*mp3”
```

Using Environment Variables

Once set in memory, as mentioned earlier, these environment variables may be used in any BASH command by typing the name of the variable, preceded by the “$” variable identification symbol. The command line interpreter will automatically insert the actual value of the variable in place of where the “$name” of the variable is located in the command.

```
[user@imac user]$ ls -l $anymp3
-rwxrwxr-x    2 user user  3359582 Jun 30 08:03 linkedsong.mp3
lrwxrwxrwx    1 user user     8 Jun 30 10:01 mysong.mp3 -> song.mp3
-rwxrwxr-x    2 user user  3359582 Jun 30 08:03 song.mp3
[user@imac user]$
```

This simple example demonstrates the use of an environment variable in the middle of an ls command. Because the variable “anymp3” is set to “*mp3,” issuing the command ls -l $anymp3 translates the command in BASH to ls -l *mp3 before it is executed.

<warning>
Environment Variables within Quotes

It is vital to note that environment variable names that are placed within single-quotes (e.g., '$myvariable') will be interpreted **literally**, and not replaced with a variable value. This is, of course, not the case with double-quotes, as can be easily demonstrated:

```
[user@imac user]$ echo '$anymp3'
$anymp3
[user@imac user]$ echo “$anymp3”
*mp3
[user@imac user]$
```

Handling Processes

All running applications within Linux have an associated *process*. This process is the technical term for the set of instructions that are being run within the operating system for that command. As long as that process is running, it will have certain associated information associated with it, including a PID (process ID) by which it is addressed by the system, an associated terminal (if it was run from a terminal

session), the user who it is running as, and so on.

In many cases the *command* is synonymous with the *process*, though this is not always the case, as a single command might create multiple processes (e.g., Apache starts about a dozen child processes from the parent process, each named httpd after the command which initiated the processes).

Viewing Processes with ps

The Linux command to view system processes is ps. Typing it without any arguments will show all of your currently owned processes that are running.

```
[user@imac user]$ ps
PID TTY  TIME CMD
936 pts/3    00:00:00 bash
976 pts/3    00:00:00 ps
[user@imac user]$
```

This example illustrates that “user” has only two processes being run at the time of the ps command being called: the BASH prompt process (“bash”), and the instance of ps itself (ps, a self-reference).

ps can also accept a number of flags to alter its behavior beyond this simple example. Some common ones include:

- -e Show all processes, not just those owned by the user.
- -u Show the username associated with each process.
- -l Show output in “long” format (includes all known details).

Using ps to View All Processes

```
[user@imac user]$ ps -e
PID TTY  TIME CMD
1 ?    00:00:32 init
2 ?    00:00:19 kflushd
3 ?    00:02:17 kupdate
4 ?    00:00:00 kpiod
5 ?    00:00:54 kswapd
6 ?    00:00:00 mdrecoveryd
80 ?   00:00:00 raid1d
81 ?   00:00:41 raid1syncd
399 ?   00:02:14 syslogd
408 ?   00:00:00 klogd
426 ?   00:00:00 atd
440 ?   00:00:18 crond
454 ?   00:00:02 inetd
```

```
464 ?    00:01:26 sshd
597 ?    00:00:03 portsentry
599 ?    00:00:03 proftpd
666 ?    00:00:56 miniserv.pl
730 tty1 00:00:00 mingetty
731 tty2 00:00:00 mingetty
732 tty3 00:00:00 mingetty
733 tty4 00:00:00 mingetty
734 tty5 00:00:00 mingetty
735 tty6 00:00:00 mingetty
19540 ?    00:00:00 bash
19541 ?    00:00:48 postmaster
19542 ?    00:00:00 logger
19567 ?    00:00:20 httpsd
19568 ?    00:00:02 gcache
11350 ?    00:00:00 safe_mysqld
11368 ?    00:00:00 mysqld
11370 ?    00:00:01 mysqld
11371 ?    00:00:00 mysqld
24048 ?    00:00:01 sendmail
30391 ?    00:00:00 sshd
30392 pts/0    00:00:00 bash
30421 pts/0    00:00:00 pine
30865 ?    00:00:00 httpsd
30866 ?    00:00:00 postmaster
30871 pts/0    00:00:00 su
30872 pts/0    00:00:00 bash
30904 ?    00:00:00 httpsd
30905 ?    00:00:00 postmaster
30934 ?    00:00:00 httpsd
30935 ?    00:00:00 postmaster
30940 ?    00:00:00 httpsd
30941 ?    00:00:00 postmaster
30947 ?    00:00:00 httpsd
30948 ?    00:00:00 postmaster
30950 ?    00:00:00 httpsd
30952 ?    00:00:00 postmaster
30972 ?    00:00:00 httpsd
30973 ?    00:00:00 postmaster
30975 ?    00:00:00 httpsd
30976 ?    00:00:00 postmaster
30982 ?    00:00:00 mysqld
30989 ?    00:00:00 sendmail
30993 ?    00:00:00 sendmail
31011 pts/0    00:00:00 ps
[user@imac user]$
```

Managing the Foreground: fg and bg

Since running a command from the BASH prompt by default tends to occupy the

shell's input, you might be wondering how you could ever accumulate more than one or two processes in a single shell session. The answer lies in *job control*.

Job control allows you to treat commands as "jobs" and move them in and out of the "foreground" and "background" via the commands CTRL+Z, fg and bg. The CTRL+Z key stroke performs the function of "suspending" a running process and returning your shell session back to a BASH prompt.

When you have any number of processes suspended, you can review those processes (or "jobs") by using the Linux jobs command. This command will show you each job that you have suspended, along with an associated job number and status.

Suspending a Process with CTRL+Z

As an example, you might be using less to view a large file, and realize that you require a piece of information from another file before you can finish reviewing the first file. Rather than open another shell, or quitting less, thus losing your place in the file, you can simply press the CTRL+Z keys together to suspend the process.

```
[user@imac user]$ less letter.txt
[1]+  Stopped     less letter.txt
[user@imac user]$ jobs
[1]+  Stopped     less letter.txt
[user@imac user]$
```

The "[1]+" signifies that the less command is associated with the job number 1, and the message "Stopped" means that the program is no longer actively executing instructions to the operating system.

This example also illustrates that by typing jobs, each suspended job (in this case, just the less process) will be displayed. The format of jobs is identical to the message received upon suspension.

Once a process has been suspended and it becomes available under job control, it can be affected by either the fg foreground or bg commands. The result of these commands depends on the nature of the job.

The fg Command

The fg command instructs a stopped job, or a job which is running in the background, to begin running in the "foreground." This will completely return the shell session to the command associated with that job number, as if the job had never been suspended.

The syntax to use fg is fg [%job], where "[%job]" is the job number representing the process preceded by the percent sign (e.g., "%1").

Putting a Job in the Foreground

```
[user@imac user]$ less letter.txt
[1]+  Stopped     less -m letter.txt
[user@imac user]$ jobs
[1]+  Stopped     less -m letter.txt
[user@imac user]$ fg %1
August 30, 2000.
From the desk of Test P. User.

To whom it may concern,

This is a letter that I am writing to you with the intent to clarify
the purpose for my writing this letter. Without being too redundant,
I would like to begin by stating that this is a letter that I am writing to
you with the intent to clarify the purpose for my writing this letter.

Furthermore, this is a just letter that I am writing to you with the intent
to clarify the purpose for my writing this letter. Without being too
redundant, I would like to begin by stating that this is a letter that I am
writing to you with the intent to clarify the purpose for my writing this
letter.

Similarly, this is a also letter that I am writing to you with the intent
to clarify the purpose for my writing this letter. Without being too
redundant, I would like to begin by stating that this is a letter that I am
writing to you with the intent to clarify the purpose for my writing this
letter.

The best way to explain this is to state that this is a letter that I am
80%
```

This example shows the suspension of the less -m letter.txt command, and subsequent "foregrounding" of that process with the fg %1 command.

The bg Command

In contrast to the fg command, the bg command instructs a stopped job to attempt to begin running again in the background. This means that the command will continue to try to execute its instructions to the system without actually returning your shell session to it. While this will leave you at a shell command line, able to run any shell command, it is important to remember that the backgrounded job will continue to send its output to the shell that invoked it.

Backgrounding a process in this way assumes that the process doesn't require interactive input from the shell in order to continue running. For example, in

the previous example, requesting that the less process continue to run in the background has no clear meaning (as the program requires user input to perform any kind of operation). An example of a process that you might request to continue running in the background is downloading a large file with an application like wget, or a large file copy.

The syntax to use bg is simply bg [%job], where "[%job]" is the job number, prefixed by the percent sign (e.g., "%2").

<note>

Running a Job in the Background

This example demonstrates the use of the bg command to put a process into the background after having been suspended. Bear in mind, however, that within BASH you may append the special "&" symbol to a command in order to automatically "fork" the process into the background.

```
[user@imac user]$ wget http://www.somehost.tld/largefile.tar
--06:45:47--  http://www.somehost.tld/largefile.tar  => `largefile.tar'
Connecting to www.somehost.tld:80... connected!
HTTP request sent, awaiting response... 200 OK
Length: 23,290,807 [application/tar]

0K -> ..... ..... ..... ..... ..... [  0%]
50K -> ..... ..... ..... ..... ..... [  0%]
100K -> ..... ..... ..... ..... ..... [  0%]
150K -> ..... ..... ..... ..... ..... [  0%]
200K -> ..... ..... ..... ..... ..... [  1%]
250K -> ..... ..... ..... ..... ..... [  1%]
300K -> ..... ..... ..... ..... ..... [  1%]
350K -> ..... ..... ..... ..... ..... [  1%]
400K -> ..... ..... ..... ..... ..... [  1%]
450K -> ..... ..... ..... ..... ..... [  2%]
500K -> ..... ..... ..... ..... ..... [  2%]
550K -> ..... ..... ..... ..... ..... [  2%]
600K -> ..... ..... ..... ..... ..... [  2%]
650K -> ..... ..... ..... ..... ..... [  3%]
700K -> ..... ..... ..... ..... ..... [  3%]
750K -> ..... ..... ..... ..... ..... [  3%]
800K -> ..... ..... ..... ..... ..... [  3%]
850K -> ..... ..... ..... ..... ..... [  3%]
900K -> ..... ..... ..... ..... ..... [  4%]
950K -> ..... ..... ..... ..... ..... [  4%]
1000K -> ..... ..... ..... ..... ..... [  4%]
1050K -> ..... ..... ..... ..... ..... [  4%]
1100K -> ..... ..... ..... ..... ..... [  5%]
1150K -> ..... ..... ..... ..... ..... [  5%]
1200K -> ..... ..... ..... ..... ..... [  5%]
1250K -> ..... ..... ..... ..... ..... [  5%]
1300K -> ..... ..... ..... ..... ..... [  5%]
```

```
1350K -> ..... ..... ..... ..... ..... [  6%]
1400K -> ..... ..... ..... ..... ..... [  6%]
1450K -> ..... ..... ..... ..... ..... [  6%]
1500K -> ..... ..... ..... ..... ..... [  6%]
1550K -> ..... ..... ..... ..... ..... [  7%]
1600K -> ..... ..... ..... ..... ..... [  7%]
1650K -> ..... ..... ..... .....
[1]+  Stopped     wget http://www.somehost.tld/largefile.tar
[user@imac user]$ jobs
[1]+  Stopped     wget http://www.somehost.tld/largefile.tar
[user@imac user]$
[user@imac user]$ bg %1
[1]+ wget http://www.somehost.tld/largefile.tar &
[user@imac user]$ . ..... [  7%]
1700K -> ..... ..... ..... ..... ..... [  7%]
1750K -> ..... ..... ..... ..... ..... [  7%]
1800K -> ..... ..... ..... ..... ..... [  8%]
1850K -> ..... ..... ..... ..... ..... [  8%]
1900K -> ..... ..... ..... ..... ..... [  8%]
[user@imac user]$
```

This example demonstrates the use of a command called wget, which has been called to retrieve a removed file called "largefile.tar." 7% into the download, "user" apparently became impatient, and hit CTRL+Z to stop and suspend the download, returning him to a BASH prompt.

At this point, to continue the download, he typed bg %1 to indicate to the shell's job control that he wanted job #1 to attempt to keep running in the background. As you can see, this caused the job to re-start, but still allowed the user to remain at a command line.

<note>
Silencing Output from Background Processes

This example illustrates a common problem: the need to run a job in the background which has a lot of output that you don't necessarily want to see while it runs. The solution to totally silencing the output from an application lies in *output re-direction*, but it is a little trickier than you might think.

While you can easily re-direct "standard output" (or "stdout") with the ">" modifier (as documented in this chapter's section on Input/Output re-direction), programs will be frequently designed to send output to what is called "standard error" or "stderr," which will **not** be re-directed by the ">" symbol.

The secret to re-directing "stderr" as well as "stdout" to another location is to use the special BASH sequence "2>&1" in your command, preceding the output re-direction. This is a special system re-direction of "stderr" (represented here by the number "2") into "stdout" (represented by the number "1"). Therefore, to illustrate the same example with this trick:


```
[user@imac user]$ wget http://www.somehost.tld/largefile.tar 2>&1 >/dev/null
[1]+  Stopped     wget http://www.somehost.tld/largefile.tar 2>1 >/dev/null
[user@imac user]$ bg %1
[1]+ wget http://www.somehost.tld/largefile.tar 2>&1 >/dev/null
[user@imac user]$
[user@imac user]$
[1]+  Done    wget http://www.somehost.tld/largefile.tar 2>&1 >/dev/null
```

This illustrates "user" expertly using the BASH re-direction syntax to first send "stderr" to "stdout" and then to send "stdout" to /dev/null, a special system file that acts like a black hole, swallowing any data that is sent to it without any kind of processing.

Eventually, when finished, the system will report that the job has completed, as you can see in the line containing the status "Done" for job #1.

The Last Resort: Killing Processes with kill

In some cases you may find that certain processes go out of the normal means of control, and need to be externally reset or killed. For this reason, Linux has the kill and killall commands.

<note>
On Running Processes

The kill and killall commands do not necessarily need to be used, depending on the nature of the process you wish to cancel. If you have executed a program in a shell session and it stops responding, you frequently have the option to press the CTRL and C keys simultaneously to attempt to cancel the program.

The kill command requires only a single argument: the PID, or process ID, that you wish to affect (though remember that you may pass numerous PIDs as well, if you need to terminate multiple processes). In addition to the PID or PIDs that you wish to terminate, you may also pass kill an optional flag describing the behavior with which to affect the PIDs in question. These flags are comprehensively referred to as "signals."

Despite the obvious connotation derived from its name, kill does not necessarily stop a running process. The available signals for kill and their associated functions include:

-HUP "hangup" signal. To programs that recognize this signal, this will typically attempt to re-start and re-initialize the process, without creating a new PID.

-TERM default "terminate" signal that will terminate most processes.

-KILL more severe termination signal, to be used only in dire emergencies, as it is the least systematically "polite" way to cause a process

to stop running, and can result in lost data or instability in other processes if misused.

Killing a Runaway Process

```
[user@imac user]$ ps -e |grep mozilla
23161 ?    00:00:10 mozilla
23162 ?    00:00:00 mozilla
[user@imac user]$ kill -TERM 23161 23162
[user@imac user]$ ps -e |grep mozilla
23161 ?    00:00:11 mozilla
[user@imac user]$ kill -TERM 23161
[user@imac user]$ ps -e |grep mozilla
23161 ?    00:00:11 mozilla
[user@imac user]$ kill -KILL 23161
[user@imac user]$ ps -e |grep mozilla
[user@imac user]$
```

The above example demonstrates "user" using the ps -e command, piped through the grep command in order to determine what process IDs, or PIDs, are associated with Netscape. (This may be required if Netscape locks up on an excessively large page, for example). As you can see, the first attempt to kill the associated process IDs with kill -TERM only killed one of the requested processes. A second attempt also failed to terminate the process.

Finally, in desperation, you can see that "user" passes the less polite -KILL flag to kill in an attempt to forcibly remove the process from existence. As you can see, the -KILL flag does the job.

The killall Command

The closely related killall command exists as a means to more easily remove processes that you know by name, but not by number. While kill requires a PID, or series of PIDs, killall requires a **name** by which it locates the process to terminate.

Like kill, you can modify killall with signals to change the behavior of the program. It accepts the same signals as kill, (e.g., -HUP, -KILL) and also defaults to the -TERM signal.

Killing Processes with killall

To use the same example as the previous section on kill, you can see how much simpler it is to use killall versus kill.

```
[user@imac user]$ ps -e |grep mozilla
23161 ?    00:00:10 mozilla
```



```
23162 ?    00:00:00 mozilla
[user@imac user]$ killall -KILL mozilla
[user@imac user]$ ps -e |grep mozilla
[user@imac user]$
```

The use of ps in the preceding example illustrates the effectiveness of killall, but the use of killall precludes the need for using ps to terminate a process. In order to use killall, you need only know the name of the process, not its numeric PID.

The flip-side to this flexibility is that killall will **kill all** processes whose name matches the supplied argument. There may be some instances where you wish to kill only a single misbehaving process, in which case kill is your preferred option.

Advanced Shell Techniques

If the preceding sections have still not satisfied your desire to learn more about the power and flexibility of BASH, this section exists to initiate you into some of the more self-referential capabilities of the BASH command interpreter. This section covers consecutive command execution from a single BASH statement, as well as the use of *pipes* and *sub-commands* to arbitrarily plug the input and output from commands into one another.

Executing Consecutive Commands

There are many instances in Linux where it is beneficial to be able to execute several commands sequentially after one another, without having to spend the time typing in each command between each actual execution. BASH allows for this kind of delayed consecutive execution with the special ";" character.

By placing a semi-colon at the end of a command, you are instructing the command line interpreter to begin processing a new command after the semi-colon, as if it was being typed directly after the prompt. You may use as many semi-colons in a single BASH statement as you like. This sort of technique can be especially useful in managing services that require little to no down time, such as a web server that requires a module upgrade.

Initiating Consecutive Commands

```
[root@imac /root]# service httpd stop; mv -fv libphp4.so /etc/httpd/modules/
libphp4.so; service httpd start
Shutting down http:        [  OK  ]
libphp4.so -> /etc/httpd/modules/libphp4.so
Starting httpd:            [  OK  ]
[root@imac /root]#
```

This example demonstrates the use of semi-colons at the BASH prompt separating

three discrete BASH commands. The first command, service httpd stop, stops the httpd service. The second command, mv -fv libphp4.so /etc/httpd/modules/libphp4.so, copies a newer PHP module over the existing one in the /etc/httpd/modules directory (with the -fv flags, indicating to force the overwrite, and to do so verbosely). The third command, service httpd start, causes the httpd service to be started anew.

By carefully arranging commands in this nature, you can queue consecutive commands to be executed nearly simultaneously at the point when you press ENTER on the keyboard. Bear in mind, though, that while these commands may execute very rapidly, they still will be run by BASH in the sequence in which they are entered, and no two will be executed at the same time (so you need not worry about the speed of one command "outrunning" another, as executive precedence is respected, from left to right).

Conversely, there may be times when you wish to queue consecutive commands which take quite a long time to run. In this method, the use of the semi-colon can allow you to easily instruct several commands to execute, in order, without requiring further input from the user.

Using Pipes and Sub-commands

The ability to plug commands into one another in a variety of ways is one of BASH's strongest features. This section introduces you to some of these more esoteric methods through the use of *pipes* and *sub-commands*. Each of these utilities will open new worlds for you to explore with the power of BASH as you delve more deeply into the functionality of your Linux system.

Using Pipes

As already demonstrated periodically in examples throughout this part of the book, the BASH prompt supports the use of a pipe, by way of the special "|" character. By placing a pipe at the end of a command, you may connect together the *output* from the command to the left of the pipe with the *input* to the command on the right.

Pipes are arguably one of the most useful features implemented in the BASH command interpreter, as they allow you to exponentially increase each command's usefulness by connecting them into one another, greatly expanding the possibilities and contexts for each program's use.

What has **not** been previously covered in detail is that pipes can be used to attach any two programs, provided that their input and output are meaningfully compatible, and that pipes may be arbitrarily added on to BASH statements that have already been piped.

Using Nested Pipes

```
[user@imac user]$ ls -l /mnt/cdrom/YellowDog/ppc |grep rpm |grep lib |less
-r--r--r--    1 root root  1952674 May 16 17:21 WindowMaker-libs-0.62.1-14.ppc.
rpm
-r--r--r--    1 root root  1653902 May 16 17:22 XFree86-libs-4.0.2-6e.ppc.rpm
-r--r--r--    1 root root   313990 May 16 17:15 compat-libstdc++-2.95-2.1a.ppc.
rpm
-r--r--r--    1 root root   346451 May 16 17:15 compat-libstdc++-egcs-1.1-2.1a.
ppc.rpm
-r--r--r--    1 root root    37964 May 16 17:15 cracklib-2.7-8.ppc.rpm
-r--r--r--    1 root root   422300 May 16 17:15 cracklib-dicts-2.7-8.ppc.rpm
-r--r--r--    1 root root   322053 May 16 17:15 fnlib-0.5-4.ppc.rpm
-r--r--r--    1 root root    16841 May 16 17:15 fnlib-devel-0.5-4.ppc.rpm
-r--r--r--    1 root root   143100 May 16 17:15 glib-1.2.8-4.ppc.rpm
-r--r--r--    1 root root   134120 May 16 17:15 glib-devel-1.2.8-4.ppc.rpm
-r--r--r--    1 root root   217810 May 16 17:15 glib-gtkbeta-1.3.1b-3.ppc.rpm
-r--r--r--    1 root root   293279 May 16 17:15 glib-gtkbeta-devel-1.3.1b-3.ppc.
rpm
-r--r--r--    1 root root 14026352 May 16 17:15 glibc-2.2.1-0f.ppc.rpm
-r--r--r--    1 root root  8805551 May 16 17:15 glibc-devel-2.2.1-0f.ppc.rpm
-r--r--r--    1 root root  7600353 May 16 17:15 glibc-profile-2.2.1-0f.ppc.rpm
-r--r--r--    1 root root  1106319 May 16 17:15 gnome-libs-1.2.8-7a.ppc.rpm
-r--r--r--    1 root root  1367134 May 16 17:15 gnome-libs-devel-1.2.8-7a.ppc.
rpm
-r--r--r--    1 root root   166349 May 16 17:15 imlib-1.9.8.1-2.ppc.rpm
-r--r--r--    1 root root   248269 May 16 17:15 imlib-cfgeditor-1.9.8.1-2.ppc.rpm
-r--r--r--    1 root root   254401 May 16 17:15 imlib-devel-1.9.8.1-2.ppc.rpm
-r--r--r--    1 root root  6171761 May 16 17:16 kdelibs-2.1.2-1a.ppc.rpm
-r--r--r--    1 root root  2588634 May 16 17:16 kdelibs-devel-2.1.2-1a.ppc.rpm
-r--r--r--    1 root root   121035 May 16 17:16 kdelibs-sound-2.1.2-1a.ppc.rpm
-r--r--r--    1 root root   164596 May 16 17:16 kdelibs-sound-devel-2.1.2-1a.ppc.
rpm
-r--r--r--    1 root root   430233 May 16 17:17 krb5-libs-1.2.2-4.ppc.rpm
-r--r--r--    1 root root    56577 May 16 17:17 libPropList-0.10.1-6.ppc.rpm
-r--r--r--    1 root root    26004 May 16 17:17 libelf-0.6.4-7.ppc.rpm
-r--r--r--    1 root root   260286 May 16 17:17 libgal3-0.4.1-3a.ppc.rpm
-r--r--r--    1 root root    37509 May 16 17:17 libghttp-1.0.6-4.ppc.rpm
-r--r--r--    1 root root    18872 May 16 17:17 libghttp-devel-1.0.6-4.ppc.rpm
-r--r--r--    1 root root   102102 May 16 17:17 libglade-0.14-3.ppc.rpm
-r--r--r--    1 root root   104947 May 16 17:17 libglade-devel-0.14-3.ppc.rpm
-r--r--r--    1 root root   188604 May 16 17:17 libgtop-1.0.9-2.ppc.rpm
lines 1-33
```

This example shows the connection of the ls -l command to the grep command (with a search pattern of "rpm"), to another grep command (with a search pattern of "lib"), and finally to the less command. This command has the effect of twice filtering the output from ls -l via the grep command, and then paging that filtered output via the less command.

Using Sub-commands

While you are putting together commands at the BASH prompt, there may be times when you wish you could actually insert the output from one command directly into the command line itself, rather than connecting two programs by their output and input via a pipe. Fortunately, BASH's command line interpreter also has support for this kind of function with its use of *backticks*.

The “backtick” character (`) is the infrequently-used inverted apostrophe in the upper left-hand corner of most keyboards, sharing space with the more commonly used tilde (~). By using the back-tick, you can instruct BASH to execute a sub-command by containing it in the command within two backticks. That sub-command will then return its output directly into the command as if you had typed it yourself!

Using Backticks

```
[user@imac user]$ which db2ps
/usr/bin/db2ps
[user@imac user]$ less `which db2ps`
#! /bin/sh
jw -f docbook -b ps “$@”
/usr/bin/db2ps (END)
[user@imac user]$
```

This example demonstrates the execution of the command which db2ps, which finds the location of the executable script file called db2ps, and returns the output /usr/bin/db2ps. Since we utilized the back-tick (`) in our command, the BASH shell interprets the command less `which db2ps` as less /usr/bin/db2ps.

<note>

Alternate Sub-command Syntax

Depending on the context, it may be preferable to use the parenthetical form of a sub-command rather than using backticks. The effect is identical, but the syntax involves preceding the sub-command with the special “$” symbol, and surrounding the sub-command in *parentheses* versus *backticks*.

```
[user@imac user]$ which db2ps
/usr/bin/db2ps
[user@imac user]$ less $(which db2ps)
#! /bin/sh
jw -f docbook -b ps “$@”
/usr/bin/db2ps (END)
[user@imac user]$
```

While this syntactic difference may seem trivial, the parenthetical form of a sub-

command is somewhat easier to *nest*, if you have a sub-command that you wish to execute inside of a sub-command.

```
[user@imac user]$ which db2ps
/usr/bin/db2ps
[user@imac user]$ less $(echo $(which db2ps))
#! /bin/sh
jw -f docbook -b ps “$@”
/usr/bin/db2ps (END)
[user@imac user]$
```

This last example illustrates the use of the echo command, to repeat the output from the which command, before dropping it to the parent command less. While this is obviously a totally unnecessary addition, it does illustrate the flexibility of nesting parenthetical sub-commands.

Mount Points and External Hardware

In Linux, removable media and drives (eg: Zip, USB digital camera) are not necessarily automatically available you when the disk is inserted. In order for the operating system to have access to some removable media, you may need to *mount* them to a specified *point* in the filesystem.

Mounting a drive means that you ask the filesystem to map the location of the physical drive to a known directory, or *mount point* in order to access its contents. Your Linux hard disk(s) with associated partitions (boot, swap, root) are automatically mounted at startup. CD-ROMs, removable media such as Zip disks, or USB digital cameras, or non-Linux partitions on your existing drives (discussed in Chapter 1) may require manual mounting.

CD-ROM vs Other Removable Media

You should notice a CD icon for your CD-ROM sitting on the Desktop. This is a special link, unlike normal file or directory links as they point to literal system devices. The CD is removable and therefor must be mounted each time it is inserted.

When you right-click on the CD icon, you will find a new option in its context-menu “Mount.” Click to activate this function, and you will cause the system to search that device for recognized media. If the system finds a valid source, it will mount that particular media to its default location, and make it accessible through its icon.

However, if you insert a Zip disk, for instance, it may not automatically be referenced on your desktop. You will then need to create a mount point in order to mount and access the contents of this drive.

<tip>
Creating a Mount Point

As root, you should first determine what mount points are available to you by glancing inside /mnt/:

```
[root@imac user]$ ls /mnt/
[root@imac user]$ cdrom
```

If you desire, for instance, to mount a Mac OS™ partition, you need to create a directory to which you will mount the HFS partition:

```
[root@imac user]$ mkdir /mnt/macos
[root@imac user]$ mount /dev/hda8 /mnt/macos -thfs
```

Where “hda8” is just an example and “-thfs” in this particular case tells the filesystem to look for the ‘t’ype “hfs”, as in Mac OS™ . The filesystem then knows how to navigate, read and write to this mounted partition.

You may now proceed to treat /mnt/macos as you would any Linux directory with the understanding that if you do not unmount this mount point before launching MOL, you will not be able to write to this Mac OS™ partition as the Linux filesystem has it locked (only one active filesystem may have write access to a single partition at any given time, or data would be overwritten).

```
[root@imac user]$ cd /
[root@imac user]$ umount /mnt/macos
```

It is important to note that if you are mounting an ext2 or ext3 formatted partition, it is not necessary to denote the *type*.

Determining the Mount Point of an External Media

If you are attaching a USB digital camera, for instance, and cannot immediately determine where you should mount from, you may study the logs as you connect and disconnect the device.

To do this, login to a shell as root and:

```
[root@imac user]$ tail -f /var/log/messages
```

Press ENTER a few times to create some blank space (for clarity of reading the screen) and proceed to attach the camera. Note the */dev/xxx* (where “xxx” may be *sda1*, for instance) reference that scrolls by. This is where you want to mount from to access your camera.

Cancel the *tail* function by pressing *CTRL-C*.

Proceed to create a mount point in /mnt/ and then connect the two in order to copy the files from your camera to your drive.

HFS+ Utilities

Mac OS™ partitions are typically either of the partition type *HFS*, or *HFS+*. While Yellow Dog Linux has built-in support for the HFS partition type (as type “hfs”), it does not yet have total support for HFS+ partitions. For this reason there is a package included with Yellow Dog Linux called hfsplusutils. This package includes several free, portable utilities that allow you to read data from HFS+ volumes.

These commands include:

hpls	The HFS+ ls (“list files”) utility.
hpcd	The HFS+ cd (“change directory”) utility.
hppwd	The HFS+ pwd (“present working directory”) utility.
hpcopy	The HFS+ cp (“copy file”) utility.
hpmount	The HFS+ mount (“mount partition”) utility.
hpumount	The HFS+ umount (“unmount partition”) utility.

If you do not have these commands available, you may need to install the “hfsplusutils” RPM (see the section on using apt-get).

Additional Shell Features

Konsole vs Virtual Terminals

As you may know, within KDE's *Konsole* application, you can have many *virtual shells* within a single window. This is based on a much older convention for virtual shells that is accessible within Linux outside of the GUI. When physically located at a Linux terminal, and using a genuine Linux shell (not within a graphical application), you may press and hold CTRL-ALT and then press any one of F1 through F6 keys at the same time to switch between six *virtual consoles (terminals)*.

<note>
Switching from X to a Virtual Terminal

The X Window System itself technically occupies the virtual terminal at F7 or F8. If at any point you find yourself wanting to drop to a complete Linux command line from within X, without shutting down X, you can press CTRL+ALT and any of the F1 through F6 keys to drop to that virtual terminal.

You may return to the GUI simply by pressing ALT+F7.

It is often simpler and faster, with hands already poised on the keyboard, to launch a graphical application from Konsole (or the “Run Command” tool which is launched via OPTION-F2) than to grab the mouse and navigate through the K Menu.

Launching an Application from Konsole

Mast of the Linux graphical applications, by default, are capable of being launched from the shell. This is done simply by entering the first few letters of the application and pressing TAB to complete the application name, ensuring it is in your current path. Press ENTER and the application will launch.

Launching an Application with Associated Files from Konsole

To both launch the application (ee, in this example, and associated files (family photos, all of the JPEG format):

```
[user@imac user]$ cd family_photos/
[user@imac user]$ ee *.jpg
```


Chapter 7. The Power of the Shell

Chapter 7. The Power of the Shell

The Basics of System Administration

Access Restrictions

Most administrative tasks covered in this section (and in general) require that the user initiating the command have <emphasis>root access</emphasis>. This means that you must have the ability to either log into the system as the “root” user, or at least have the password required to su from your normal user to root.

Switching to root

The command to switch from your current login to another user is su. When issued without a username, su assumes that you wish to switch to the root user. There are several optional parameters, but the most important flag that you can pass to su is the login flag, represented by -l, or abbreviated as just -.

Passing the login flag instructs the shell to fully log you in as the user you wish to switch to. If you do not pass it, certain system files (e.g., .bashrc, .bash_profile) are not processed when you switch your user id. This can be a problem, as frequently your environment will not be fully configured to behave as if you had genuinely logged in as the switched user. The most obvious example of this is the $PATH environment variable, which is checked whenever you attempt to execute a command. The root user has a different set of paths in its $PATH, as it has access to more restricted programs.

Here is the most concise example of how to switch from a normal user to the root user:

```
imac login: admin
Password:
Last login: Fri Jun 22 06:46:07 from terrasoft.com
[admin@imac admin]$
```

Use the su command with the -l flag (or abbreviated as “-”) to switch to root.

```
[admin@imac admin]$ su -
Password:
[root@imac /root]#
```

Congrats! You have now switched to the *root user*.

User Management

As you come to understand your Linux system as a multi-user environment, you will naturally want to have the ability to easily add and remove users from your

system, and control the degrees of their access to the system.

This section covers the addition, restriction and removal of users from the command line.

Adding users

The basic syntax for adding a user in Linux is quite simple. The command to use is adduser. As a basic example, we'll add a new user named “foo.”

Adding a New User

Open a shell, and switch to the root user. (Only the root user has the rights to add system users.) Once you are logged in as root, type:

```
[root@imac /root]# adduser foo
```

If another user with the name you have passed already exists, you will be prompted with the message:

```
adduser: user foo exists
```

Otherwise, the command will succeed silently (i.e., it will return you to a blank command prompt). The user has been added to the system, and should have a home path in /home/foo.

To set the password for the new user, you use the passwd command, passing it the username of the new user as the only argument. You will be prompted to enter the new user's password (twice, for verification).

```
[root@imac /root]# passwd foo
Changing password for user foo
New UNIX password:
Retype new UNIX password:
```

As always, you should choose passwords on your Linux system with care. It is not a good idea to choose a word that is based on a dictionary word, or one which is less than six characters long. Don't be surprised if you see something like the following when first entering a new password:

```
BAD PASSWORD: it's WAY too short
```

After you have entered a suitable password, twice, you should see the following message:

```
passwd: all authentication tokens updated successfully
[root@imac /root]#
```

This signifies that the user's password is set, and the user may now log in to the system with the password assigned to it.
This is just a simple example of the adduser command, as there are numerous options that can be passed to it to alter the characteristics of the added user (e.g., the expiration date of the user, their home directory, their system UID, etc). For a complete list of the options available for this command, type man adduser.

<note>
useradd **versus** adduser

As a point of interest to system administrators, the **actual** command invoked when calling adduser is called useradd. The former is merely a symlink (symbolic link) to the latter. While the adduser command is more intuitively named, it may not exist on other Linux systems, though they probably have the useradd command.

Restricting users

Sometimes you may wish to have a system user who is restricted from actually being able to login to the system. This may be a mail-only user (who retrieves their e-mail through a POP3 or IMAP capable e-mail program), or possibly an FTP-only user.

Regardless of the reason for the restriction, the command used to restrict an existing system user from being able to access a shell upon login is chsh. This is short for "change shell." In order to disable a user's login you will actually use the chsh command to change their login shell program to /bin/false, an invalid shell, thus disabling their ability to login.

To use chsh, open a shell and switch to the root user with su. The syntax for this operation is chsh [username], where "[username]" is the user whose shell access you wish to restrict. When prompted for the new shell program, type /bin/false, and press ENTER. The following is a simple example, following from the last, in revoking our new user "foo":

```
[root@imac /root]# chsh foo
Changing shell for foo.
New shell [/bin/bash]: /bin/false
Warning: "/bin/false" is not listed in /etc/shells
Shell changed.
[root@imac /root]#
```

The "Warning" provided simply means that /bin/false is not a valid shell. In this case, this is the intended outcome of using the chsh command, and it can be ignored. When you see the words "Shell changed" the command has finished successfully.

Verifying Shell Restriction

You can verify the restriction of this user's shell using the finger command. Just type in "finger foo" to view the current setting of that user's shell.

```
[root@imac /root]# finger foo
Login: foo      Name: (null)
Directory: /home/foo    Shell: /bin/false
Never logged in.
No mail.
No Plan.
[root@imac /root]#
```

<note>
Restricting Users Automatically

Knowing how to restrict an existing user is useful, but what if you wish to add a newly restricted user? You can do so by passing the -s flag with the argument /bin/false to the adduser command discussed in the last section. For example:

```
[root@imac /root]# useradd -s /bin/false restricteduser
[root@imac /root]# finger restricteduser
Login: restricteduser       Name: (null)
Directory: /home/restricteduser     Shell: /bin/false
Never logged in.
No mail.
No Plan.
```

As you can see, the -s /bin/false arguments instruct the new user to be added with the shell set to /bin/false.

Deleting users

The command to delete users in Linux is userdel. As with useradd (i.e., adduser), only root has the rights to remove a user from the system.

To remove a user from the system, open a shell, and switch to root with su. The syntax `is userdel [username], where "[username]" is the name of the user you wish to remove from the system. This is a permanent function, though users can always be re-added in the future. You will not be prompted for any kind of confirmation, so b**e certain of what you do** before you enter the command!

The only significant flag available to userdel is the -r flag. This flag causes userdel to permanently remove the contents of the deleted user's home directory. (By default userdel leaves these intact, unless explicitly instructed to remove with the aforementioned flag.)

Managing Ownership and Permissions

In any serious multi-user environment you will find specific rights and restrictions that are applied to users, and their ability to affect files and directories on the system. Linux's way of handling these rights can be summed up in two major concepts: *ownership*, and *permissions*.

Understanding Ownership and Permissions

In Linux, any file (or directory) has both a *user owner* and a *group owner*. The *user* who owns the file is frequently the user who created the file, though this is not always the case, as ownership can be transferred. The *group* which owns the file is frequently the group of the user who owns the file as well, though this is highly variable and dependent on the file.

As a result of this dual-ownership, there are three possible relationships between a user and a file: “user-owner,” “group-owner,” or “other.” These three relationships are the foundation for understanding *permissions*.

Interpreting Ownership and Permissions

The simplest way to identify a file's permissions is to use ls with the long -l output flag. You will see the permissions string in the far left column of the subsequent output for each file. The permissions are represented by a 10-digit string of characters. Just to the right of this string you will see a digit, followed by the *user* and the *group* who own the file in question.

Here is an example of how to interpret these codes:

Type ls -l and press ENTER in the path whose files you wish to examine.

```
[user@imac user]$ ls -l
total 8
drwxr-xr-x    4 user user     4096 May 28 05:59 Desktop
-rw-rw-r--    1 user user     1593 Jun 23 04:40 letter.txt
lrwxrwxrwx    1 user user   10 Jun 23 04:41 mysong.mp3 -> ./song.mp3
drwxrwxr-x    3 user user     4096 Jun 23 01:11 personal
-rw-rw-r--    1 user user  5071935 Jun 23 04:40 song.mp3
```

The first character in the permissions string actually represents the *type* of the file. This is usually d for directory, l for link, or the dash symbol (-) for a normal file. In this example, you can see that there are two directories, two normal files, and one symbolic link.

Following the type are three groupings of three characters each. The first grouping represents the permissions set for the *user-owner*, the second grouping represents the permissions set for the *group-owner*, and the final grouping represents the

permissions set for *all other users*.

Now, to interpret each *grouping*, we will look at the letters one at a time. The first character in the permissions group is typically either r or the dash symbol (-). This is the flag for *read rights*; when this is set, read-capability is enabled, and when set to a dash (-), read-capability is disabled.

Immediately to the right of the read flag is a character which is typically w or -. The w is the flag for *write rights;* when set to “w,” write-capability is enabled, and when set to a dash (“-”), write-capability is disabled. Finally, to the right of the write flag is a character which is typically “x” or dash (“-”). This is the *execute rights* flag; when set to “x,” execute-capability is enabled, and when set to a dash, it is disabled. The executable flag generally only refers to *programs*, though on directories it refers to the access to actually cd into the affected directory.

In short, any flag which is set to the (- character signifies that the right in question is restricted, or unset. If set, it will reflect the appropriate right which it represents, usually either r, w, or x.

Understanding this scheme, we can look again at our output from ls to determine who is allowed to perform what operations on the files in our user's home directory. This time we'll add the all “-a” flag in order to also view the permissions set on the current working directory, and parent directory (as well as a few hidden files).

```
[user@imac user]$ ls -la
total 76
drwx------    8 user user     4096 Jun 23 04:41 .
drwxr-xr-x    8 root root     4096 Jun 22 09:20 ..
-rw-------    1 user user     8 May 28 06:17 .bash_history
-rw-r--r--    1 user user    24 May 28 05:59 .bash_logout
-rw-r--r--    1 user user   230 May 28 05:59 .bash_profile
-rw-r--r--    1 user user   124 May 28 05:59 .bashrc
drwx------    2 user root     4096 May 28 05:59 .xauth
drwxr-xr-x    4 user user     4096 May 28 05:59 Desktop
-rw-rw-r--    1 user user     1593 Jun 23 04:40 letter.txt
lrwxrwxrwx    1 user user    10 Jun 23 04:41 mysong.mp3 -> ./song.mp3
drwxrwxr-x    3 user user     4096 Jun 23 01:11 personal
-rw-rw-r--    1 user user  5071935 Jun 23 04:40 song.mp3
[user@imac user]$
```

By looking at the permission string of the present working directory (the first line of the given output above, symbolically represented as “.”), we can derive the following:

1. “.” is a directory (**d**rwx --- ---).
2. “.” is readable, write-able, and executable by the user who owns

it. (drwx --- ---).

3. "." is NOT readable, write-able or executable by the group who owns it. (drwx --- ---).

4. "." is NOT readable, write-able or executable by any other user. (drwx --- ---).

5. "." is user-owned by the user "user" (8 user user).

6. "." is group-owned by the group "user" (8 user user).

This means that only the genuine user "user" may affect this directory (not even members of the group "user"). Additionally, no other user may delete any file *within* this directory, regardless of that file's permissions, because it resides within a permissions-protected path. For example, even though the "other" write-able flag is set on the "mysong.mp3" file, no other user can remove it from /home/user because the "group" and "other" write-able flags are NOT set for the directory which it is within.

Sticky Situations

There are a couple of exceptions to the r, w, and x flags in the read-write-execute permissions blocks. Specifically, in addition to the "set" and "unset" switches, there are the "sticky" flags that can be set for the user and group *execute* permissions. These flags, when set, cause an application to run either as the *user* that owns it, or the *group*, depending on whether or not it is set to the user permissions, or group permissions.

Setting this flag can obviously be a security risk, depending on what the program itself does. For example, setting the sticky flag on the rm command's user executable permissions would cause it to run as root (because root owns /bin/rm), thereby allowing **any user** to delete **any file**. This is obviously not good security!

To determine whether or not a sticky flag is set, look at the user and group permission blocks, and check the executable flag. If it is set to "s" in the user block (rather than "-" or "x"), that means the program in question is set to run as the *user* who owns the program. If it is set to "S" in the group block (rather than - or x), that means the program is set to run as the *group* which owns the program.

Setting Permissions with chmod

Now that you have an understanding of *what* permissions are, you are probably interested in knowing *how* to set them. The Linux command in question is chmod, and there are two common ways to use it. These are known as numeric mode and symbolic mode.

Using chmod, by the Numbers

Every possible combination of the r, w, and x modes has a corresponding number between 0 and 7, making a total of eight modes (2^3, for the math majors in the audience). For example, no rights ("---") is represented by 0, all rights ("rwx") is represented by 7, read and execute writes ("r-x") are represented by 5, and so on.

Numeric Permissions

0 No Rights
1 Execute Rights
2 Write Rights
3 Write and Execute Rights
4 Read Rights
5 Read and Execute Rights
6 Read and Write Rights
7 All Rights (Read, Write, Execute)

Using these numbers, you can create a complete representation of a file's *user*, *group*, and *other* permission block. For example:

"777" represents permissions which are set to read, write, and execute for ALL users.

"755" represents permissions which are set to read, write, and execute for the user who owns the file, read and execute for any other user.

"750" represents permissions which are set to read, write and execute for the user who owns the file, read and execute rights for the group who owns the file, and no rights for any other user.

The syntax to apply a numeric permissions mode to a file with chmod is chmod [mode] [filename], where [mode] is the numeric mode you wish to apply, and [filename] is the file or files that you wish to apply the new permissions to.

```
[user@imac user]$ ls -l letter.txt
-rw-rw-r--    1 user user     1593 Jun 23 04:40 letter.txt
[user@imac user]$ chmod 640 letter.txt
[user@imac user]$ ls -l letter.txt
-rw-r-----    1 user user     1593 Jun 23 04:40 letter.txt
[user@imac user]$
```

In this example, the user logged in has changed the file letter.txt from having permissions 664 to 640, revoking all rights to other users (64**0**), limiting group users from being able to write to the file (6**4**0), and leaving the user rights

unchanged at *read* and *write* (**6**40).

Using chmod in Symbolic Mode

If the preceding section on setting modes numerically seems counter-intuitive, there is another way to use chmod in its *symbolic mode*. The symbolic mode allows you to pass symbolic letters to affect one or more group's permissions, rather than setting the literal numeric mode.

The symbolic argument format is made up of two components: the first dictates which users' access will be changed, and the second dictates what the change itself is. To address which users' access will be changed, you may specify one or more of the following letters:

- u "user" permissions
- g "group" permissions
- o "other" permissions
- a all permissions

Once you have specified the target of the modification for chmod with the preceding codes, you must of course instruct chmod as to what modification to make. The modifiers are the plus sign ("+") which enables rights, the dash ("-") which disables rights, and the equals sign ("=") which sets the arguments given as the only rights. The modifier must be followed by the permissions which you wish to enable, disable, or set exclusively. For these rights, as you might expect, you may specify one or more of the following letters:

- r"read" permissions.
- w "write" permissions.
- x "execute" permissions.
- s "sticky" permissions (see previous note on **Sticky Situations**)

An example:

```
[user@imac user]$ ls -l letter.txt
-rwxr-----    1 user user     1593 Jun 23 04:40 letter.txt
[user@imac user]$ chmod a+rw letter.txt
[user@imac user]$ ls -l letter.txt
-rwxrw-rw-    1 user user     1593 Jun 23 04:40 letter.txt
[user@imac user]$
```

In this example, our user named "user" has decided that anyone on the system should have the right to read and write to his text file, called letter.txt. By specifying "a" as the first component in the symbolic mode, "user" has specified

that this chmod modification should affect *all* users' permissions. By following the "a" with +rw, "user" has specified that the change to the permissions themselves should be to *enable* (a**+**rw) "read" and "write" access (a**+rw**).

Now if "user" had decided that this was a rash decision, he or she might take the following action:

```
[user@imac user]$ ls -l letter.txt
-rwxrw-rw-    1 user user     1593 Jun 23 04:40 letter.txt
[user@imac user]$ chmod go-w letter.txt
[user@imac user]$ ls -l letter.txt
-rwxr--r--    1 user user     1593 Jun 23 04:40 letter.txt
[user@imac user]$
```

By specifying "go" as the first component of the symbolic mode, "user" has specified that this chmod modification should affect both "group" owners, as well as "other" users, and by specifying "-w" as the permissions change, "user" has specified that chmod should *disable* (go**-**w) "write" permissions (go**-w**) for those users.

Changing Ownership with chown

Knowing how to set permissions on files can be a moot point if you do not also know how to set ownership on files. Ownership is, of course, what makes the "user" and "group" rights meaningful; it is what dictates who the *user* and *group* permissions practically affect. In Linux, the command to change ownership of a file is chown, short for "change owner."

The syntax to chown is similar to chmod, in that the first argument describes the modification to be made, and the second argument is the file (or files) to be modified. The syntax for chown's modification, however, is far simpler than the code-like arguments of chmod.

The only caveat to using chown is that only root has the right to change the *user-ownership* of a file to another user, while *group-ownership* can be set by either root or the user-owner of the file.

Changing User Ownership

In its simplest form, the modification argument can consist solely of the *username* of the user that you wish to be assigned as the *user-owner* for chown's targeted file or files. As a simple example:

Changing User Ownership with chown

```
[user@imac user]$ su -
```

```
Password:
[root@imac /root]# cd ~user
[root@imac user]# ls -l letter.txt
-rwxr--r--    1 user user     1593 Jun 23 04:40 letter.txt
[root@imac user]# chown admin letter.txt
[root@imac user]# ls -l letter.txt
-rwxr--r--    1 admin    user     1593 Jun 23 04:40 letter.txt
[root@imac user]#
```

This example shows the “user” user switching to the root user (as only “root” may change user-ownership of files), changing to the home directory of “user” (~user, which is the same as /home/user), and changing the ownership of letter.txt to the user named “admin.” As you can see, after passing “admin” as the user who should own the file to chown, the targeted file (letter.txt) now belongs to the user “admin.”

In addition, as stated previously, chown can be used to change the *group-ownership* of a file. To change both a user-ownership and group-ownership at the same time, you may separate them with a “.” in the arguments list. To change just the group-ownership, simply omit the username by beginning the first argument with a period (“.”) as in this example:

Changing Group Ownership with chown

```
[admin@imac admin]$ cd ~user
[admin@imac user]$ ls -l letter.txt
-rwxr--r--    1 admin    user     1593 Jun 23 04:40 letter.txt
[admin@imac user]$ chown .admin letter.txt
[admin@imac user]$ ls -l letter.txt
-rwxr--r--    1 admin    admin    1593 Jun 23 04:40 letter.txt
[admin@imac user]$
```

This example shows the “admin” user changing to the “user” user's home directory (~user, which is the same as /home/user), and changing the group-ownership on letter.txt to the group “admin.” This use of chown is allowed, because the user “admin” owns the file, and belongs to the “admin” group. (A user may only change files they own to belong to groups which they are a member of.)

Assuming that the “admin” user also belongs to the “user” group, the following syntax would also be just as valid to explicitly set both the user and group settings in a single command:

```
[admin@imac user]$ ls -l letter.txt
-rwxr--r--    1 admin    admin    1593 Jun 23 04:40 letter.txt
[admin@imac user]$ chown admin.user letter.txt
[admin@imac user]$ ls -l letter.txt
-rwxr--r--    1 admin    user     1593 Jun 23 04:40 letter.txt
[admin@imac user]$
```


<note>
Knowing Which Groups You Belong To

If you are unsure of what groups you have been assigned to, just type the command <emphasis>groups</emphasis> and press ENTER.

```
[admin@imac admin]$ groups
admin user
[admin@imac admin]$
```

This program will output each of the groups that you are currently registered to. If you have been recently assigned to a new group, you may have to log out and log back in for the new group settings to take effect.

Furthermore, as the root user, you may specify a username as an argument to the *groups* command in order to view what groups another user belongs to.

```
[root@imac /root]# groups admin
admin : admin postgres apache
[root@imac /root]#
```

Group Management

The file /etc/group manages the details of groups and their members. This file is only alterable by root. If you wish to view which groups each of the system users are assigned to, login and switch to the root user and type less /etc/group.

```
[user@imac user]$
[user@imac user]$ su -
Password:
[root@imac /root]# less /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon
tty:x:5:
disk:x:6:root
lp:x:7:daemon,lp
mem:x:8:
kmem:x:9:
wheel:x:10:root
mail:x:12:mail
news:x:13:news
uucp:x:14:uucp
man:x:15:
games:x:20:
gopher:x:30:
dip:x:40:
ftp:x:50:
```

```
nobody:x:99:
users:x:100:
slocate:x:21:
floppy:x:19:
utmp:x:22:
rpc:x:32:
mailnull:x:47:
rpcuser:x:29:
xfs:x:43:
gdm:x:42:
lx:x:500:
admin:x:501:user
postgres:x:26:
apache:x:48:
user:x:502:admin
abrookins:x:503:
foo:x:504:
[root@imac /root]#
```

In this example, the only users who have been added explicitly to groups other than their own are “admin” and “user,” who have both been added to one another's groups. This means that as a member of the “admin” group, the *“group” permissions* on “admin” group-owned files will apply to “user,” instead of the default *“other” permissions*.

If you wish to assign a user to a set of supplementary groups other than their *initial group* (which is usually either the group with the same name as the user), you may do so with the usermod command. The flag to pass to usermod to assign supplementary groups to a user is -G. Follow the flag with a list of groups, separated by commas, that you wish the new user to belong to. For example, if you want to add the user “admin” to the “apache” and “postgres” groups, log in, switch to the root user, and use the syntax: usermod -G apache,postgres admin.

Assigning Users to Groups with usermod

```
[user@imac user]$
[user@imac user]$ su -
Password:
[root@imac /root]# usermod -G apache,postgres admin
[root@imac /root]# less /etc/group
root:x:0:root
bin:x:1:root,bin,daemon
daemon:x:2:root,bin,daemon
sys:x:3:root,bin,adm
adm:x:4:root,adm,daemon
tty:x:5:
disk:x:6:root
lp:x:7:daemon,lp
mem:x:8:
```



```
kmem:x:9:
wheel:x:10:root
mail:x:12:mail
news:x:13:news
uucp:x:14:uucp
man:x:15:
games:x:20:
gopher:x:30:
dip:x:40:
ftp:x:50:
nobody:x:99:
users:x:100:
slocate:x:21:
floppy:x:19:
utmp:x:22:
rpc:x:32:
mailnull:x:47:
rpcuser:x:29:
xfs:x:43:
gdm:x:42:
lx:x:500:
admin:x:501:user
postgres:x:26:admin
apache:x:48:admin
user:x:502:
abrookins:x:503:
foo:x:504:
[root@imac /root]#
```

As you can see, the user “admin” is now listed to the right of the groups “postgres” and “apache,” signifying that the command successfully completed.

<warning>
Caution Using usermod **to Add Users to Groups**

When using usermod with the “-G” flag, it is important to remember that the list of groups you pass the command will be the <emphasis role=”strong”>only</emphasis> groups which the passed user will belong to once the command has successfully run. The user will be removed from any group that is not specified after the “-G” flag. Therefore, if you intend to add a user to a new group, or set of groups, be sure you also specify any existing groups for that user (as root, you can easily check this by typing groups [username], where “[username]” is the name of the user you wish to look up).

The alternative to using usermod to add a user to a group is to manually add the user to the comma-separated lists in /etc/group. This can be done in any standard text editor, though it is vital that you take care when editing this file, as it affects some very fundamental operations of the operating system and its permissions scheme. If, in an emergency, you have mangled your /etc/group file, remember that there is a system utility called grpck that is used as an analytical tool to repair your

group file.

Service Management

In a system such as Linux, which has few truly authoritative guides, it can frequently be difficult to keep track of the myriad ways that have evolved in order to start and stop your installed services and applications. Some packages may install their binaries in paths such as /usr/bin/ and /usr/bin/, others may keep their utilities within their own sub-directory in /usr/local, /var/local, or /opt. Still others may install their utilities into someplace you might not even know about!

While it can sometimes be overwhelming, the freedom to experiment and innovate that Linux provides often results in competing modes of thought co-existing on a single system. Ultimately, with any set of competing ideas, the best ones will prevail, even in a "power-distributed" system such as Linux.

That said, it should be comforting to note that the above discussion addresses primarily the installation of new software from scratch. Knowing where to go to start, stop or check on a service should never be so confusing with software provided from a single vendor, such as the packages installed with your Yellow Dog Linux system. Therefore, this section covers the use of two vital commands to system administration: service and chkconfig.

The service Command

The service command will be a Yellow Dog user's best friend when it comes to managing your Linux system services. It streamlines the tedious process of locating the appropriate script files, and allows you to start, stop and query for the status of installed services from any path on the system with a single, universal command.

<note>
For the Technically Minded

For those interested, the service command is, in fact, a wrapper to many existing scripts, which are usually copied at the time the software services are first installed. These services typically store their own startup scripts in the /etc/rc.d/init.d/ directory. If you wish to examine the individual scripts in more detail, you may change to this directory and view them in a program such as grep (or in your editor of choice).

You must be logged in as the root user in order to use service. Switch to the root user with su, if you need to. Remember to pass the - argument to su to completely login as root (i.e., with the appropriate environment variables to enable all root commands).

```
[user@imac user]$ su -
```



```
Password:
[root@imac /root]#
```

Once fully logged in as root, you may use the service command by typing service [service_name] [command] where "[service_name]" is the name of the service which you wish to affect, and "[command]" is the command you wish to issue to the service in question.

Some common services include: httpd (Apache), postmaster (PostgreSQL), and sendmail (SMTP Sendmail Daemon). In order to view the commands available for any service, you may type service [service_name] (where "[service_name]" is the name of the service), omitting the command. This will display the potential commands for that service.

Determining service commands

```
[root@imac /root]# service httpd
Usage: /etc/init.d/httpd {start|stop|restart|reload|condrestart|status}
```

The valid commands for a service vary from service to service, and will not always be consistent in their output. Some common commands to pass to a service are start, stop, restart and status.

As an example, if you are uncertain as to the status of the Apache webserver, you can login, switch to the root user, and issue the service command, with the arguments "httpd" (the name of the Apache service), and "status" to display the status.

Querying status with service

```
[root@imac /root]# service httpd status
httpd is stopped
[root@imac /root]#
```

As you can see, this example is run while the webserver is not running. Its status is therefore reported as "stopped."

Assuming that you want the webserver to be running (for example, if your business depends on it!), you would then want to issue the service command with the arguments "httpd" for the service name, and "start" for the command.

Starting services with service

```
[root@imac /root]# service httpd start
Starting httpd:          [  OK  ]
[root@imac /root]#
```

The “OK” following the command signifies the success of the command. To verify the successful start-up you may call service once more with the same service name and “status” as the command.

Verifying status with service

```
[root@imac /root]# service httpd status
httpd (pid 3755 3754 3753 3752 3751 3750 3749 3748 3745) is running...
[root@imac /root]#
```

Some scripts will return extra information after being successfully queried by service for their status. In this case, the script returned the PID (process ID) of each of the httpd server processes currently running on the system.

Conditions that might cause service to fail include the possibility that the server may already be running, or a misconfiguration in a required configuration file (e.g., /etc/httpd/conf/httpd.conf in the above case).

Querying All Services

If you are unsure of what services are installed on your system, there is one other command line option for service. With the --status-all option, you can instruct the service command to return the status of all installed services that it can find on your system. For example:

Checking all Services with --status-al

```
[root@imac /root]# service --status-all
anacron is stopped
atalkd is stopped
atd (pid 362) is running...
crond (pid 407) is running...
gpm (pid 392) is running...
httpd is stopped
identd (pid 350 349 348 347 343) is running...
Chain input (policy ACCEPT):
Chain forward (policy ACCEPT):
Chain output (policy ACCEPT):
lpd is stopped
Configured devices:
lo eth0
Currently active devices:
eth0 lo
lockd is stopped
rpc.statd is stopped
portmap is stopped
postmaster (pid 2093) is running...
```



```
adsl-status: Link is down (can't read pppoe PID file /var/run/adsl.pid.pppoe)
The random data source exists
rwhod is stopped
sendmail is stopped
sshd (pid 1858) is running...
syslogd (pid 307) is running...
klogd (pid 317) is running...
xfs (pid 454) is running...
xinetd (pid 377) is running...
ypbind is stopped
[root@imac /root]#
```

This example illustrates sample output from installed services, and will vary depending on how you have configured your system services with chkconfig.

Runlevels, init **and the** chkconfig **Command**

Linux inherits from traditional UNIX a systematic way of grouping settings referred to as *runlevels*. A *runlevel* is a somewhat arbitrary software configuration that allows you to be able to define various levels within Linux. Having these levels allows you to easily group together sets of applications and services that should be started and stopped at each sequential level.

By default, most of your services and system settings are already set up to work within standard *runlevels*. When you first login, your system will typically be set to *runlevel 3*. The chkconfig utility handles the task of adding, enabling and disabling services into your Linux *runleve*l system, while the init command allows you to navigate these levels.

Understanding Runlevels

There are eight common <emphasis>runlevels</emphasis> in Linux: 0 through 6, and “S.” Of these levels, 0, 1 and 6 are considered <emphasis>reserved</emphasis>. Level 0 is used to halt the system, level 1 is used to drop the system into single-user mode (denying outside access, sometimes necessary for system emergencies) and 6 is used to reboot the system. (The “S” runlevel is not meant to be used directly, and is available internally for scripts to use at level 1).

Each of the remaining levels are freely available for your own configuration with chkconfig. These can often be useful if you have more than one or two common configurations which you regularly switch between. Each *runlevel* can have its own services turned off and on via chkconfig however you see fit, and then traversed in and out of with the init command.

<note>
Runlevels and X

While not reserved, *runlevel 5* is typically the level used to instantiate the X Windows System services. Therefore, switching to *runlevel 5* will most likely cause your system to try to launch its Graphical User Interface, if it has not already started up.

Using the init and runlevel Commands

If you are uncertain of what runlevel you are working in, the simplest way to determine this is to use the *runlevel* command. (This command is usually only available to the root user, so you may need to use su to switch to root.) This command will list two runlevels: the current *runlevel*, preceded by the most recent *runlevel* that the system was in. If the system has not changed *runlevels* since it last re-booted, it will show "N" as the previous runlevel.

```
[root@imac /root]# runlevel
N 3
[root@imac /root]#
```

This is the most common output that you will receive from the *runlevel* command: this output indicates that the current *runlevel* is "3" (shown by the second number) while there has been no other recent *runlevel* ("N").

To switch in and out of existing *runlevels* with the init command, the syntax is simply init [level], where "[level]" is the *runlevel* that you wish to switch into. For example, if you have some special power-saving services set up for a laptop at level 4, you might type init 4 to switch to that level.

Using the init command

```
[root@imac /root]# init 4
INIT: Switching to runlevel: 4
Starting apm:          [  OK  ]
[root@imac /root]# runlevel
3 4
[root@imac /root]#
```

As you can see, after using the init 4 command, the system called the INIT process, which in turn checked the *runlevel* differences between "3" and "4" and started the apm (Advanced Power Management) service. (The use of *runlevel* after the init command confirms this level switch, as well as shows us that the last *runlevel* preceding "4" was "3."

Using the chkconfig Command

The chkconfig command can be used to list the existing *runlevel* scheme, add new services to the *runlevel* scheme, delete services, or change what levels have

configured for each service.

To achieve these functions, the available arguments that chkconfig understands are:

--list	list the levels for a specific service, or all services.
--add	add a service to the system.
--del	delete a service from the system.
--level	configure a service's levels.

Each of these respective functions have their own required arguments.

Listing Services

To list **all** installed services, and their respective *runlevels* and configurations therein, type simply chkconfig --list, and press ENTER.

Listing Installed Services with chkconfig

```
[root@imac /root]# chkconfig --list
anacron     0:off   1:off   2:on    3:on    4:on    5:on    6:off
netfs       0:off   1:off   2:off   3:off   4:off   5:off   6:off
network     0:off   1:off   2:on    3:on    4:on    5:on    6:off
random      0:off   1:off   2:on    3:on    4:on    5:on    6:off
rwhod       0:off   1:off   2:off   3:off   4:off   5:off   6:off
atd         0:off   1:off   2:off   3:on    4:on    5:on    6:off
atalk       0:off   1:off   2:off   3:off   4:off   5:off   6:off
gpm         0:off   1:off   2:on    3:on    4:on    5:on    6:off
portmap     0:off   1:off   2:off   3:off   4:off   5:off   6:off
ypbind      0:off   1:off   2:off   3:off   4:off   5:off   6:off
ipchains    0:off   1:off   2:off   3:off   4:off   5:off   6:off
xinetd      0:off   1:off   2:off   3:on    4:on    5:on    6:off
sendmail    0:off   1:off   2:off   3:off   4:off   5:off   6:off
identd      0:off   1:off   2:off   3:on    4:on    5:on    6:off
pppoe       0:off   1:off   2:off   3:off   4:off   5:off   6:off
nfs         0:off   1:off   2:off   3:off   4:off   5:off   6:off
nfslock     0:off   1:off   2:off   3:off   4:off   5:off   6:off
kdcrotate   0:off   1:off   2:off   3:off   4:off   5:off   6:off
lpd         0:off   1:off   2:off   3:off   4:off   5:off   6:off
xfs         0:off   1:off   2:on    3:on    4:on    5:on    6:off
syslog      0:off   1:off   2:on    3:on    4:on    5:on    6:off
crond       0:off   1:off   2:on    3:on    4:on    5:on    6:off
postgresql  0:off   1:off   2:off   3:off   4:off   5:off   6:off
httpd       0:off   1:off   2:off   3:off   4:off   5:off   6:off
sshd        0:off   1:off   2:off   3:off   4:off   5:off   6:off
xinetd based services:  chargen:  off  chargen-udp:  off  daytime:  off
daytime-udp:  off  echo:  off  echo-udp:  off  time:  off  time-udp:  off
```

```
telnet: on   cvspserver: on
[root@imac /root]#
```

This lists not only the services which have script files installed in the /etc/rc.d/init.d/ path, but also those services installed for xinetd, a service itself, whose configuration files are kept in /etc/xinetd.d/.

You may also specify a service's name immediately following the chkconfig --list command, if you wish to only view the *runlevel* configurations for a single service.

```
Listing Service Runlevels with chkconfig
[root@imac /root]# chkconfig --list network
network     0:off   1:off   2:on    3:on    4:on    5:on    6:off
[root@imac /root]#
```

This example shows that the *network* service is set up to only turn on at *runlevels 2, 3, 4 and 5*. Therefore, typing *init 1* to drop to single-user mode will turn off the networking services in the process.

Adding Services

To add a service, you must first have a chkconfig compatible start-up/shutdown script to install. This kind of script is frequently installed by default with many systems, but not always. If you find yourself with such a script with a new software package (e.g., in /usr/local/myprogram/bin), you can check it for compatibility by examining the first few lines of the script.

If a script is written for chkconfig, you will find a commented line including the phrase "chkconfig: " followed by the levels at which the script should start (or "-" if no default levels), and a pair of numbers signifying the relative sequence of start-up and shutdown.

Looking at a valid chkconfig script

```
[root@imac /root]# more /etc/rc.d/init.d/httpd
#!/bin/sh
#
# Startup script for the Apache Web Server
#
# chkconfig: - 85 15
# description: Apache is a World Wide Web server. It is used to serve \
#       HTML files and CGI.
# processname: httpd
# pidfile: /var/run/httpd.pid
```

The two relative sequence identifiers are numbers between "00" and "99" which signify the order in which they will be called by init. In this case, the numbers are

"85" and "15" respectively. These numbers specify relative precedence in the *init* process. The first number is the relative precedence with which it will be started, and the second number is the relative precedence with which it will be killed (as the system changes runlevels).

A high number means it occurs later in the *init* process. The higher the number, the later the execution (though it is relative to the other scripts installed, and their startup and shutdown numbers; for example, 85 will still occur before 90).

If you are satisfied that the script is a valid chkconfig script, copy the script into the /etc/rc.d/init.d/ path. Be sure to rename the script to the keyword that will identify the service, if necessary (e.g., names like "start-script" and "linux" are not good names for start-up scripts, as they do not describe specific services).

Once copied, call the chkconfig command with the syntax chkconfig --add [service_name], where "[service_name]" is the name of the valid script.

Adding a service with chkconfig

```
[root@imac /root]# chkconfig --add postgresql
[root@imac /root]#
```

If the script does not complain, the service should be successfully installed! The service will be enabled for whatever levels were specified by default in the chkconfig script. In the case of PostgreSQL, the script automatically installs, such that the PostgreSQL database service will start at levels 3, 4 and 5, which can be verified with the chkconfig --list command.

Deleting Services

To remove an existing service from the *runlevel* initialization sequences you use the syntax chkconfig --del [service_name], where "[service_name]" is the name of the service which you wish to disable. As an example, if you wished to disable the Apache webserver services, you could type chkconfig --del httpd (as "httpd" is the service named for Apache).

Deleting services with chkconfig

```
[root@imac /root]# chkconfig --del httpd
[root@imac /root]#
```

If you do not receive any kind of error (e.g., a misspelling of "httpd" could result in the message: "error reading information on service httpd: No such file or directory"), the service should be removed from any system *runlevels*.

Configuring Runlevels

In order to actually toggle *runlevels* off and on for an already installed service, the syntax is chkconfig --level [levels] [service_name] [modifier], where "[levels]" represents the levels you wish to affect run together into a single argument (e.g., "345"), "[service_name]" is the name of the service you wish to modify, and "[modifier]" is either "on," "off" or "reset." Using "on" will turn the service on for that level, "off" will turn that service off, and "reset" will reset the script for that level as specified in the script's default settings.

For example, if one wished to explicitly turn the Apache services off for all levels except for 3, you could use the syntax chkconfig --level 3 httpd on, followed by chkconfig --level 012456 httpd off>.

Setting levels with chkconfig

```
[root@imac /root]# chkconfig --level 3 httpd on
[root@imac /root]# chkconfig --level 012456 httpd off
[root@imac /root]# chkconfig --list httpd
httpd   0:off   1:off   2:off   3:on    4:off   5:off   6:off
[root@imac /root]#
```

As you can see above, this example is an explicit way of being sure that all *runlevels* except for *runlevel 3* are turned off for the "httpd" service.

IV. Troubleshooting

Troubleshooting

Common Configuration Problems

This part of the book offers some basic Troubleshooting for common configuration problems (a.k.a “user errors”). For further support resources, be certain to review the Appendices and visit www.yellowdoglinux.com/support/

Forgot Your root Password?

Forgetting the root password on your Linux system is obviously a dangerous predicament, as many operations cannot be performed by any other users. For example, without root access you do not have the ability to upgrade packages that may be found to have security holes.

In the event that you forget your root password, Linux does have a built-in mode called “single-user” mode that allows you to bypass the multi-user authentication scheme and boot manually to a Bash prompt. At this point you will have the ability to use the passwd command to reset the root password. This technique requires physical access to the machine that you wish to perform this operation on.

Booting into Single User Mode

In order to boot into single user mode, you must reset the computer so that you can pass special arguments to yaboot. At the machine, you should be able to press CTRL+ALT+DELETE simultaneously to re-boot from a terminal.

Once yaboot prompts you for the kernel image that you wish to load, enter the kernel argument: init=/bin/sh rw (IMPORTANT: remember to include rw at the end as this tells linux to boot up in *read/write mode*, which it needs to be in to save your changes).

```
linux init=/bin/sh rw
```

This example illustrates booting a kernel image named “linux” into *single user mode*.

Passing this argument causes the normal initialization procedure to be pre-empted by a simple instantiation of the <emphasis>sh</emphasis> (Bash) shell.

Resetting the root Password

Once you have booted into single user mode, you need only type passwd to invoke the password-resetting facility for the root user.

Before you re-boot the machine, you should first type sync to be sure that the filesystem buffers have been flushed, and that the changes have been written to disk. You may then re-boot the system into normal multi-user mode with the /sbin/reboot command.

Re-Configuring the Keyboard

If for any reason you need to re-configure your keyboard (e.g., installation of a new keyboard with a different key mapping), you may use the setup command to invoke the keyboard configuration module. Simply type setup, as the root user, and choose the “Keyboard configuration” option.

You will then be prompted to choose from the list of available keyboard configurations.

Frozen Shell?

If at any point in using the Linux shell it appears that your shell session has locked up completely, you may first want to check that you haven't accidentally activated the *scroll lock*. The scroll lock is turned on by pressing CTRL and S simultaneously, and can be turned off by pressing CTRL and Q in the same fashion.

This simple solution may often come in handy, as it is not difficult to accidentally hit CTRL+S.

Konsole Terminal Emulation

Misconfigured terminal emulation settings can frequently be the cause of a number of headaches when using the KDE Konsole shell application. A simple example of this sort of misconfiguration is pressing SHIFT and ENTER at the same time when the keyboard emulation is set to “xterm.” Rather than executing the command buffer as you would expect it to, the pressing of both SHIFT and ENTER can cause unexpected behavior.

If you find yourself constantly having to re-type your password, or other inconsistent oddities having to do with inputting commands in the Konsole, try setting the keyboard emulation to “linux console” (under Settings -> Keyboard).

Resetting Terminals

There are numerous circumstances when any terminal emulator can become confused by bad display characters sent to standard output. If you ever find your terminal garbled, or unable to properly display your commands as you type them, you can usually type reset to return the terminal to its default configuration.

<warning>
Reset is not Reboot!

Take special care to remember the difference between reset and reboot! While rebooting your system will usually solve any kind of terminal emulation problems, it is generally not the most appropriate solution!

Re-Configuring the Mouse

If for any reason you need to re-configure your general mouse settings (e.g., installation of a new mouse with a different protocol), you may use the setup command to invoke the mouse configuration module. Simply type setup, as the root user, and choose the "Mouse configuration" option.

You will then be prompted to choose from the list of available mice.

Losing "Paste" Buffers

When highlighting buffers to copy and paste within X it is not uncommon to lose copy buffers by accidentally highlighting a section of text. The reason for this is that many X applications implicitly copy highlighted text directly into the copy buffer.

Within KDE, you have the option to save recent buffers that are accidentally written over. See the section in Part II of this book on Klipper, the KDE Clipboard application.

Where to Find Help

Terra Soft Solutions

Paid Installation Support

Terra Soft Solutions incorporates Installation Support (www.yellowdoglinux.com/products/faq/install.shtml) with some of its Yellow Dog Linux packages, offering a web-based interaction with a real human being who will work with you to solve Installation issues.

If you purchased YDL with Installation Support and require assistance with the installation of Yellow Dog Linux, please have your *Product ID* and *password* (found on your packing slip) or if you purchased from a reseller, within the package.

www.terrasoftsolutions.com/support/

Fee-Based Support

Terra Soft Solutions offer fee-based, hourly support. If you require assistance, please visit www.terrasoftsolutions.com/services/ to initiate your request for assistance. Please note that Installation Support questions **will not be answered** via this form.

The Linux Community

The Linux Community is a somewhat ambiguous but highly effective, loose conglomeration of individuals, university students and organized bodies, and corporations that contribute to the advancement of Linux.

This community of kernel, library, driver, and application developers work together to constantly improve Linux, providing the fruition of their labor for free, to you. Even individuals or corporations that offer their software or services for a fee usually provide a version or portions of their software for free via internet download.

Application and Server Support

While Terra Soft is responsible for the maintenance of the packages included on the CDs, making certain they install and function well on PowerPC systems, Terra Soft is not the author of nor responsible for individual applications (OpenOffice, XMMS, AbiWord), server packages (Apache, sendmail), nor development packages (gcc, php).
True to the Linux spirit, each author or maintenance group provides a website

(typically not too shabby) which offers the latest updates, Users Guides, bug-fixes and HOWTOs.

Typically, you can enter the name of the application (followed by ".org" to find the website associated with the application, server, or development tool for which you seek additional information or assistance. If this does not work, visit www.google.com and search for the application by name. This should produce a link in the top ten or so of the search results.

The following provide some popular online sites:

www.kde.org	KDE window manager
www.gnome.org	Gnome window manager
www.openoffice.org	OpenOffice, a complete office suite
www.xmms.org	XMMS, X Multimedia System
www.apache.org	Apache, the famous webserver
www.sendmail.org	sendmail, a popular email server
www.mysql.org	MySQL, database server
www.python.org	python, run-time programming language
www.php.org	php, web programming language
www.slashdot.org	*The center of the known universe!*

Online Help

www.yellowdoglinux.com	Yellow Dog Linux (our favorite)
www.justlinux.org	New to Linux? Seeking advanced help?
www.linux.org	General Linux questions, introduction
www.penguinppc.org	PowerPC Linux development (geeks)

Books

For assistance beyond this book, there are hundreds of Linux books available to you online or at your local book seller. As Yellow Dog Linux is a Red Hat-based distribution, you may purchase a Red Hat guide to configuration, maintenance, or administration. *We can't recommend just one as everyone has their own learning style*.

Configuring Konqueror

A Highly Configurable Application

Konqueror exemplifies in its widely configurable design many of the best characteristics that the K Desktop Environment has to offer. The modularity of its components and the extent of available dynamic modifications can truly make it an application which is a genuine joy to use.

This section documents several methods by which you can alter the look, feel, and function of Konqueror, both through the KDE Control Center and dynamically through its own native interfaces.

Konqueror Settings in the KDE Control Center

To locate the general Konqueror configurations you will need to first launch the Control Center. As most of the configurable options we will discuss pertain to Konqueror as a web-browser, the Control Center module that you will need to open is at the bottom of the list, titled *Web Browsing*.

Configuring Konqueror's Appearance

To begin configuring Konqueror's general appearance, enter the Control Center's *Web Browsing* module list and select the *Konqueror Browser* module entry.

You should now see five tabs in the body of the Control Center: “HTML”, “Appearance”, “Java”, “Javascript” and “Plugins”. The two tabs we are immediately interested in are *HTML*, and *Appearance*.

The HTML Tab

Click the *HTML* tab to browse the configurable display options related to HTML browsing. These include when to underline hyperlinks to other documents, whether or not to change the cursor (e.g., from an arrow to a pointing hand), whether or not to automatically load images, and an option to enable a user-defined style sheet.

Configure the “Underline Links” by clicking the radio button next to your preferred option. The options are relatively self-explanatory. Note that “Hover” refers only to when the mouse pointer itself is physically displayed over a link.

Then, check and uncheck each of the options depending upon your preferences. Note that if you wish to enable a style sheet you must have one prepared in CSS (cascading style sheet) format.

The Appearance Tab

Click the *Appearance* tab to view font-specific configuration options for Konqueror's display. While each web page has its preferred specified fonts, you have some control over the web page styles on this tab.

By choosing the desired “Font Size” radio button and setting a minimum font size, you can ensure that regardless of a web page's style definitions the fonts will be in your preferred size range. You may additionally choose appropriate replacement fonts for common font-definitions such as “fixed” and “serif.”

Once you have finished configuring Konqueror's appearance, select “Apply” for the changes to take effect. The changes should become instantly visible upon clicking “Apply,” even in Konqueror window's backgrounds.

Enabling Java and Javascript

Within the “Web Browsing/Konqueror Browser” Control Panel you have two tabs available to configure the functionality of Java and Javascript within Konqueror. They are aptly named *Java* and *Javascript* and are immediately to the right of the *Appearance* tab.

The Java Tab

Konqueror does have support for executing Java applets via the Java Runtime Environment, though it is not necessarily enabled with your system. Click the *Java* tab to open up the available configurable options for Java within Konqueror.

By default, Java is not enabled globally. You either select the “Enable Java globally” checkbox, thus allowing any website to execute or embed Java applets, or you may configure domain-specific policies on how to handle Java applets.

To implement a policy, just select the “Add” button on the right-hand side of the Control Panel. It will prompt you for the name of the domain that you wish to enable, and whether or not to accept or reject Java applets from that domain. This is a security feature for you to be able to either disallow specifically untrusted sites, or to enable just a few trusted ones.

If you choose to enable Java globally you will have a few extra Runtime settings at the bottom of the Control Panel. If you are not an expert in Java you most likely do not need to make any adjustments to these settings.

The Javascript Tab

Click the *Javascript* tab to the right of the *Java* tab to open the configurations available for Konqueror's Javascript support. This tab is very similar to the *Java* tab

in that you may either enable Javascript globally or set up specific policies on how to accept or reject Javascript code.

In addition to these very general enabling and disabling options Konqueror also has a very handy feature in its "Disable window.open()" checkbox. By checking this you can instruct Konqueror to *never* open a new window via a Javascript function. While some sites require this in order to take advantage of dynamic content, this can be a very useful way to turn off unwanted advertisements that frequently open through this Javascript function.

Managing Plugins

A plugin, put simply, is a small program that you can add into another program to provide extra functionality. An example of a plugin is the common Flash application used extensively by web sites on the Internet. Like other web browser software such as Internet Explorer and Netscape, Konqueror is able to make use of plugins to enhance your browsing experience.

To use plugins with Konqueror you will first need to enable them. You will also need to tell Konqueror which plugins it can use and where it can find them on your computer; this process will be covered later in this section. The options for both enabling plugins and for configuring them are located within the Control Center. Launch this by selecting the K Menu, and then *Control Center*. To view and change Konqueror's options double-click on *Web Browsing* and then double-click on *Konqueror Browser*. There are two important areas within this sub-list of options. The first is the option that tells Konqueror whether or not to even use plugins. Double-click *Konqueror Browser* and then click the *Plugins* tab on the right KPanel of the window. Now put a check in the box where it says "Enable Plugins globally" to enable plugins.

The second important area of this list of options is the *Netscape Plugins* item. Double-click on this to display the Control Panel for Netscape Navigator's installed plugins.

After double-clicking on *Netscape Plugins*, you should see two KPanel on the right side of the window: *Scan* and *Plugins*. *Scan* should be opened by default; this KPanel keeps the list of directories in which Netscape plugins can be found. There are two ways to add to this list: you can use the "Scan for new plugins" button, or you can type a directory location into the white box next to the *New* button (or, alternatively, click the "..." button and browse to the directory you wish to add) and then click the *New* button to add the directory to the list below. Once you have added a new directory, use the *Scan for new plugins* button to search for plugins located within that directory. The option *Scan for new plugins at KDE startup* will cause KDE to scan the list of directories for new plugins when you first start it. You can also use the *Up* and *Down* buttons to move a directory up and down through the list, and the *Remove* button to remove a directory from the list. (Note: to use

the *Up*, *Down* and *Remove* buttons you will first need to select a directory from the list.)

The second KPanel in this window is the *Plugins* KPanel; here you can view the list of the various installed plugins. You are able to browse through the information through a collapsible list. By clicking on the plus (+) signs next to list items, you can see a list of sub-items for that item. You cannot make changes from this KPanel. It is only for viewing the list of installed plugins.

Dynamic Konqueror Re-Configuration

There are various Konqueror options that you can configure dynamically. You can change these options on the fly, and the screen you are currently using will reflect these changes immediately. You will find that some of these options are similar to those that can be found in the file-browsers of more mainstream operating systems (such as the Windows™ Explorer or the Mac OS™ file manager). As you'll soon see, while other operating systems have file browsing applications with sometimes few and mostly trivial dynamic options, Konqueror has many powerful options.

Configuring Toolbars

As in many applications, toolbars play a rather important part in Konqueror. They contain shortcuts to commonly-used functions within the program and are available for your convenience. If used correctly they can improve your experience with the software, but their use is not necessary for the program to function properly.

Showing and Hiding Toolbars

There are a few different toolbars available within any given Konqueror window. The visibility of these toolbars is controlled by a check-mark list located in the *Settings* menu, found at the top of Konqueror windows. The available toolbars are as follow (for a normal Konqueror window): menubar, toolbar, extra toolbar.

The *menubar* is the bar at the top of the screen that lists the *File*, *Edit*, *View*, *Go*, *Bookmark*, *Tools*, *Settings*, *Window*, *Help*. If you uncheck this option you will lose the direct ability to use any of these menus. To show the menubar again, simply right-click in the body of the window and click on "Show Menubar." Alternatively, you can hold down the *alt* and *m* keys on your keyboard to hide and unhide this Title bar.

The *toolbar* is the bar near the top of the screen that displays small icons as shortcuts to often-used functions of the window.

The *extra toolbar* is a small bar that displays a few lesser used functions, such as dynamic window alterations.

The *location toolbar* displays the path or network address you are currently accessing. It also allows you to type in a path or network address that you'd like to access.

Moving and Flattening Toolbars

There are two different ways to dynamically move toolbars around a Konqueror window. You can right-click on a toolbar to bring up its context menu (titled "Toolbar menu"), and select *Orientation*, then choose the orientation you would like the toolbar to use. This will reposition the toolbar at your specified location in the window.

The second way you can dynamically move a toolbar is to left-click on it and then drag it to the desired position on-screen.

Configuring Views and Profiles

Splitting Views

Unlike many file and web browsers, within Konqueror it is possible to change the structure of a window in various ways. These options are powerful and can enhance your experiences with the program. One of the most important ways in which you can alter the structure of a window is by splitting the views. If you click on the *Window* menu option of any Konqueror window, you will see a list of Window options. The following options are relevant to splitting windows: Split View Left/Right, Split View Top/Bottom, New View at Right, New View at Bottom, and Remove Active View.

- This option causes the view of a window to be split in half through the middle, causing two separate "views" (windows) to appear within the same Konqueror window. By clicking in the body of one view you make it active, meaning you can type in an address in the location bar and Konqueror will know which view to send the address. This also works the same way with other window and program options: if you click within a view to make it active, Konqueror will send any change of options or information to the active view.
- As in *Split View Left/Right*, this option splits a window into separate views. As implied by the option's label, this splits a window into a top and bottom view. As before, each view becomes active when clicked on, and when you perform an action in a window that has been split it is sent to the active window.
- This option adds another view on the right side of the window.
- This option adds another view on the bottom of the window.
- To remove a view, click within its body to set it as the active window, then

choose this option from the list of window options.

Adding Components

As you can see by looking at the *Window* menu, Konqueror provides further window options beyond splitting views. Two of these are the optional presence of a sidebar and an embedded Konsole terminal. Enabling these lessen the viewable area of a window, but they provide functionality that can be helpful in certain situations.

- When enabled, the *sidebar* will appear as a large white frame on the left side of the window. Within the sidebar there are five items: Bookmarks, History, Home Directory, Network, and Root Directory. Double-clicking on *Bookmarks* will display a list of all your Konqueror bookmarks, which you can then select to visit. The *History* item displays a list of all recently visited websites and locations, organized by the website or location address. The *Home Directory* item lists the contents of your home directory. Directories within your home directory are displayed with a plus sign (+) next to them to denote that they are expandable. The *Network* option displays different items related to your network (most importantly, it is here you are able to browse your local network if it is configured properly). The *Root Directory* item displays the contents of your root directory (the / directory; i.e., the beginning of the Linux file tree). You can use the *Root Directory* item to easily browse through your filesystem.
- If checked, the embedded Konsole option will open a terminal window embedded directly your Konqueror window. It displays at the bottom of the window and is not movable, though the height is re-sizable. You can use this terminal as you would a normal Konsole terminal window.

Using Profiles

Konqueror uses "Profiles" to control various options and aspects of the layout of its windows. There are four different profiles: File Management, File Preview, Midnight Commander, and Web Browsing. Konqueror will open with different profiles depending on how you accessed it originally; for example, if you open it as a web browser, it will logically load the web browsing profile. Likewise, if you load it to browse through your filesystem it will load the file manager profile. You can change which profile Konqueror uses at any time by selecting *Window* and then *Load Profile*. You can now select one of these profiles to use. Note that this will change how your window looks and will not carry over to previous locations you were viewing.

Configuring Profiles

Along with the ability to select pre-defined profiles, you are also able to configure

them with the *Configure View Profiles* menu option. This opens a window titled “Profile Management.” This allows you to do three things: save your current settings as a profile (to do this you must first type in the label you'd like to save the profile under; after you type in the label, the 'Save' button will become clickable); rename existing profiles; and delete profiles.

A History of Linux

Starting in the 1960's, two programmers for AT&T Bell Labs, Ken Thompson and Dennis Ritchie, developed a multi-user, multi-task-ing operating system, called UNIX.

In 1991, Linus Torvalds, a student of the University of Helsinki in Finland, began a project to explore the 386 chip on his computer. Through a university class he became interested in UNIX, bought a PC, and worked in Minix, a small UNIX operating system. Unsatisfied with Minix, Linus began to develop the kernel (the heart of the operating system) that eventually became the Linux operating system.

The history of Linux is closely connected with the history of the Internet. From the beginning, Linus posted his ideas and the progress of his project to newsgroups on the Internet. Other students and software engineers quickly became interested in what he was doing and became excited by the chance to work on the source code of an operating system themselves. Linux grew with the contributions of fellow programmers around the world into a full multi-user, multitasking operating system. Linus and thousands of other programmers around the world continue to work on the Linux kernel today.

The GNU Project and Free Software

The GNU (literally "GNU is Not UNIX") Project began in 1984 with the intent of developing a free UNIX-like operating system.

The Linux operating system is protected under the terms of the GNU General Public License, a special software license created by the Free Software Foundation. The Free Software Foundation is a tax-exempt charity that promotes the philosophy of free software. Free software is not necessarily free in terms of price, but it does allow the user of it certain liberties and freedoms not allowed by proprietary software. These liberties include complete access to the source code of the software for anyone who wants to change and improve it, and the right to copy the software and give it away or distribute it for a fee. The working concept of free software is this: when someone buys or acquires a piece of software he or she becomes the owner of it and should have the right to change and alter it as seen fit. The Free Software Foundation was formed in order to promote and protect these rights.

Open Source software is software that allows access to the source code or is distributed with the source code in addition to the executable. "Open Source" typically falls within the definition of free software, though there are individuals who still debate heatedly the finer points between the two terms.

At the time of Linux's inception, the GNU project included many Free Software applications but needed a kernel. Linus Torvalds released Linux under this software license in order to ensure that his software would remain free and to promote the philosophy of free software.

A complete copy of all software licenses, including the BSD Copyright, X Copyright, and GNU Public License are available online at the Yellow Dog Linux website:

www.yellowdoglinux.com/resources/gnu.shtml

<title>B</title>

Bash
Bash (the "Bourne Again SHell") is the default Linux shell environment. It is the GNU version of the original Bourne shell. Other shells, such as <emphasis role="strong">csh</emphasis> and <emphasis role="strong">tcsh</emphasis> can also be used within Linux.

Boot
The series of actions a computer takes after it is turned on. These actions startup and run the currently chosen Operating System.

<title>D</title>

Device Driver
Device drivers are small programs designed to help linux communicate with and use a piece of hardware (a device) that is attached to the system. Usually the manufacturer of your hardware will provide you with a device driver to use with Linux; if not, you should often be able to use a standard driver. If nothing else, you can always search the web for drivers for your device.

<title>K</title>

K Desktop Environment
The "K Desktop Environment." This is the default desktop that comes installed with Yellow Dog Linux 2.x. (No, the K doesn't actually stand for anything.)

<title>S</title>

Shell
A term for a virtual terminal, or console. Bash is the default shell for Yellow Dog Linux 2.x.

<title>X</title>

The X Window System
The de facto standard for Graphical User Interfaces in Linux. This is the software the desktop environments such as KDE and GNOME rely upon to actually display

to the screen.