PeepCode
press

$9

Using Unix

Get started with the command line

by Casimir Saternos

Unix Basics

CONTENTS

ABOUT THIS BOOK

Using UNIX can be intimidating! If you are familiar with Windows or the graphical elements of Mac OS X, you may be overwhelmed when presented with a blank terminal screen, waiting for a command.

But don't panic! You only need a dozen short commands for most of what you'll need to do as a web developer. As you become comfortable, you'll learn to customize your environment for greater efficiency. This book is designed to provide the information to quickly and effectively address your daily challenges and is extensive enough to give experienced developers insights that will increase their effectiveness and productivity.

NOTE Some developers have blogged about their most frequently used commands. They can be found by searching Google for command history meme (http://www.google.com/search?q=command+history+meme).

The purpose of this book is to give you insights into how to develop, install, monitor, and maintain web applications on UNIX servers. We'll assume that either:

- You have access to a UNIX server (a shared host, a virtual host, or a dedicated server), or,
- You're using Mac OS X (which uses BSD UNIX)

The intended audience for this book is web developers who create, deploy, maintain, and troubleshoot web applications that run on UNIX servers. Developers planning to deploy to a Linux-based virtual server are faced with a challenge if the majority of their development was completed on a Windows or Mac OS.

Sometimes web developers naively think they can avoid learning about UNIX details that can be left to system administrators, but this is not a realistic ideal. Many Rails developers deploy their web applications with Capistrano. Although it utilizes Ruby-based configuration files, a quick glance at Getting Started with Capistrano (http://www.capify.org/getting-started/basics) reveals that the often used `run` function is simply a means to execute system commands.

> *You should be comfortable working from a command line. You do not need to know advanced shell scripting techniques or anything like that (though it helps, if you want to start writing complex Capistrano recipes), but you should be able to navigate directories and execute commands from the command-line. Capistrano has no GUI interface; it is entirely command-line driven.*

So to be effective with Capistrano, you need to be conversant with the commands available on the underlying operating system (and better yet, bash shell scripting).

NOTE We'll use the terms *terminal*, *command line*, *command prompt*, and *shell* to refer to the application used to enter UNIX commands.

With web development in mind, there are a number of subjects traditionally covered in books on UNIX which are not addressed here. System administration tasks are limited to the types that developers are most likely to address and will provide enough information for you to communicate effectively with system administrators. There is no coverage of kernel development, file system features, or graphical user interface environments except when a subject is related to the work of a web application developer. The aim is to filter out the noise and give you immediate access to the features that will help you in your development tasks.

Even if you are not using UNIX on a daily basis, it is worth noting that UNIX has exerted a considerable influence on the design of other technologies. Mac OS X, Tivo, routers, and hand held devices run Linux under the hood. In addition, many of the utilities have corollaries in classes in the Ruby programming language. See the chart below for some examples of features of Ruby that are based upon UNIX utilities.

Command	FileUtils	String	Array
cd	X		
pwd	X		
mkdir	X		
ln	X		
cp	X		
mv	X		
rm	X		
chmod	X		
chown	X		
touch	X		
grep		X	
tr		X	
uniq			X
sort			X

Numerous other examples might be mentioned including regular expressions and the `date`, `time`, and `find` classes. The bottom line is that becoming better acquainted with UNIX functionality will make you a more effective developer even when you are not working directly on a UNIX system. It introduces you to a way of thinking—and a programming culture—that continues to exert a significant influence on software design decisions today.

There are a number of UNIX variants, branches and distributions (called *distros*) of UNIX operating systems. Each of these have specific features and unique implementations that differentiate them from their siblings. However, there is a rather large body of common functionality which allows most users to use a consistent set of commands on each system. This book will focus on functionality that is available to the majority of those in popular use.

The commands that are introduced in this book will be described, but not in a comprehensive manner. The book can be read from beginning to end to provide a relatively comprehensive overview of UNIX that focuses on the interests of web developers. You can also read individual sections to get an introduction to the resources available in a particular problem domain and specific examples of how to use UNIX effectively for a particular task.

About UNIX

CHAPTER 1

History

In the mid 1960's, a group of companies set out to build a computing utility service that could be used by many people at once like the phone or electric utilities. MULTICS (http://en.wikipedia.org/wiki/Multics) never became as popular as originally hoped, but some of the engineers who worked on it went on to develop a more focused operating system called UNIX.

UNIX was originally developed in 1969 at Bell Labs. Throughout the 1970's UNIX was used and improved upon, primarily by scientists and technicians in research and educational settings. By the late 70's UNIX began to be adopted in business settings by students who had done undergraduate and graduate work on UNIX. During the next decade a number of different versions of UNIX were in use and standards related to UNIX were developed. Linus Torvalds introduced Linux in the early 1990's as a rewrite of MINIX (http://en.wikipedia.org/wiki/MINIX) (a minimal UNIX operating system).

It's important to realize that UNIX is an operating system that was written by programmers for programmers. Programmers tend to think of writing programs in their language of choice to solve problems. However, it is possible to solve complex problems using nothing but built-in UNIX functionality. You might think of UNIX as a component architecture in this context. With a proper understanding of the programs available (and how they can be used together) you will often find yourself quickly solving problems through a script or series of commands without the overhead of having to resort to coding an entire independent program.

Philosophy

The UNIX philosophy was never formalized, but here is an attempt to summarize the fundamental design principles that guided the developers of the operating system.

Do One Thing Well

Throughout this book you will encounter UNIX commands and utilities. Each of these is designed to perform a single task well. New programs were created when there was concern about extending the functionality of a program in a way that would confuse the original intent of a program. Each command has a short, descriptive, lowercase name that reflects its central intent.

NOTE Most UNIX commands are only a few characters long. You may want to shorten commonly used commands even further with *aliases*, which we'll talk about later.

Output is Input

Very few UNIX utilities require interactive input, and their output is in a form that is suitable for use by another program: plain text! This makes it easy to use several programs together, even if they aren't aware of the other program. Programs can be chained together in a sequence, and some programs have a primary usage as filters for other programs. Each filter in a series of commands serves one small purpose in modifying a stream of text before it is transformed into its final output.

NOTE A simple example is `wc`, a program that reports on the number of characters, words, and lines in any string of text.

Everything is a File

Files can be classified as ordinary files, directories, or special files in UNIX. However, *everything* that the OS is concerned with is represented as a file. This means that diverse resources such as documents and directories as well as hard-drives, CD Drives, DVD Drives, keyboards, printers, monitors, and terminals are all accessible through a common method of communication.

This results in some nonintuitive uses of commands. For instance, according to the man pages the `lsof` command lists open files. However, you can use the utility to determine the program listening on port 80 (`lsof -i :80`). Network resources are also represented in files in this context, which is a source of confusion to the uninitiated.

Help!

Since this book is not comprehensive, you should be aware of how you can expand your knowledge on your own in a particular area of interest. There are a huge number of utilities, scripts and programs available on most standard UNIX installations. You will most likely find that there are several *thousand* programs available in yours—some of which are so complex that entire books are dedicated to them.

The `man` and `info` commands are the standard utilities to use to learn more about each command you encounter. The `man` command can be called with a program as the argument, and the program's manual page will be displayed. If you are not sure of what program you need, the `-k` option will be of assistance. For instance, type the following to get information on programs related to the Apache web server:

```
% man -k apache
ab(8)          - Apache HTTP server benchmarking tool
apachectl(8)   - Apache HTTP Server Control Interface
apxs(8)        - APache eXtenSion tool
```

```
httpd(8)      - Apache Hypertext Transfer Protocol Server
```

Once a given utility has been identified, you can simply type `man` followed by the utility name to see the manual page. Press `space` to scroll through the page and `q` to quit. And in typical recursive fashion, you can `man man` to find what other `man` options are available.

```
% man ab

NAME
  ab - Apache HTTP server benchmarking tool

SYNOPSIS
  ab  [ -A auth-username:password ] ...
```

Command	Description
man	Display the manual page. Press `space bar` to scroll through and `q` to quit
man -k	Search the `whatis` data-base for commands that match
man -f	Equivalent to `whatis`, which provides a short description of a system command
info	Display an info page (more comprehensive than `man`)
<command> -h	Display the help screen
type -t	Identify a command as an alias, keyword, function, builtin, or file

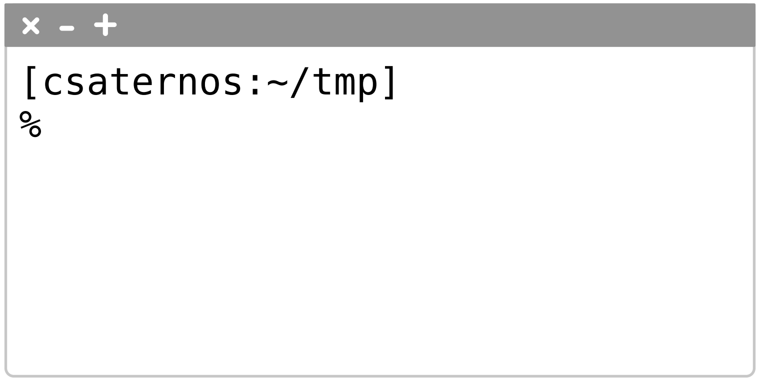
Navigation

CHAPTER 2

You'll learn best if you try to follow along with these examples or the accompanying screencast.

Starting Here, Starting Now

If you're using Mac OS X, you can follow along with most of this book using the Terminal application on your local machine. Otherwise, you'll need to connect to a UNIX server.



```
[csaternos:~/tmp]
%
```

FIG. A **Command line prompt**

There are a number of ways that you might access a server. Although there are a variety of graphical desktop options available when logging at the console or through VNC (remote screen sharing), the focus of this book is going to be command line access. Much of the power of the UNIX operating system can *only* be realized when accessing the system from the command line.

The `telnet` and `rsh` (remote shell) applications are old, standard,

THE PROMPT

The command line prompt can be customized. You can put almost any information into it, including:

- Your username: `csaternos`
- The name of the host your commands are running on
- Your current directory: `~/tmp`
- Contextual information, such as the name of the current Git repository branch

We'll use a `%` to signify the place where commands are issued.

insecure means for accessing systems. It is more common (and secure) to use `ssh` today.

On Mac OS X, go to Applications → Utilities → Terminal. On Windows, PuTTY (http://tinyurl.com/2r4w) is a popular SSH client that can be used to connect to a UNIX server.

To connect to a remote server, you'll need the following:

- Your username, such as `csaternos`
- Your password
- The domain name of your server, such as `peepcode.com`.

To connect from the command line, run this command with your information:

```
% ssh csaternos@peepcode.com
```

You will be prompted for your password and can now issue commands on the remote server.

NOTE Later we'll talk about how to login automatically without a password using SSH keys.

Where Am I?

Logging in locally or via SSH will drop you into your home directory. This is the directory where you can store files and create directories.

The `pwd` command (print working directory) can be used to print your current location in the filesystem. The `cd` command can then be run to navigate to a new directory. Within a directory or set of directories, you can view the contents using the list (`ls`) command.

Path Descriptions

Many commands take a filename as an argument. The argument can be an absolute path, a relative path, or a special character that designates a directory. Here are a few:

Path	Description
~/	Your home directory
/Users/topfunky	An absolute path. Starts with a slash and specifies all the intervening directories needed to get to a location
./bin	A relative path. The two periods and a slash signify one directory up from your current location. These can be added an several times to navigate up several directories.
-	The last directory you were in. This might be in a completely different location and is useful for returning there (with `cd`) or copying files.

The `ls` command includes a number of options that allow you to filter, sort, and format file listings as well as to specify relevant information about the files listed. By combining the results of an `ls` command with other utilities (such as `awk`, `grep`, `sort`, and `uniq`) a huge number of options become available. Although it's a well-known command, many people are not familiar with the wide range of uses available to `ls`.

The use of an asterisk as a *wildcard* or *globbing pattern* is a common idiom. Although `glob` is not a command, you can `man glob` for more information. If you need to include a wildcard name in a command, you can indicate that the wildcard is to be treated as a literal by the shell by enclosing the name in quotes.

Wildcard	Description
?	Represents any single character to be matched
*	Represents any number of characters to be matched
[]	Specifies a range matched
{}	Contains a list of comma-separated values, any of which will be matched
[!]	Excludes a range matched
	Used as an escape character to treat a special character as a literal

The tool `time` is noteworthy when trying to track down what file in a group has changed recently. To sort the results of `time`, use the `-t` option. If you are more concerned about the size of the files, include the `-S` option. Piping the results to `more` or `less` will allow you to page through a large list of results. Grab the first line in a file piping to `head -1`, or the last line using `tail -1`.

If you want to use the the listing of files as string literal input to a subsequent program, you can use the `-Q` option to enclose each name in quotes.

If you wanted to remove a set of files that you have listed, you could use `xargs` command (rather than directly piping to the `rm` command):

```
% ls \*/\*diffs.txt | xargs rm
```

Command	Description
cd	Change directories
ls	List files
xargs	Build and execute command lines from arguments in standard input

Where Am I Going?

Once you have logged on to a server, you will want to get familiar with what resources are available and begin making customizations to meet your particular needs. Typically, you will immediately check your that your environmental variables (e.g. PATH) are set as intended, and perhaps validate the system specifications.

In the bash shell, a file named `.bash_profile` within the user's home directory contains environmental settings specific to the user. You might want to adjust the settings in this file. In addition, there are times that you might *source* this file (execute the script in such a way that the variable settings become present in the current session). Scripts scheduled through `cron` often have this requirement. The way to *source* your bash profile is as follows:

```
% . .bash_profile
```

TIP See dotfiles.org (http://dotfiles.org) for profile samples from other developers.

You can use the `env` command to see the settings of variables within your environment. You can use `export` to make new variable settings available to the environment.

Command	Description
env	Outputs the variables that are set in the environment
export	Make variable settings available to the current environment
alias	Display or set substitutions for commands or groups of commands
who	Display information about users that are logged in
uname	Output information about the system (kernel release, hardware info, processor, etc)
hostname	Output the name that identifies the server

Who Am I?

As in the rest of life, it's important to know who you are.

Because UNIX supports multiple users and each user has specific privileges, it is important to understand who you are logged in as. Problems can occur when a file is created by one user and a later attempt is made to access the file by another user.

Most installations of UNIX start out with several user accounts and groups. Here are a few that may be present on your system:

User	Description
root	The administrative super-user account. The root user can perform any command on the system. If an attacker gained access to the root account, he could take over the rest of the system.
nobody	An unprivileged account used by applications that want to restrict access as much as possible.
daemon	A faceless account used to run background tasks.

Rather than logging on with root privileges, a user can be granted access through the `sudo` command to perform privileged operations.

In this example, I'll try to find the file sizes of all the directories on the current machine. Some are protected, so I'll have to rerun the command with `sudo` in order to have access to all the files.

```
% du -hs /*
du: `/usr/share/ssl/CA’: Permission denied
```

```
% sudo du -hs /*
Password:

3.6M      /bin
2.7M      /boot
568K      /dev
4.5M      /etc
772M      /home
1.3G      /usr
```

Command	Description
whoami	Identify the user you are currently logged in as
passwd	Change your password
su	Switch user
sudo	Execute a command as another user. Frequently used to allow super user (root) access
visudo	Editor for the sudoers configurations file which limits access available through `sudo`

Power Tip: SSH Keys

It's tedious to type in a password every time you need to connect to a server.

If you are interested in being able to log in without being prompted for a user name and password, you need to set up a private/public key pair using the key generation utility. Rails developers planning to use Capistrano are required to use ssh to make connections. Because of the requirement of using the same password on all servers when using regular logins, it is advisable to access your servers using a public key using ssh-keygen.

The following steps can be used to generate a key pair, transfer the public key to a server, and log in (or copy files via scp) without a password. In this example, the destination server name is destinationserver and the OS username is webuser.

1. Generate the key on the client.

```
  % ssh-keygen -q -f ~/.ssh/id_rsa -t rsa -C “Demo of Key Generation”
```

2. Transfer the Public file (id_rsa.pub) to the server in a file named authorized keys in the .ssh directory within the user’s home directory.

```
  % scp ~/.ssh/id_rsa.pub webuser@destinationserver:.ssh/authorized_keys
```

3. Login and transfer are now available from the client to the server without a password.

```
  % ssh webuser@destinationserver
  % scp source_file.txt webuser@destinationserver:destination_file.txt
```

It is generally a bad idea to log on directly as root for normal maintenance. If multiple users log on as root, there is no way of auditing the activities that have occurred on the server. The sudo command (and visudo command which allows an administrator to maintain the sudoers configuration file) can be used to provide auditable access to actions and resources reserved for superusers.

Command	Description
ssh	Secure shell remote login
ssh-keygen	SSH key generation and management

Modification

CHAPTER 3

Editing Files

Because everything in UNIX is a file, it is crucial to be conversant with the options available for viewing, editing, and manipulating files. The `vi` editor is a common, powerful interactive text editor. You will need to invest a bit of time to be somewhat conversant with it, but is well worth the time. It also serves as the editor for a number of special editors that are designed to safely modify certain system configuration files.

Special Editor	Description
crontab -e	Edit the file used to specify special processes run by `cron`
visudo	Editor for the sudoers file that allows controlled access to privileged operations
vipw	Editor for the password file
vigr	Editor for the group file

Although interactive editors are irreplaceable, there are many tasks that can be completed more easily and accurately using stream editors, commands, or scripts. The contents of two files can be concatenated using the `cat` command, but a more common usage is to pass a single filename as a parameter and view the results on the screen. (The lesser-known `tac` command functions the same way but processes files in reverse). When viewing the contents of a larger file using `cat` and you are interested in viewing the contents a screen at a time, the results can be piped to `more` or `less`. Amusingly enough, `less` *is* `more` and, well more. If you are interested in only viewing the beginning or end of a file, `head` or `tail` can be used respectively. A common usage of the `tail` command is to view the end of a log as output is added to it.

For example:

```
% tail -f production.log
```

Press CTRL-C to escape.

The `sed` stream editor can be used to do search and replace operations in files. When more complex stream-oriented modification of files is needed you will likely want to switch over to ruby or perl. Finally redirection to a file using the `>` (create new) or `>>` (append) operators is a way to write the results of a command to a new or existing file.

Command	Description
cat	When passed a single file as an argument, displays the contents to standard out
tac	Concatenate/print a file in reverse
more	Filter for viewing text one screen at a time
less	Functions like `more` but allows backwards scrolling and other features
head	Output the first part of a file

Command	Description
tail	Output the last part of a file
vi	An editor standard to most UNIX installations
emacs	An extensible customizable text editor that includes a Lisp interpreter

Permissions

Many problems encountered by new UNIX users are related to setting file ownership and file permissions. Permissions can apply to users, groups or others.

The `read`, `write` and `execute` bits are the standard permissions and there are also some special modes that can be set. There is an *octal notation* that is used to specify file permissions summarized the chart below.

Permission	Description	Digit
--	No permissions	0
-x	Execute	1
w-	Write	2
`-wx`	Write and Execute	3
`r-`	Read	4
`r-x`	Read and Execute	5
`rw-`	Read and Write	6
`rwx`	Read/Write/Execute	7

You can change both the owner and the group of a file at one time:

```
% chown oracle:dba x.txt
```

The octal notation or the letters can be used to specify permissions. The command to change group, owner, and *all* permissions is listed below using letter abbreviations and octal format.

```
% chmod goa+rwx  x.txt
% chmod 777 test.txt
```

Command	Description
chown	Change Owner of files/ directories
chmod	Change Modification of files/directories
chgrp	Change Group of files/ directories
umask	Set the default file system mode for newly created files and directories
alias	Display or set an alternate name for a command

Aliases and Inodes

There are several different *names* that can be used to specify commands and files in UNIX. As mentioned earlier, the `ln` command can be used to either specify another name for a file or provide a pointer to a file. An `alias` can be used to explicitly specify a shortcut (including a parameter or even an entire command sequence) for a common activity. Besides these mechanisms that allow you to create new *names*, the file names that we ordinarily act upon are really pointers to a numerical designations for each file, known as inodes.

ALIASES

The @alias@ command with no arguments displays the aliases in use in the shell. Aliases can be used to help users who are familiar with commands on other operating systems.

```
% alias dir='ls -l'
```

They can also be used to correctly interpret typos (e.g. aliasing `ls-l` which is missing the space between the command and option to `ls -l`). Commands such as `cp`, `mv` and `rm` can be aliased to prompt before overwriting or removing files.

An alias can be removed by using the `unalias` command. Aliases are expanded by default in interactive shells (see the `shopt` command `expand_aliases` option for details). This can result in unexpected behavior when they are used in scripts.

INODES

You can use `ls -i` to print the index number, or inode of each file. The `stat` command will display the inode number as well as its status and attributes. An inode contains all of the file's properties (permissions, ownership, type of file) as well as the addresses for the data blocks containing the file's content. Several file names can link to the same inode. The `rm` command removes the link that points to its inode, but does not remove the file contents (so the inode has no name linking to it at all). If another process still has the file open, you can even restore a deleted file.

```
% echo "This file will be removed..." > file_to_remove.txt
```

```
  % echo “But we will get it back.” >> file_to_remove.txt
```

Now let’s force a process to hold on to the file while we deleted it. We open it with `more` but only read a line at a time

```
  % more -1 file_to_remove.txt
```

Type `Ctrl-Z` while `more` is still running. This will force the process into the background. The following line is initially displayed and also will be seen if you type `jobs`.

```
  [1]+ Stopped more -1 file_to_remove.txt
```

Now let’s remove the file

```
  % rm file_to_remove.txt
```

Try to `ls` or `view` file. It does not appear to be available. However, the file is still open by the `more` command as can be seen using the `lsof` command.

```
  % lsof | grep file_to_remove
  % more  26133 csaternos  3r  REG  8,5  54    4823091 /home/cs/file_to_remove.txt (deleted)
```

The output from the command provides the information necessary to restore the file by copying it out of `/proc`. The contents of the original file are designated as follows:

```
  /proc/<process id from column 2>/fd/<number embedded in column 4>
```

Based upon the output displayed above, we could run a command to restore the file and view the contents:

```
% cp /proc/26133/fd/3 restored.txt
% cat restored.txt
```

Command	Description
alias	View or create aliases
unalias	Remove aliases
lsof	List open files
shopt	View or modify shell options

Customization

CHAPTER 4

Tow the (Command) Line

Once you are logged onto the server, you will want to collect some basic information about what is available to you. The *shell* is an interface of sorts to the underlying operating system. It dictates what built-in commands are available to control the OS kernel.

There are a number of different ones available. The *Bourne Again* shell (bash) is the default shell on the Mac OS X and most Linux systems. It is available on most UNIX systems and will be the shell assumed for the rest of the book.

Once you become familiar with the basics, you may want to look at the unique features of other shells:

- zsh (http://www.zsh.org)
- fish (http://fishshell.org)
- tcsh (http://www.tcsh.org)

Shortcuts are a key factor in improving your accuracy and productivity in any environment. The table below lists some of the popular ones. Others are available online at the Nuby on Rails blog (http://nubyonrails.com/articles/useful-shell-shortcuts).

Shortcut Key	Description
Ctrl-R	Search your bash history. Type the first few letters after hitting Ctrl-R and press enter when the command is selected
Ctrl-T	Swap the last two characters you typed
Ctrl-C	Cancel the currently running command
Ctrl-D	Log out of the current terminal
Ctrl-Z	Stop a running Job (see `bg` and `fg` for more information)
!!	Repeat the last command
!$	Recall the previous argument
<arrow-up>/<arrow-down>	Scroll through Command History
<TAB>	Complete a command or name based upon entries available in a given context

Being able to repeat the previous argument allows for convenience and accuracy (fewer key strokes) as well as safety when executing commands.

Command	Description
history	Print the command history

Command	Description
whatis	Short description of a command
clear	Clear the screen
stty	Change or display terminal settings

Organization

CHAPTER 5

Moving in Stereo

Files can be moved around the file system using `mv` and copied using `cp`. Moving files between computers involves the use of other commands or protocols (`ftp`, `scp`).

Moving *large* files around can be a challenge, but numerous options are available. Files can be compressed using a number of different utilities including `tar`, `zip` and `cpio`. Large files can be broken into more manageable bits using the `split` command. However, for large files that need to be kept in sync across servers the `rsync` is the most efficient means to minimize the amount of data that needs to be transferred.

It's in the Files

Shortcut Key	Description
touch	Change the file time or create a file if it does not exist
ln	Create a link (short-cut) to another file
sed	A stream editor

If you have a script where a command relies upon the existence of a file, you can either test for the file's existence and take an appropriate action or simply `touch` the file in advance to ensure that it exists.

Links are analogous to Windows shortcuts. Symbolic links allow a given *symlink* or *soft link* with one name to refer to a file with another. Some operations (like an `rm`) operate on the link itself but most operations affect the targeted file. Symbolic links are created using the `-s` option:

```
  % ln -s /<target> <my_link_name>
```

Hard links function as a different name of an existing file. Hard links are generally not allowed for directories (unless a special option is specified for a super user). You can, however make symbolic links to directories. You might want to create a link for a directory if you have an additional disk added to your system that is available on a different file system. However, it can be a bit confusing when working with links to directories because of the fact that some operations work on the destination file and some work on the link. Consider the following:

Start with an empty directory

```
  % mkdir testing
  % cd testing
  % ls -lth
  total 0
```

Create a Symbolic link to another directory:

```
  % ln -s /work
  % ls -lth
  total 0
  lrwxrwxrwx  1 csaternos users 5 2008-03-20 09:34 work -> /work
```

Navigate to that directory

```
% cd work
```

Now let's look at the parent directory—Oh wow—it's the parent of the destination!

```
% ls ..
bin  boot dev etc home lib mnt opt proc root sbin sys tmp usr var work
```

Hey let's go up a directory...

```
% cd ..
```

We are back in the directory where originally created the link

```
% ls
work
```

There are numerous interactive editors available for UNIX. The classic `vi` editor is available on nearly all platforms and `emacs` has become a favorite among rails developers. One early editor is named `ed`, and a stream oriented variant of this editor is named `sed`. The `sed` program can be used to do search and replace operations in files.

```
% sed s/black/white/ original.txt > modified.txt
```

If the tasks become more sophisticated, perl and ruby can be used to your advantage. In particular, *one-liners* may be created that require a simple invocation of the perl or ruby interpreter along with arguments that include a few commands and files to process.

Find Your Way Back

The ls command is sufficient when viewing the contents of a single directory or a few directories withing a given hierarchy. However, it is not useful for identifying files that are in a number of different locations. The find utility can be used to search for files in a more sophisticated manner. One simple example to get you started:

```
% find . -name *.rb -print
```

The dot indicates to search within the current working directory. The *.rb indicates that any file name ending with the .rb extension is an eligible target. Other file attributes (e.g. size, permissions, type) can be used; for instance, this command shows all files within the current working directory that have been modified in the past 5 minutes:

```
% find . -mmin -5
```

The find command is not limited to simply listing file names. find can execute commands directly, and you can also pipe results to xargs:

```
% find . -name *pc.rb | xargs ls -l
```

This opens up all kinds of possibilities—executing every file found, removing files of a certain type or age, etc. At some point, it becomes simpler (or perhaps clearer) to write a script within ruby to accomplish the task:

```
#!/bin/env ruby
require ‘find’
Find.find(‘.’)do |x|
  # Process the file here...
end
```

The find command is very flexible and comprehensive in its ability to search for files, but it can be slow because of its design. The `locate` command relies upon the existence of a database of files. Typically a system is set up to run a process on a regular basis to keep the database up to date. The `locate` command is quicker than find, but is only as accurate as the database it is accessing. In the following example, the `locate` database needs to be updated before it can be used to find a file.

```
% locate test.txt
warning: locate: warning: database
/var/lib/slocate/slocate.db’ is more than 8 days old

% locate -u .
% locate test.txt
/root/Desktop/download/Python-2.5.2/Lib/test/test_doctest.txt
/root/test.txt
/usr/local/lib/python2.5/test/test_doctest.txt
```

The find and `locate` commands are great general purpose utilities, but a rather common need is to determine the location of a program or determine the version of the program in use. The `whereis` and `which` commands are targeted for this usage.

Command	Description
find	Search for files within a given directory hierarchy
locate	Faster than find—but uses cached information in a database

Command	Description
whereis	Display the location of the binary, source, and man page
which	Show the full path to commands—useful for determining the version of the command in use

Concatenation

CHAPTER 6

Here, There and Everywhere

Many programmers know how to execute a few simple commands on the UNIX OS. The real power of UNIX becomes more evident when commands are combined in a creative fashion. The next several sessions introduce the concepts related to combining functionality of several commands.

Sequencing

In simplest form, you can execute one command per line. However, you can execute several commands in sequence by separating them by semicolons. Note that the input and output of each command is independent of the others in the series.

Redirection

Before UNIX, programs generally had to explicitly connect to the appropriate input and output data. Standard Input (STDIN) is data that serves as input to a program. It is expected to come from the text terminal that started the program. However, you can redirect input instead. Standard Output (STDOUT) is the output from a program. The output by default is directed to the text terminal that started the program, but this can be redirected as well.

A basic example of redirecting STDIN and STDOUT follows:

```
 % echo “GoodByeWorld” > test.txt  # Redirecting STDOUT to a text file
```

```
% read x                          # Reading from STDIN
HelloWorld

% echo $x                         # Displaying to STDOUT
HelloWorld

% read x < test.txt               # Redirecting to STDIN

% echo $x                         # Displaying to STDOUT. See, it worked!
GoodByeWorld
```

Standard Error (STDRR) consists of error messages produced by a program. Its output, like that of STDOUT, is directed to the text terminal that started the program. However, it is a separate stream which is useful to keep errors and diagnostic messages separate from the actual output of a program.

To redirect to standard error:

```
% echo “An error occured” >&2
```

To redirect standard error to a script:

```
% my_script.sh 2> errors.log
```

Both STDERR and STDOUT can be redirected to a file using:

```
% my_script.sh > output.txt 2>&1
```

Appending

To append to the end of a file (when using STDERR or STDOUT), change the > to >>. For example

```
% echo this > test.txt
% echo that >> test.txt
% cat test.txt
this
that
```

Pipes

The output of one program can serve as the input to another one. This is one area where UNIX really shines. Many UNIX users never make the connection that the many small utilities can be combined to very powerful effect. Pipes serve as the connectors between utilities that are combined to perform tasks.

Redirect data to a file and write it on STDOUT at the same time

```
% echo “It’s tee for two!” | tee result.txt
It’s tee for two!

% cat result.txt
It’s tee for two!
```

The pipe redirects STDOUT to STDIN of the next process. The `tee` command reads from STDIN and redirects to STDOUT and the file `result.txt` at the same time

Filters

SORTING/ORDERING

Many commands include options that allow you to sort their output (such as `ls` discussed earlier). If you need to order the results in a command in a way that is not possible for the command in question, you can use the `sort` command. The `sort` command allows sorting that varies with space used, type of element (numeric, alphabetic etc), and location of the element that serves as a key for sorting.

The `sort` command is often used to filter input that is passed to the `uniq` command. `Uniq` removes duplicates from the list of lines passed to it. If the lines have not been previously ordered by `sort`, the output of the `uniq` command will only eliminate duplicate entries that appear sequentially in the list of lines.

The `sort` command includes an option that allows you reverse the output of an other command. To sort on the 6th field first use `-t,` switch to tell sort that the delimiter is a comma and `-k` to tell `sort` which field to sort on:

```
  % sort -t, -k6 file
```

Descending on the same field:

```
  % sort -t, -k6 -r file
```

But there are also several other commands that can be used to reverse output in various ways. To illustrate, lets start by producing a list of zero-padded numbers between one and eleven.

```
  % seq -w 11
```

We can reverse the list by piping to `tac` (which works like `cat` in reverse):

```
% seq -w 11 | tac
```

We can reverse each element in the list using the `rev` command:

```
% seq -w 11 | rev
```

LIMITING

There are a number of different options available to limit the output of files. The `grep` utility and `awk` programming language (that is commonly used to execute one liners) have a broad range of functionality that are well beyond the scope of this book. A few minimal examples will suffice to get you started.

In general, use `grep` to decide which lines you want displayed in a group of processed lines. Regular expressions can be specified to identify a wide range of strings. The following example allows you to identify every line that contains the word *split* in files with a ruby extension.

```
% grep split *.rb
```

The name of the file in question will be displayed before each line if there are multiple entries. If you wanted to list every file that did not contain the word split, you would use the -v option.

```
% grep -v split *.rb
```

The following example uses a regular expression that lists lines that include 2 or 9.

```
  % seq -w 1000 1010 | nl | grep -E ‘2|9’
```

The somewhat cryptically named awk language is derived from the final initials of its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. The `awk` utility can also be used to limit which lines are displayed.

```
  % cat *.rb |awk ‘/split/{print}’
```

But `awk` is particularly useful for quickly breaking up a line into a a series of tokens based upon a delimiter (by default a space). Each token is identified by a dollar sign followed by the number representing the order where it appears (starting with one). The following example lists all of the shell scripts in the directory and displays their timestamp in YYYY-MM-DD format.

```
  % ls -l --time-style=long-iso |awk ‘/.sh/ {print $6}’
```

If you want to use a different delimiter, use the `-F` option and specify the delimiter name.

```
  % echo “one,two,three” | awk -F, ‘{print $2}’
```

ENHANCING

Enhancing might not be the best way to describe this type of filtering, but adding line numbers to output is a common requirement.

```
  % cat environment.rb | nl
```

GROUPING

It is common to want to redirect the output of a number of a group of commands. Rather than individually specifying the redirection:

```
% ls >> output.txt
% df >> output.txt
% who >> output.txt
```

the commands can be rewritten enclosed in parenthesis.

```
% (ls
df
who) >> output.txt
```

The `uniq` utility with the `-c` option provides a count of unique entries. This is somewhat analogous to the use of a GROUP BY in SQL. The following example displays each file owner within the directory followed by the number of files that they own:

```
% ls -l | awk ‘{if ($1 != “total”) print $3}’ | sort | uniq -c
```

Command	Description
grep	Print lines that match a specified pattern
awk	Special purpose pattern matching and data formatting language
sort	Order lines based on specified criteria
uniq	Remove any duplicates from input

Command	Description
nl	Number lines
tac	A version of cat that reads through a file from the end to the beginning
rev	Reverses inputted lines

I Know the Difference

The original `diff` utility used for file comparisons was written in the early 1970's. Since then, numerous enhancements and variations have been created. The `diff` utility displays the differences between two files on a line by line basis. The output of `diff` is in a format that can then be used by the `patch` utility to apply the changes to a file. The `sdiff` utility produces a side by side comparison between two files. The `comm` utility produces three columns of output that show differences side by side—but requires the files to be sorted.

If you simply want to see the side-by-side output of two files to make comparisons for yourself, the `paste` utility can be used. The `join` command can be used to display the lines with a common key field between two files. Although this is useful for simple comparisons, comparisons using key fields are better handled in an RDBMS.

If you are required to reconcile the differences between three files, the `diff3` utility is available. There are a number of GUI programs (`kdiff3`) that can be used to display differences with color-coding and other visual cues. Such programs also include options for comparing entire directories to identify differences.

Command	Description
Diff	Display the differences between two files
patch	Use the output of diff and apply it to a file
cmp	Compare two files
join	Join lines in two files using a key field
paste	Combine two files side by side
comm	Compare two sorted files side by side

Time After Time

With no parameters, `date` returns the current date. You can also manipulate date strings passed in as parameters or in files. For example:

```
% date -d 19700201 +%m/%d/%Y
```

This example might not work on all implementations (I noticed that it does not work using MKS Toolkit which provides UNIX emulation on Windows). See the implementation below using substrings for an alternative. To determine the length of time a command or script takes to execute, you might be inclined to do something like:

```
% date;<my long running command>;date
```

However, you would be better off using time:

```
% time <my long running command>
```

This command returns timing statistics including the elapsed time, user CPU time and system CPU time.

Command	Description
date	Return the system date or manipulate date strings
time	Measure the resource usage and elapsed execution time of a command
cal	Display a calendar

World on a String

The bash shell provides for string manipulation—but in a rather inconsistent manner. Some commands are based on parameter substitution while others require the `expr` command.

The following function receives an input of a date string in MMDDYYYY format. It returns the date in MM/DD/YYYY format.

```
% function slash_date {
    d=$1
    echo ${d:0:2}/${d:2:2}/${d:4:4}
}
```

The second line in the function could also be written as follows:

```
% echo `expr substr $d 1 2`/`expr substr $d 3 2`/`expr substr $d 5 4`
```

The length of a string can be outputted as follows:

```
% expr length “This is twenty chars”
```

A string can be converted from upper to lower case using the `tr` command.

```
% echo “Hello” | tr “[:upper:]” “[:lower:]”
```

A default value can be assigned to a string as follows:

```
% echo ${username-`whoami`}
```

Command	Description
expr	Evaluate an expression
tr	Translate or delete charac-ters in a string

Administration

CHAPTER 7

As mentioned earlier, the focus of this books in not on system administration, and so this section is intentionally brief. It is intended to serve as a point of reference for communication with system administrators or as a stepping stone to further investigation.

There are a myriad of installation options that are dependent upon the particular UNIX variant in use, available hardware, and network configuration. This section will provide a brief, high-level overview of these concerns. The remainder of the book will assume that this work has already been completed—exactly the situation that you will find yourself in when logging into your Internet Service Provider or accessing a system within a corporation's internal network.

Installation

Tools such as `apt-get`, `yum`, and `port` can compile and install software easily. See the accompanying screencast for an example.

Unix/Linux Installation

If you are installing an operating system for the first time, you will probably perform the installation in an interactive manner. If you are going to have to repeat the installation, you should probably consider automating the installation process.

Part of the initial configuration involves the modification of certain system files to set kernel parameters, mount points (`/etc/fstab`), and other system settings.

Besides the base operating system, you will have to install other software (web servers, databases, other language environments, and packages). You will need to become familiar with the installation utilities and package manager (`rpm`) associated with the particular system in use. You may need to build from source using a series of commands that will become all too familiar:

```
% ./configure
% make
% make install
```

Network configuration, including assigning your machine's name and IP address, will follow the initial installation. Prior to placing your system on a network, you need to consider the security of your system. Programs such as `tripwire` can be set up after the initial install and can be used to determine if files are being tampered with by an intruder.

Starting Up / Shutting Down

Administrators need to consider how to start up and shut down the system in a way that will not be disruptive to running processes and users.

The administrator is responsible for configuration of the processes that need to be started. UNIX runs through various *run levels* as the system proceeds through the startup process.

The `shutdown` command is used to shut down or restart the system. A conscientious administrator will warn his user base in advance of system-wide activities like `shutdown` (using the `wall` command or good old-fashioned email).

Users/Group Administration

Administrators are responsible for creating, updating, and deleting users (using the commands `useradd`, `usermod`, `userdel`). Users are categorized into groups, and administrators are also responsible for maintaining these (`groupadd`, `groupmod`, `groupdel`).

System Monitoring

System administrators typically monitor servers to determine the general availability of the system (the server is running and accessible, file system space is available, etc). Web Application Developers need to provide additional information to system administrators if they are expected to intelligently monitor application resources as well.

Backup/Recovery

Perhaps the most important (and least appreciated) job of the administrator is to perform backup procedures that guarantee that a system can be recovered and data restored in the event of a system failure. Archives can be created to back up essential files (`tar`, `zip`, `cpio`) and the `rsync` utility can be used to mirror files on another server. There are also a variety of proprietary tools for performing such tasks.

Somebody's Watching Me...

Monitoring is an activity that is often seen as the system administrator's responsibility. However, web developers have a better understanding of their application's features and requirements. A system administrator might verify that a server is up or that a port is available—but without assistance from a programmer, they cannot (and frequently do not) verify that all needed functionality is available.

System information about running processes can be viewed in the /proc directory using cat. This directory contains special files which provide information about the current state of the kernel. Some of the relevant files that you might want to review:

```
  /proc/cpuinfo
  /proc/meminfo
  /proc/version
```

On some systems you can use the procinfo command to get a summary of this information.

To access a specific bit of information from one of these files, pipe the results to grep and filter on the specific bit of information desired. For example, to see Physical RAM:

```
  % cat /proc/meminfo | grep MemTotal
```

Monitoring network resources can become necessary when developing multi-tiered applications. At minimum, you may need to ping another server to verify its name/ip address and whether it is running at all.

The fuser command can be used to identify processes using files or sockets. The following command returns the pid of the web server running on port 80:

```
  % fuser -n tcp 80
```

The ps command can then be used to obtain more information about the process or the nmap command can be used to verify that the port is open:

```
% nmap --p 80 localhost
```

Other UNIX utilities (`netstat`, `iptab`, `netcat`) are available for network administrative and security concerns.

The `curl` and `wget` utilities can be used in scripts used to monitor web applications or copy web resources. They can be useful for creating web crawlers and automating web application testing. The following is a simple example that could be used to populate a string after a URL returned an HTTP status of 200. If an empty string was found, an error would be reported.

```
#!/bin/bash
result=`curl -sIL localhost:8080/AFASWebConsole | grep ‘HTTP/1.1 200 OK’`
if [ -z “$result” ]; then
  echo “ERROR: HTTP Response 200 expected.”
fi
```

Even though these utilities provide a good deal of functionality, more complex tasks are better addressed through other software. Although submission of forms, cookie management and other features are available, browser automation is not directly possible (but is possible using Watir, FireWatir, or Selenium). Parsing of HTML is better handled using Hpricot or another parsing toolkit.

Command	Description
ps	Display process information
pidof	Returns the pid of a specified process

Command	Description
top	Display process information interactively
vmstat	Display memory statistics report
free	Display information about system memory (e.g. RAM, swap space)
du	Display file space usage. `sudo du -hs /*` will show the directory sizes of the root directories.
df	Display file system space usage
watch	Periodically execute a command
last	List most recently logged in users
lastlog	List the last login time for each user
ping	Send an echo request to a server
fuser	Display a process using a file or socket
nmap	Display network hosts and services
wget	Non interactive download of files from the web
curl	Command line browser typically used to copy URLs

Monitoring File Space

Although monitoring the file space of a system generally is the responsibility of a system administrator, you may need to monitor specific file systems that are heavily used by your application. The following example sends an email if the `/home` filesystem's disk space usage exceeds 90%. The script could be added to a user's `crontab` and scheduled to run on a periodic basis.

```
#!/usr/bin/bash
i=`df | grep /home | awk '{sub("%","",$5);print $5 }'`
if [ "$i" -gt "90" ]
then
  str="`hostname` File System at $i"
  echo $str | mail -s "`hostname` Alert" developer@mydomain.com
fi
```

This type of *one-off* script ceases to make sense when a larger number of systems are in use. You can end up effectively spamming yourself and receiving duplicate emails (for instance if several web application servers report a database server being unavailable). At that point, monitoring scripts might alert a centralized application which sends emails out as appropriate.

Monitoring a Process

The `ps` command can be used to view running processes. To get all information available (all processes and full listing) you can specify a number of different options by including `-ef`. The resulting listing can be limited by piping to `grep`. The process id (PID) is listed and is helpful for other monitoring activities or stopping the process.

The `top` command can be used in an interactive mode to provide a dynamic, real-time view of running processes that is somewhat akin to window's task manager. Various key strokes can be used to modify the display while the program is running.

If you find yourself repeating the same command over and over (e.g. running `ls -l` to check if a file has been created or changed), the `watch` command can be used to periodically execute the command in question. If a file is being changed by another program and you would like to watch the file as it changes (a typical situation when monitoring log files), the `tail -f` command can be used.

A job can be put into the foreground (`fg`), background (`bg`, `Ctrl-Z`, `&`). A signal can be sent to a process—usually to stop it—using `kill`. The `kill -9` option sends a KILL signal that cannot be caught or ignored. Because it does not allow for graceful termination, it should be used sparingly.

Scripting

The Job That Ate My Brain

Normally we think about running programs by executing them at the command line. You can start a process in the background by typing an `&` after the command.

```
% command &
```

There are commands available to move a process between running in the foreground or background. If you do start a process in the foreground, you can stop the process in the background by pressing `Ctrl-Z` and then out it in the background by typing `bg`. You can then type `jobs` to see the job running in the background and `fg` to bring the job back to the foreground.

Command	Description
CTRL-Z	Stop a running job
bg	Put a job in the background
fg	Bring a job into the foreground
<command> &	Run a job in the background
crontab	View or modify the list of scheduled jobs
at	Execute a job at a specified future time
watch	Execute a job periodically and display the output to the whole screen

Here are a few lines from a cron configuration file (the first line is commented):

```
# Run script at reboot
@reboot sh /home/csaternos/bin/rebooted.sh
# Run every 30 minutes
# MIN HOUR  MDAY MON  DOW  COMMAND
30    *     *     *   *    sh /home/csaternos/bin/file_system_alert.sh
```

REVIEW

- cron task (http://www.redhat.com/docs/manuals/linux/RHL-7.2-Manual/custom-guide/cron-task.html)
- cron (http://www.unixgeeks.org/security/newbie/unix/cron-1.html)

Debugging

Shell scripting is an activity that you will naturally come to as you develop sets of commands that can be organized into monolithic tasks. Because the bash shell serves as an interactive interpreter, the commands and syntax you are already using can be leveraged when writing a script. This process is much like the process used by rubyists to solve problems—experiment in Irb (the interactive ruby `shell`) and commit final ideas to a script.

The environment available when running a scheduled script might differ from when it is invoked interactively. If this is the case, you might want to invoke the script with the `-l` option which makes the script act as if it were invoked as a login shell, or source the relevant files to set up your environment.

- ~/.profile
- ~/.bash_profile
- ~/.bash_login

The simplest (and most common) technique for debugging scripts is to use display output using `echo` and redirect output at various points during a script's output for subsequent analysis. The shell might be invoked with a `-v` option (verbose) which prints shell input lines as they are read. However, there are a options available that can minimize these activities using the `set` built-in. The `set -x` option will display each command as it is executed. This enables you to narrow down what is happening at a given point in a script.

By default, if an error occurs, your script will continue processing. This can be problematic if a command fails and its output was expected by a subsequent command. The `set -e` option tells the script to exit as soon as an error is encountered.

The `set` options can be specified within the script like so:

```
% set -ex
```

They also can be turned off (by changing the – to +) if you only want to set behavior for a portion of the script.

They also can be set when invoking a script:

```
  % sh -ex <script_to_debug>.sh
```

Scripts, like other programs, can suffer from performance issues. The `time` command can be used to measure how much time a program takes to complete.

```
  % time <script_to_time>.sh
```

Appendix: Online Resources

CHAPTER 8

LINUX

- The Linux Documentation Project (http://tldp.org/LDP/abs/html)
- UNIX Philosophy (http://catb.org/~esr/faqs/loginataka.html)
- FAQs (http://www.faqs.org/docs/artu/ch01s06.html)

UTILITIES

- sed FAQ (http://www.linuxtopia.org/online_books/linux_tool_guides/the_sed_faq/sedfaq4_005.html)
- top and tail (http://www.brunningonline.net/simon/blog/archives/002192.html)
- Network Settings (http://www.faqs.org/docs/linux_network/index.html)

BASH SCRIPTING

- IBM Article Part 1 (http://www.ibm.com/developerworks/library/l-bash.html)
- IBM Article Part 2 (http://www.ibm.com/developerworks/library/l-bash2.html)
- IBM Article Part 3 (http://www.ibm.com/developerworks/library/l-bash3.html)

LINUX ON ORACLE TECHNOLOGY NETWORK

- Filtering (http://www.oracle.com/technology/pub/articles/saternos-filtering.html)
- Scripting (http://www.oracle.com/technology/pub/articles/saternos_scripting.html)
- Kickstart (http://www.oracle.com/technology/pub/articles/saternos_kickstart.html)

PIPELINES

- Wikipedia on Pipelines (http://en.wikipedia.org/wiki/Pipeline_%28Unix%29)

Appendix: Info Script

```
#!/bin/bash
echo “ “
echo “---------------------------------”
echo “-     S   E   R   V   E   R     -”
echo “---------------------------------”
echo “ “
echo “Node name          : `uname -n`”
echo “Machine name       : `uname -m`”
echo “Operating System : `uname -o`”
echo “Kernel name        : `uname -s`”
echo “Kernel version     : `uname -v`”
echo “Kernel release     : `uname -r`”
echo “Processor Type     : `uname -p`”
echo “Hardware Platform: `uname -i`”
echo “ “
echo “---------------------------------”
echo “-     M   E   M   O   R   Y     -”
echo “---------------------------------”
echo “ “
cat /proc/meminfo | grep MemTotal
echo “ “
echo “---------------------------------”
echo “- P  R  O  C  E  S  S  O  R  S  -”
echo “---------------------------------”
echo “ “
cat /proc/cpuinfo |grep “model name”| nl
echo “ “
echo “---------------------------------”
echo “- D  I  S  K    S  P  A  C  E   -”
echo “---------------------------------”
echo “ “
df -h
echo “ “
```