Mac OS X Terminal Basics v2.1

Neal Parikh / nparikh@freeshell.org / neal@macnn.com

September 29, 2002

1 Table of Contents

2 Introduction

This FAQ is intended to be a quick primer on Mac OS X's BSD Subsystem. The BSD Subsystem is a powerful tool that gives you an immense array of new capabilities and access to a large number of new applications. If you learn to use them wisely, you can do some truly incredible things.

3 Why UNIX?

That is the main question, isn't it? Many people are confused as to why Apple has picked UNIX in the first place. There are several reasons why Apple has picked UNIX to be the core of their new OS (not in order of importance):

1. The historical reason: Mac OS X's roots trace back to NeXTSTEP, and that used UNIX.
2. Developers who may be unfamiliar with the Mac platform will likely have some level of familiarity with Unix, which aids porting efforts.
3. Most users who have studied computer science in either school or college have encountered Unix on some level, and must have some basic familiarity with it.
4. There's a reason almost every server in the world runs Unix. A vast amount of administration software, and a lot of it that runs very stably. A Unix server, properly configured, can be very low maintenance.
5. Having an open source core makes the OS more flexible. For example, there's no way of upgrading the OS 9 version of Personal WebSharing without Apple releasing a software update. If you find that Apache (the world-standard webserver included with OS X) is out of date, you can just go to the website, download the source code, and update that part of the OS. Having much of the core capability written in open source code also allows third party developers to help solve problems and contribute new features.
6. Application availability. As OS X becomes a more and more compatible Unix, more and more Unix applications will become available.
7. Having an operating system that can run consumer applications off-the-shelf as well as run higher-end open source server software is somewhat of a holy grail. It's a win-win situation for Apple, developers, and users.

There may be other reasons as well, but in short, Unix is a foundation that will carry Apple through the next decade or more.

4 What's Darwin?

Darwin is an open source operating system based in part of BSD UNIX and is in the same family as FreeBSD, NetBSD and OpenBSD. Darwin is the core of Mac OS X but it is, on its own, a complete Unix operating system. The Terminal program located in `/Applications/Utilities` allows you to use the Unix core.

To see a history of the development of various Unices, go to *http://perso.wanadoo.fr/levenez/unix/*. NeXTSTEP, Darwin, and Mac OS X are at the very end of the 12-page timeline.

5 Darwin Basics

Q: What is a Terminal?
A: A Terminal is simply a text-based program that is used to send commands to the OS and interact with it. In the case of Mac OS X, the Terminal program allows the user to interact with the BSD subsystem directly.

Q: What can I do with the Terminal?
A: You can do almost anything, because the Terminal is basically a window into another OS. Mac programs now often take advantage of the Unix layer, although UNIX programs rarely take advantage of the Mac layer. In the Terminal, you can run an IRC client like epic or irssi, browse the web with links or lynx, read Usenet newsgroups with the tin newsreader, play Tetris in emacs, write programs, write documents, manage your filesystem, run maintenance and/or system checks, inspect network traffic, handle system administration, and so on and so forth. You can do almost anything in Terminal that doesn't require a full-blown GUI if you're so inclined. In fact, many applications bundled or written for Mac OS X are actually just Cocoa shells for UNIX commandline applications: BrickHouse, a program that lets you configure the built-in firewall, is a shell for ipfw; Virex 7 is a Cocoa shell for the Virex 7 commandline scanner; TeXShop, the program that this document is written in, provides a GUI for accessing the fully standard teTeX distribution of the TeX typesetting language; ProjectBuilder uses gcc, gdb, and other standard UNIX programming tools and wraps them in a GUI with an editor; Cronnix wraps a GUI around the crontab and makes it easier to edit; and so on and so forth. In fact, people are often using Unix at times when they don't really think they are (albeit indirectly). (You can also run Unix programs that require a GUI, but not through the Terminal application. More on this later.)

Q: I've heard it's easy to wreck my system using the Terminal. Is this true?
A: Due to Unix file permissions, users are essentially in a protected space and don't have access to critical system files unless they login as the superuser, or system administrator account (root). You can, however, delete all your own files, but then again, you could do that anyway. You can't really cause any damage you couldn't have caused with the GUI, although one should note that typing a careless delete command is much easier than accidentally dragging all your files to the Trash, selecting Empty Trash, and clicking the OK button. There are no "Are you sure?" prompts (although you can turn these on. More on this later.); the shell assumes by default that you know what you're doing. Being careful should stand you in good stead.

6 Introduction to shells

The shell is, in some sense, the most fundamental user-level Unix program. It takes input from the user, runs other applications, and gives output back to the user. As Unix evolved, however, shells added more features, and they began to differ from each other more. To the novice user, there is no difference what shell you use. They handle some things differently, such as aliases, shell preference settings, shell scripting, job control, command completion, and so on and so forth.

The most popular shells are tcsh, zsh, and bash (not necessarily in that order). tcsh is the default shell in Mac OS X, and bash is the default shell in Linux. zsh is bundled with OS X, and you can try it out by typing `zsh` at the prompt. You can run shells within shells, and just exit out of them in order.

In this document, I'm going to be using tcsh (since it's the default shell), but if you're interested in trying out zsh, the zsh manpage is a good place to start. bash isn't bundled with OS X, but you can install it pretty easily via source or Fink (http://fink.sourceforge.net).

6.1 Aliases

Aliases are very, very useful. Sometimes, Unix commands can get long, and if you use certain commands very frequently, it becomes a bigger and bigger waste of time to keep retyping them. This is where aliases come in. Let's say I make and close lots of Terminal windows throughout the day, so I want to make exiting shells quicker. I could make an alias for the exit command so I could just type `x` and have the window close. In tcsh, the syntax for this would be:

```
alias x exit
```

That's it. If the command you're aliasing is more than one word, you can put single quotes around it, like this:

```
alias server 'ssh neal@server.example.com'
```
[1]

In order to ensure that your aliases (and other settings) are available whenever you open a new Terminal window, you put them in a shell startup file. Shell startup files (as well as startup files for many other Unix applications) are hidden files (they begin with a .) that end with rc. The tcsh startup file is `.tcshrc`, and it's stored in your home directory. (Similarly, you might have `.zshrc` or `.bashrc`.) Don't worry that this file isn't there by default; we'll get to creating it later.

[1] SSH (http://www.openssh.org) is a secure, encrypted remote login program – an alternative to telnet.

7 Running system commands

To run a system command, just type its name at the prompt. **Warning**: There is no undo in UNIX. For example, if you tell the `mv` command (essentially rename) to rename a file to another existing file, it will just get rid of the first file. You can overwrite entire directories if you're in as root and use careless syntax. The bottom line is to be very, very careful about syntax (especially when moving, copying, or deleting – mv, cp, and rm respectively) and to (a) check the man page for the command and (b) try the command out on some test files to make sure you don't mess up something important. Wiping a blank folder will be much better than wiping your applications directory, obviously. So experiment, but be careful, and if you're in as root or you use the sudo command to run something as root – **think twice**.

A reader suggested using shell aliases to bypass this problem of having no "Are you sure?" prompts. Shell programs allow you to create aliases for commonly used commands, and you write these aliases in a file located in your home directory called `.tcshrc`. This file doesn't exist by default, so you have to create it. If you're not familiar with Unix text editors yet (we'll get to this later, too), here's how to go through it. Open a new Terminal window, and type `pico` and hit return at the prompt. You'll be confronted by an odd text-based interface, which is the interface to the Pico text editor. Paste in the following lines normally, then press control-O. It will prompt you for the filename, and now you can enter `.tcshrc`. Press control-X to quit pico.

```
alias rm ‘rm -i’
alias mv ‘mv -i’
alias cp ‘cp -i’
```

Now quit and relaunch Terminal. Now, when you use mv, cp, or rm (explained below), the shell will give you a prompt making sure you want to do what you say you did when you ask to delete or overwrite files. Note that if you want to remove an entire directory, the shell will now prompt you for each and every file in the directory. If the directory has a couple hundred files, this can be extremely inconvenient. Unix applications generally have commandline 'flags' or 'switches' that tell the program to run with or without certain options. The `-i` flag above tells the utilities to prompt before overwriting. If you want to temporarily disable this, use the `-f` flag to *force* deletion.

Command	Usage	Function
`man`	`man [commandname]`	Read manual page for `[commandname]`.
`man -k [topic]` searches all manpages for [topic].		
`ls`	`ls [dir]`	Lists files in [dir]. Leave [dir] blank for current dir.
`ls -alF` shows hidden files (-a), provides extra info (-l), and graphically shows type of file (-F).		
`pwd`	`pwd`	Prints path to current directory you're currently in.
`cd`	`cd [dir]`	Changes working directory to [dir].
`ps`	`ps`	Shows processes you're currently running.
Generally, `ps aux` is used for a complete process listing. Read `man ps` for details.		
`top`	`top`	Interactive display of system stats and processes.
`less`	`less [file]`	View [file], using arrow keys to navigate. q to quit.
`mkdir`	`mkdir [dir]`	Makes a directory called [dir].
`rmdir`	`rmdir [dir]`	Deletes [dir]. Equivalent to rm -r.
`chmod`	`chmod [privs] [files]`	Changes [file]'s permissions to [privs].
Format of privileges is complex. See below and/or `man chmod` for details.		
`chgrp`	`chgrp [group] [file]`	Changes [file]'s group to [group].
`chown`	`chown [user] [file]`	Changes [file]'s owner to [user].
You can set a user and group in one command by doing `chown [user]:[group] [file]`		
`grep`	`grep [pattern] [file]`	Searches for [pattern] in [file].
Ex: `grep ''Mac OS X'' unix.txt`		
`mv`	`mv [file1] [file2]`	Moves [file1] to [file2]'s location.
`cp`	`cp [file] [file.copy]`	Copies [file] and calls copy [file.copy]
Use `cp -R [dir1] [dir2]` to copy an entire directory.		
`df`	`df`	Prints out disk usage data.
`du`	`du -ms [file]`	Calculates the filesize in MB of [file].
`du -msc *` will calculate the filesizes of everything in the current dir as well as a total.		
`tar`	`tar cfz archive.tar.gz [dir]`	Creates an archive called archive.tar from [dir1].

This is basically what you need to get on your way. There are so many UNIX system commands that more would be unnecessary. If you want to know about other commands (many are located in `/usr/bin` and `/bin`), use the man pages, ask in MacNN's OS X UNIX forum, or visit Mac OS X Hints.

8 Basic shell customization

Now that you know how to run all these commands, you're going to want to make your shell work the way you want. Shells nowadays are pretty customizable, and these are some settings you can change in tcsh that you may find useful. tcsh variables are set in `.tcshrc`, just like aliases, and follow this format:

```
set setting=value
```

For example, if you wanted to make your prompt just a >, you would put this in your `.tcshrc`:

```
set prompt="> "
```

(The space at the end is desirable at the end of prompt settings, otherwise the beginning of your commands would become very hard to read.) Some other useful options are the following:

`rmstar` – If you set this to 'on,' the shell will ask you if you're sure when you run the `rm *` command. This is a useful countermeasure to typing too fast for your own good, and it's definitely a desirable thing to have in the root `.tcshrc`.

`prompt` – You can set this to almost anything, but there are a number of useful variables that can help you make a really nice prompt. See the tcsh manpage for a full list. Prompts become fun when you color parts of them, but this is more complex a topic than I can discuss here.

`watch` – This notifies you at the shell when users log on or off. If you use `set watch = (any any)`, you'll be alerted to all logins/logouts on the system. tcsh will tell you the user, login device, and IP address from which the login took place. You can also watch specific users and groups; see the tcsh manpage for more details.

`rprompt` – This takes all the same parameters as prompt, but it's right justified. Some people like putting the time in their rprompt (%p). If you type a very long command, the rprompt will vanish and let you type over it.

`complete` – Setting this to "enhance" lets you tab-complete without having to properly capitalize the beginning of the filename. This is convenient on OS X especially, because so many directories are capitalized (like the ones in your home folder).

`autolist` – If the autolist variable is set, the shlel lists the remaining choices (if any) whenever completion fails.

`implicitcd` – If set, the shell treats a directory name typed as a command as though it were a request to change to that directory. If set to verbose, the change of directory is echoed on the screen as it happens. Changing directory takes precedence over executing a similarly-named command, which can become annoying, but it's an interesting option nevertheless.

`edit` – This is an interesting one. This enables "Command-line Editing," which basically allows you to edit the commandline the way you'd edit a document in emacs or vi. If you become a fan of either of those editors, this is a fun one to try out. (Note, however, that by default the shell uses Emacs keybindings, and you'll have to change them if you want to use vi editing.)

These are just some of the options available to you in tcsh, and they were chosen just to give a taste of the customization possible. The tcsh manpage has a complete list of settings, and it's worth your time to sit down and read through them at some point.

9 Permissions

Every file or directory on a Unix operating system has permissions settings. Permissions settings tell the system what access privileges the owner of the file, the group of the file, and all other users have. The standard permissions are simply Read, Write, and Execute. Read privileges allows viewing the contents of a file, Write allows updating the contents of a file, and execute allows executing an application.

When you do a long directory listing (`ls -l`), the output for a given file will look like this:

```
[neal@localhost /etc/httpd]% ls -l httpd.conf
-rw-r--r-- 1 root wheel 36910 Jan 2 01:09 httpd.conf
```

The -rw-r–r– at the beginning describes the permissions. The first character actually doesn't describe permissions, but rather the type of file. For example, a directory's permissions might be drw-r–r–, and a symbolic link's could be lrw-r–r–. The following nine characters then, describe the permissions. The first 3 (rw-) describe the permissions that the owner of the file has (in this case, 'root'). root can read and write to this file (which, in the case of text files, is all that is needed). The second three (r–) describes what the other members of the group 'wheel' can do to the file (all users designated as Administrators are in the group wheel); namely, they can read it, but not modify it. The final three (r–) describe what all other users on the system (that is, ones that are not in the group 'staff') can do, which is also read, but not modify. If you viewed the permissions on a program, say, 'ls' itself, the permissions would probably look something like this:

```
[neal@localhost /bin]% ls -l ls
-r-xr-xr-x 1 root wheel 27160 Dec 20 20:56 ls
```

This means that everyone can execute the file (you must be able to read the file in order to execute it), and no one can modify it.

(Note that there are some other permissions besides read, write, and execute, but given their rarity I will not cover them here. See man chmod for details.)

A **large** number of the problems you will run into when using a Unix operating systems will involve files not having the correct permissions. Therefore, it is very important to know how to change file permissions quickly and efficiently. Not surprisingly, you use the `chmod` command (change mode). The syntax for chmod is:

```
chmod [permissions] [file1] [file2] ...  [file n]
```

So, to give the file 'blah' read and write permissions for the owner, you would type:

```
chmod u=rw blah
```

Similarly:

```
chmod g=rw blah
chmod o=rw blah
```

would change the permissions to 'rw' for the group and other, respectively. However, there's no need to type out 3 commands if you want everyone to have read/write permissions:

```
chmod a=rw blah
```

'a' stands for all. Also, sometimes you don't want to modify existing permissions, but merely add write access for everyone:

```
chmod a+w blah
```

This will not modify the read or execute bits, but will add write access to all files given as arguments. Similarly, you can do:

```
chmod a-w blah
```

to remove write access for everyone. There are also numerical shortcuts for certain frequently used modes (actually, for all modes, but not many people remember many of them). A few examples:

`chmod 777 blah` – changes to rwx permissions for everyone.
`chmod 755 blah` – changes to rwx for user, r-x for everyone else.
`chmod 700 blah` – changes to rwx for user, nothing for everyone else.
`chmod 666 blah` – changes to rw- for everyone.

Finally, you can use `chmod -R [modes] [dir]` to recursively change permissions to [modes] for [dir] and all files inside it. If you're curious and want to know about the less common modes like the sticky bit or setuid/setgid, read the chmod manpage.

10 Running Programs

For budding UNIX developers: if you have a source file called hello.c, you compile it by typing `cc -o programname sourcefile.c`. This makes a file called `programname` in the same directory. To run it, type `./programname`. (Note: you do need the Developer Tools installed to be able to compile anything. I strongly recommend you install the Dev Tools because you can't compile UNIX applications (for installation, since most UNIX applications are distributed as source, not binaries) without them – and if you're a developer, then the reasons are obvious.)

Additionally, you can use ProjectBuilder to write your commandline application if you're not fond of Emacs or some other text-only editor. Open ProjectBuilder (`/Developer/Applications/`), go through the basic configuration screen if you haven't already, then press Command-Shift-N or File - New Project, and then choose Standard Tool. Note that while PB is a fine environment for writing even CLI apps, the Run environment is fairly buggy, and you'd do well to have a separate Terminal

window open to run the program.

To run shell scripts (end in .sh) like the LimeWire installer (Mac OS X Gnutella client), you type `sh whatever.sh`.

11 What's NetInfo?

NetInfo is a directory system for system settings. Some people have said it's a bit like the Windows registry. It stores such things as the users of the system, and basic networking information. Mostly, you shouldn't need to know about it. You can poke around with it in NetInfo Manager, or using the niutil command (`man niutil`).

NetInfo is generally quite thorny and not heavily documented, and of little use to average users or even most developers. More information on NetInfo can be found at

http://www.macaddict.com/osx/xphiles/index.html

Additionally, Apple has recently published a document entitled *Understanding and Using NetInfo*, and although the actual usage of NetInfo is primarily targeted at system administrators running Mac OS X Server 10.x, the document is still the best source if you're just trying to understand what kinds of things NetInfo is used for, and why it's useful.

12 Basics of Compilation

http://fink.sourceforge.net/doc/porting/index.php is a good primer on getting started porting CLI apps. Even if you know what you're doing, porting apps can be extremely difficult, so I won't say much. Most apps, however, compile just fine – here's a quick primer:

Assuming you've downloaded and decompressed the source code:

1. Change to the directory containing the source code.
2. Read the README and INSTALL files with pico. Type "pico README" or "pico INSTALL" to read the files. Use your arrow keys to navigate. Control-X will quit the program and have you back to the prompt. pico lists some shortcuts for page up, page down, etc at the bottom of the screen. Learn to use pico to some extent; you'll need it at times.
3. Once you're out of pico, you need to copy Apple's custom configure files to the folder, so the installer script will recognize which UNIX you're using (Darwin isn't too popular yet). Type:

   ```
   [prompt] cp /usr/libexec/config.* .
   ```

4. Then run the configure script:

   ```
   [prompt] ./configure
   ```

5. Assuming there were no errors, compile the program:

   ```
   [prompt] make
   ```

6. Finally, install the program. This always requires root access.

   ```
   [prompt] sudo make install
   ```

   If this is the first time you've used sudo, it'll give you a little lecture and then prompt you for your password. Enter your user's password. It should start installing. Finally:

   ```
   [prompt] rehash
   ```

7. At this point, I usually update the locate database in case I need to search for any of these files:

   ```
   [prompt] sudo /usr/libexec/locate.updatedb
   ```

   Then you can just type "locate whatever" to find something with "whatever" in the path.

Often, you can get strange and very annoying errors, and if you do, post them at MacNN's OS X UNIX forum or ask a friend. It's often convenient to bounce errors off people to get some ideas.

However, the Fink Project has done a fantastic job with their third-party package manager for Mac OS X, and Fink removes almost *all* the hassle of installing Unix programs. The Fink website (http://fink.sourceforge.net) is well worth the visit, and I'd recommend using Fink to install almost anything at this point.

13 Process Management

Every app you run spawns a new process and is given a PID (process ID). For right now, we can just cover killing off apps (sometimes Force Quit doesn't do it). Let's assume OmniWeb is the offending app for right now (random choice). Type:

```
[prompt] ps auxc | grep Omni
```

This is a piped command. What it does is that the shell first executes ps aux, and then uses the output of that command as input for the command `grep 'Omni'`. The grep command is basically a search through text tool, so it will basically search through the listing of processes for any app with "Omni" in the name. Hopefully, you'll get OmniWeb's listing pop up, along with it's PID. To kill it, just type `kill PID` and insert the PID number. For example, `kill 419`. That should knock it off. If even that doesn't work, you could use `kill -9 419`, which basically means "kill with extreme prejudice". If that can't quit it, nothing can. Process Management can be quite complex, if you have several suspended tasks running (with bg and fg and related utilities), but if you don't spend a lot of time with UNIX, those commands aren't very useful. If you're running a UNIX program that you don't know how to quit, type Control-C to quit it.

(Note: If you're wondering why I used "ps auxc" and not "ps aux", it's because piping ps aux into a grep statement will also list that grep command as part of the output, which is annoying. Also,

The 'c' option displays just the process name without the entire path, which ensures that 'grep' will find the desired text.)

14 Introduction to Text Editors: Pico, Emacs, and Vi

As said before, pico is a text editor. It's the friendliest and easiest to use text editor in UNIX. Text editors become very, very powerful but they also become very, very complex. In this FAQ, we'll cover the basics of pico, emacs, and vi, the standard triumvirate of Unix text editors.

14.1 Pico

pico is easily the simplest and most user-friendly of all the UNIX editors, and as a result, most users who are new to UNIX choose pico as their editor of choice. To launch pico, simply type:

```
[prompt] pico (filename)
```

The (filename) implies that providing an input file when launching pico is optional; if you just launch pico normally it'll open a new document. The basic commands are Control-O for save, Control-X for quit, arrow keys for navigation, Control-W for find, and Control-G for the quick and excellent built-in help system. The help will walk the user through all the other commands available in pico (cut/copy/paste, etc), but for most simple documents, those few commands should be enough for basic literacy.

Pico comes with no built-in functions for search-and-replace, which is one of its major flaws. However, there is no need to learn Emacs or Vi if this is the only feature you're missing; GNU has written nano, which is basically a clone of pico, but with a few more features that don't sacrifice ease of use:

Goto line and replace functions (without needing command-line flags).
Interactive replace function and spell checker.
Auto-indent support.
Variable displayed tab width.
Regular expression search and replace.

14.2 Emacs

Emacs is an amazingly large subject, and for a complete reference, the GNU Emacs manual serves very well: *http://www.gnu.org/manual/emacs/*. However, learning the few basic commands in Emacs is not very difficult, and can serve you well when Pico is not up to the task and the GNU manual is overkill.

Firstly, a quick briefing of the Emacs "GUI" is necessary. The command-line version of Emacs basically has buffers and windows, and you use the minibuffer to communicate with the editor and issue various commands. When you have "opened" a file in Emacs, what the editor has actually done is load the contents of the file into a temporary buffer (until it is saved to disk), so the actual file is not being edited. You have one buffer per file, but you can split the display so that there are two windows looking at different parts of the same buffer. Alternatively, you can have two windows up looking at different buffers, of course. You can have an infinite number of windows, although more than four or five gets extremely unwieldy. Having a dozen buffers open, however, is often rather convenient, as jumping between them is fast and easy.

Also, some explanation of Emacs' feature set is necessary. Emacs is organized by "modes." Basically, for different kinds of editing, there are different modes you use. There are major modes and minor modes. The main difference is that you can only have one major mode enabled at a time, while you can have several minor modes enabled. On the whole, major modes don't overlap, while minor modes provide functionality useful in many different contexts. For example, for editing text, you use Text Mode. For writing Java code, you use Java Mode. For writing C++ code, you use C++ Mode. And so on and so forth. There are Lisp modes, C modes, and modes for almost any significant language. In different modes, Emacs automatically changes various settings to make coding in that specific language more streamlined. However, there are also Calendar modes for viewing and organizing your calendar, Diary mode for using the built-in organizer, debuggers, makefiles, and so many others that it would be impossible to cover them all.

However, since Emacs has so many features, and the mouse is unsupported, there are a ridiculous number of key bindings. There are so many key bindings, in fact, that some of them use Control keys (C-h means Control-H in Emacs syntax), while others use the Meta key (M-x means Meta key-X in Emacs). In Mac OS X, the meta key is simply the Escape key (however, in Mac OS X 10.1, there's an option to set the option key as the meta key; for now, though, stick with Esc). Also, even this number of key bindings weren't enough, so they had to have **multiple key bindings**. For example, typing C-x C-c means hold down Control, then type X, then C (holding down control the entire time). That quits Emacs. However, this also implies that C-x C-b is different from C-x b. In the former, Control is held down while you press B, while in the latter, you release control after pressing X but before pressing B. These commands do completely different things, also. C-x C-b lists all the buffers Emacs has at that time, while C-x b lets you switch buffers (the minibuffer will prompt you to type in the name of the buffer, although you can use Tab to autocomplete). Meta keys work the same way, on the whole. However, there are a large number of commands in Emacs that aren't even assigned key bindings, so you have to type out rather long commands, in some cases. For example, M-x compile is the command you can use to compile something within Emacs; M-x gdb is what you use to load GUD and run gdb on a binary within Emacs; M-x copy-region-as-kill will copy a region of text; and so on and so forth.

Some useful commands are C-x C-s for save, C-x C-f to visit (open) a file, C-x o to switch windows, C-x 1 to close all other windows, C-x 2 to split a window horizontally, C-x] for end of file, C-x [for beginning of file, C-a for beginning of line, C-e for end of line, C-k for kill line (kill is like cut), C-y to yank (paste) whatever you last killed, C-space to mark the beginning of the region, M-x copy-region-as-kill to copy an entire region (from the mark to the current location of the cursor), C-x C-b to list all buffers, C-x b to switch buffers, C-x k to kill the current window, C-x C-z to suspend Emacs, and C-x C-c to quit. Some commands for switching major modes are M-x text-mode, M-x

c-mode, M-x calendar, M-x diary, and so forth. C-h stars the built-in (and extremely extensive) help system, and C-h a is just one of the options in it (apropos, which is the most convenient). For example, if you want to find out how to launch an interactive shell within Emacs, you could type C-h a shell, and this would tell you that M-x shell opens up a shell.

While this is anything but an in-depth look at Emacs, it should give you the fundamentals that lets you navigate around the environment (because it's so much more than an editor, really) and be on your way to learning your way around Emacs. For any additional information, the GNU Manual should be all you need (it has in-depth coverage of everything from saving and quitting to using the powerful built-in versional control system).

14.3 Vi

Vi is written in a very interesting way. It's almost entirely incomprehensible to someone just starting out with it, but veterans can work faster with it than they can think. Most standard operations can be performed by just pressing one or two keys, which is hard to learn, but very fast to use once you remember. There's no real reason to bother learning it if you're just going to be using Unix on your OS X machine, since Emacs is much easier and just as powerful; however, if you're going to be using numerous Unix machines (e.g. in a university environment), it's almost necessary to know how to use Vi, since it's available on every Unix machine, while Emacs is not.

First, launching and quitting. To launch vi, just type `vi` at the prompt. This will launch into a screen with a number of symbols down teh left side. To quit, type `:q` (that's colon, and then q) and then press return.

There are very few commands you really need to get on your way, but before we get right into it, it's important to understand the way vi works. It uses a modal system not unlike the one Emacs uses, but is more simplistic in its implementation. First, the last line of the screen is the **status line**, where you give commands to vi and vi gives information back to you. Second, vi is a "modeful" editor, which means that you are either giving commands or entering text. You must be in the right mode in order to do either. When you first launch vi, you're in command mode. Any insert command will switch you into edit mode, and at this point only the Escape key will take you out of edit mode into command mode. When you're in edit mode (or insert mode), vi will show

```
-- INSERT --
```

in its status line (if it does not, press ESC twice and then type `:set showmode`).

Next, actual operation. The cursor keys are used to move around. Additionally, the following standard commands are available:

`a` – append new text after the cursor.
`i` – insert new text before the cursor.
`o` – insert a new line below the line you're currently on and start entering text.
`O` – insert a new line above the line you're currently on and start entering text.

`x` – delete the current character.

These are the core editing commands. Additionally, the following commands are useful so editing text doesn't turn into an arcade game session:

`yy` – copy the line the cursor is on.
`p` – paste the copied line after the line the cursor is currently on.
`dd` – delete the entire line the cursor is currently on.

Additionally, many commands in vi can be given arguments that tells vi how many times to execute the command. For example, typing '3x' would delete 3 characters instead of having to type x three times. Typing 80dd will delete the next 80 lines of your document.

Saving files:

`:w` – save the file
`ZZ` – save and quit
`:wq` – save and quit (equivalent to ZZ)
`:q!` – quit without saving

Searching and Replacing text:

`/text` – search for the next occurrence of the string "text" in the file.
`:s/old/new` – replaces the first occurrence of old with new on the current line.
`:s/old/new/g` – replaces all occurrences of old with new on the current line.
`:10,20s/old/new/g` – replaces all occurrences of old with new on lines 10 through 20.
`:%s/old/new/g` – replaces all occurrences of old with new in the entire file.
`:%s/old/new/gc` – same as above, but vi will ask for confirmation before replacing.
`n` – repeats previous search
`.` – repeats previous replacement

Vi also has more advanced pattern matching with search and replacement features, as well as full regular expression support, but a beginner's manual is hardly the place to explain such features.

Vim also offers additional features such as syntax highlighting and visual text selection, but since it isn't included, I won't spend time on it here. Vim's benefits are summarized here:

http://www.vim.org/why.html

And can be downloaded here:

http://www.vim.org/download.html

All the above commands work in both Vim and Vi.

14.4 BBEdit

BBEdit isn't a Unix editor, but it has *great* Unix support. You can edit hidden Unix files and files you don't have permission to in BBEdit, and you can even open Unix files with BBEdit from the commandline once you install the BBEdit Unix support package. Although learning a Unix editor is a necessary part of learning the Unix operating system environment, BBEdit can be a very useful tool - especially for novices - as well, and should not be forgotten.

15 Introduction to X Windows / X11

Q: What is the X Windows System?
A: The X Windows System is a graphical user interface framework for Unix and Unix-like operating systems. It's a little bit like Aqua, but uglier and more customizable. `:-)`

Xfree86 is by far the most common and most popular. Neither Darwin nor Mac OS X include an X11 distribution (like Xfree86, which is free, as the name implies). You can basically run any of numerous "window managers" for X Windows; a window manager is basically a complete, self-contained GUI. Different GUIs have different styles, different menu management and are more than just themes for the same interface – rather, they're separate UIs altogether.

Q: Why do I need X11?
A: Any UNIX apps with a GUI (such as the GIMP) require X11. If you're not interested in running any of these programs, then you don't need X11.

Q: What are these "window managers" I hear about?
A: Window Managers run on top of the X Windows System, providing a much more robust and usable GUI. Some are more complex than others. The more famous window managers are AfterStep, WindowMaker, and Enlightenment. Gnome and KDE are desktop environments and run on top of (underneath?) a window manager. The window manager handles how windows are drawn on screen - how the title bar looks, etc. There's a lot more than that in Gnome and KDE.

Reader Esme Cowles offers the following explanation: "Desktop environments run on top of the window manager, in that they usually handle the task bar, program menu, etc. They also usually run services in the background, too.

"For window managers, I think it's important to say that the inside of the window is controlled by the program, but the borders and placement is handled by the window manager. Most people who aren't familiar with UNIX are really surprised by this, so I usually explain that it's so an application can be running on one machine, with it's display on another machine, but still have the window borders and stuff fit in on the display machine."

Q: How do I get an X Windows System installed on Mac OS X?
A: There are two ways: Download and install the XFree86 binaries from www.xfree86.org or buy a prebuilt X Windows from Tenon (www.tenon.com), Xtools. Xtools is extremely overpriced, but

lets you run X windows side-by-side with aqua windows - they behave like normal Aqua windows and you can have, say, the GIMP running next to OmniWeb. XFree86 with the patch from http://mrcla.com/XonX lets you run XFree86 side-by-side with Aqua, and now, rootless patches are available so you can run X Windows applications and Mac OS X applications side-by-side. These patches should be rolled into Xfree86 4.1 fairly soon, and once these rootless patches mature, running X11 programs should become more and more convenient in Mac OS X.

Quite frankly, if you want to use X Windows on Mac OS X, Fink is the path of least resistance (Tenon's solution is probably easier, but XFree86 is more standard. Also, let's not forget that XFree86 is free, while Tenon's product costs money.)

More information on X11:
http://www.xfree86.org
http://www.mrcla.com/XonX
http://xwinman.org
http://fink.sourceforge.net

16 Conclusion

That's all for now; I hope you found this useful. I'd love to hear about your experiences with the FAQ, or anything you'd like to see improved or added. My primary email address is nparikh@sdf.lonestar.org, and you can also reach me at neal@macnn.com. My contact information is also kept up-to-date on the first page of this document.

This Terminal Basics FAQ will be (and has been!) updated via user comments and is kept up to date at the following locations:

http://homepage.mac.com/rgriff/TerminalBasics.pdf
http://nparikh.freeshell.org/pub/faq/TerminalBasics.pdf
http://neal.macnn.com/pub/faq/TerminalBasics.pdf

The Terminal Basics document has been written and typeset entirely in LaTeX with TeXShop.