

An Introduction to UNIX¹

George Ferguson
ferguson@cs.rochester.edu

July 9, 1999

¹UNIX is a trademark of Bell Laboratories.

Chapter 1

Introduction

This document describes the UNIX system for the Computer Science Department of the University of Rochester. Most of the description applies to any UNIX system, except for locally installed software which may not be present everywhere. The document starts with the fundamentals of getting logged on and setting up an account, and then goes into more detail on various topics of common interest. This document is meant to get you started, not teach you everything there is to know about UNIX in one go. It provides brief descriptions of or pointers to many of the features of the system as well as describing the fundamental concepts of UNIX from a user's point of view.

Chapter 2

Getting Started

2.1 Your Account and Your Password

You are identified to the system by your *login name*, a short identifier generally related to your last name.¹ In order to log in, you type your login name at the login prompt of an idle machine.

The first thing you should do as a new user is set your password. By default you have no password, which is a security hole for everyone on the system. To do this, run the program `passwd` by typing its name (followed by hitting the `<Return>` key) It will prompt you for your desired password, which must be at least eight characters and cannot be all the same case. From now on you will be prompted for your password after typing your login name. You can change your password anytime. It is a good idea to do this regularly.

You have just run your first UNIX command. Just to get things straight, note that in general you run a command by typing its name (see below concerning the filesystem) followed by pressing `<Return>`. From now on we will simply use the term “run the command” to refer to this procedure. In general, UNIX commands accept *arguments* to modify their behaviour. These are generally typed after the command name although there are exceptions. Often arguments which modify the behaviour of a program (sometimes called *flags*) are introduced with a leading hyphen (or minus sign, “-”) to distinguish them from *filenames* which specify data for a pro-

¹Actually, you are really identified by your *uid*, a small integer, stored in the password file `/etc/passwd`.

gram. Again, UNIX is full of exceptions and special cases, but this is a general rule of thumb. When describing commands and arguments below, we will use **boldface** for the command name and for flags, and *italics* for items that you would supply, such as filenames.

Now that we've told you how to run commands (well, at least one command), we had better tell you how to stop commands if they get out of control. If you want to terminate the current command, you should "interrupt" it by hitting `<Control-c>` (that is, hold down the "Control" key and hit "c"). Usually this will return you to your shell (to be described soon) where you can type your next command. Note that some programs "trap" the interrupt, that is they refuse to be killed. There are ways of dealing with such recalcitrant processes, but usually the programs that do this provide better ways of quitting them anyway.

Additional commands that you may be interested in in connection with the password process include `chsh` and `chfn`. The former is described in the section on the shell, the latter in the section on network utilities.

Finally, a word about getting off the system when you're finished. To logout, just give the `logout` command.

2.2 Getting Help: The Manual

Before going any further, it makes sense to describe how to get help. UNIX provides a help facility based on viewing pages of the UNIX Programmer's Manual. To get help about a command, you should type

```
man command
```

Unfortunately, the UNIX manual (as its name implies) was written by programmers, for programmers, and can be fairly cryptic. If you don't know the name of the command you want, but have some idea of what it does, you can use

```
man -k keyword
```

which will give a brief listing of items which match the topic. You can then look those items up individually.

Manual pages (often called "man pages" generally follow a standard layout. First is the one-line description of the command (NAME), then a

short listing of possible arguments (SYNOPSIS), followed by a more detailed discussion of the arguments (OPTIONS) or the program as a whole (DESCRIPTION). The end of a man page typically contains information about where relevant files are stored (FILES), references to other man pages (SEE ALSO), shortcomings of the program (BUGS), and the program's author (AUTHOR). Usually the information you need is in a man page somewhere; you just have to persevere long enough to find it.

Chapter 3

The Shell

In UNIX, there is a program which is started for you by the system when you login. It is called your *shell* and interprets your commands to the system to allow you to run other programs. You have already interacted with the shell when you set your password or read the manual. Since the shell is just another program as far as the system is concerned, there are several shells available and you can choose one that you like. Many shells provide additional functionality beyond simple command interpretation; most even provide a programming language for writing shell programs (called *scripts*).

The original UNIX shell is called `sh` (usually pronounced by spelling it, “ess-aitch”), also commonly referred to as the Bourne shell. Many system utilities use or are written for `sh`. The most commonly used shell is the C shell `csh` (usually pronounced “seesh”). It provides several features which `sh` lacks. We will describe it in more detail in the next section. Other choices of shell include the Korn shell `ksh`, and the CMU C shell `cmucsh`. These are described in more detail in their man pages.

To change your shell, issue the command

```
chsh login-name
```

where *login-name* is your system login name. You will be prompted for the full pathname of your new shell. Table 3.1 lists the available shells and their full pathnames (see also below concerning the filesystem for just what a pathname is). Note that you have to login again for the change of shell to take effect. You can also check on the system’s notion of your shell by looking at the password file, `/etc/passwd`.

Shell	Description	Path
sh	Bourne Shell	/bin/sh
csh	C Shell	/bin/csh
cmucsh	CMU C Shell	/usr/grads/bin/cmucsh

Table 3.1: Available shells

You can tell which shell you are using by looking at the prompt. By default, the C shell's prompt is percent sign (“%”) while the Bourne shell's is a dollar sign (“\$”).¹ Often in UNIX documentation, examples of commands have such a prompt to identify which shell they are referring to, although many commands work for either. We will follow this convention in what follows; you should not retype the prompt when trying the examples.

3.1 The C Shell

The C shell is the most commonly used shell and so merits some extra discussion since you use your shell far more than any other program. Unfortunately, it is a mishmash of features and bugs, which makes it a little hard to describe. If this section seems confusing, then try to reread it and experiment with the concepts using your account.

3.1.1 The `.cshrc` file

In order to allow users to customize the behaviour of the C Shell, it reads the file `.cshrc` from the user's home directory every time a shell starts. This includes when the user first logs in, any shells started in a windowing environment or from remote logins (see below), and any shells started as “escapes” from within programs.² Also, any shells started to execute scripts read this file. The `.cshrc` file can contain C shell builtin commands and commands to execute regular (external) programs.

Since it is read so often, it is important that the `.cshrc` file not contain extraneous information. The general rule is that things which only need to

¹You should be aware that people often change their prompts from the default. Sometimes they follow this convention, sometimes not.

²Starting a new shell is sometimes called “forking” a shell, from the low-level system call used to create new processes.

be done once belong in the `.login` file to be described in the next section. Even besides this distinction, not all shells are created equal. The shells which are actually being used as command interpreters reading from a terminal are referred to as *interactive* shells, and there is a mechanism to allow certain things in the `.cshrc` to be done only if the shell reading it is interactive.³

The most important thing to have in your `.cshrc` is code to set your *path*. This is a list of directories (see below) where the shell will search for programs to execute if it cannot find them in the current directory. This allows programs to be gathered in standard places and removes the need for you to type the program's entire pathname. You can set your path by having a line of the form

```
set path = (dir1 dir2 ...)
```

in your `.cshrc`.⁴

In addition, you will typically set any *aliases* in the `.cshrc` file so that they are always available. Other things that might go into a `.cshrc` file include setting shell parameters (such as `history`), setting the prompt, and setting environment variables that cannot be set in `.login`.

3.1.2 The `.login` file

Like the `.cshrc` file, the `.login` file is read before the shell starts, but in this case only once, after you've logged in and your `.cshrc` has been read for the first time. You should put once-per-session commands here, such as starting background processes (such as news and mail watchers) and setting various variables which are inherited by other shells (environment variables describing your login session). As well, in a workstation environment one typically starts the windowing session from the `.login` file (see below concerning X Windows).

3.1.3 Variables and Wildcards

The C shell provides variables, like a programming language, which you can set and inspect. Some variables, like `path` described above, have spe-

³Namely, test if the prompt is set using the `if ($?prompt)` control construct.

⁴See the next section about Editors to find out how to create or edit your `.cshrc`.

cial meaning to the shell. Others you can create and use for your own purposes.

You set a variable with the syntax

```
% set var=value
```

If *value* is a space-separated list of values, then *var* will be an array, each element of which is one of the values, otherwise *var* will be set to *value*. To get a variable's value, use either

```
$var or ${var}
```

for scalar (non-array) variables, or

```
$var[index] or ${var}[index]
```

for arrays. To prevent such a sequence being interpreted as a variable (for now anyway), you can prefix it with a backslash (“\”) or enclose it in single quotes. Various other things can modify the value of a variable; you are referred to the `cs` man page for details. Table 3.2 lists variables that are special to the shell; their functions are described in the man page. Other

Variable	Meaning
<code>argv</code>	arguments to the shell
<code>cwd</code>	current working directory
<code>home</code>	home directory
<code>path</code>	executable search path
<code>prompt</code>	string to print as prompt
<code>shell</code>	full pathname of shell
<code>status</code>	return value of last command

Table 3.2: Special C Shell Variables

variables are also special, see the man page.

Certain variables are called *environment variables* and, unlike normal shell variables, they are propagated into any shells that are started by the shell in which they are set. That is, they will have the same value in any subshells of the shell in which they are set. Environment variables are set using the command `setenv` rather than `set`, but are otherwise used the same as normal shell variables (more or less, see “Variable Modifiers”

in the `cs` man page). You can list the current environment using the `printenv` command. Environment variables by convention have all uppercase names and are usually set in the `.login` file since their values are useful in all aspects of the login session.

A previous paragraph glossed something which deserves more attention, namely the role of quote marks in the C shell's interpretation of commands. Basically that role is as follows: anything enclosed in single quotes ("'") is left untouched by the shell, anything in double quotes ("") has variables substituted but is treated as a single word (ie. spaces aren't separators), and anything in backquotes ("`) is interpreted as a command which is run and its output substituted in place of the backquoted string. Table 3.3 summarizes the role of quotes in the C shell. Handling vari-

Quote	Type	Meaning
'	single	no substitution
"	double	variable substitution
`	back	command substitution

Table 3.3: C Shell Quote Marks

ous combinations of the quote marks effectively is one of the black arts of UNIX.

Finally, you can refer to files using patterns (called *wildcards*) rather than typing their entire name (or to refer to multiple files at once). Table 3.4 summarizes the various special characters. These special meanings

Metacharacter	Meaning
*	any 0 or more characters
?	any 1 character
[<i>abc</i>]	any of <i>a</i> , <i>b</i> , or <i>c</i>
[<i>a-z</i>]	any character in range <i>a</i> to <i>z</i>
~	your home directory
~ <i>user</i>	<i>user's</i> home directory

Table 3.4: C Shell Filename Wildcards

can be prevented ("escaped") either by preceding them with a backslash or enclosing them in single or double quotes. It is important to note that all these substitutions happen *before* a program sees its arguments. That

is, the shell expands your typed command line and passes the result to the program as its arguments. While this provides a degree of uniformity among applications, it can also be the source of some tricky bugs. It also means that if you want to use a shell meta-character as part the arguments to a program, you must escape them from shell.

3.1.4 Job Control

One of the main reasons for preferring the C shell over the Bourne shell is that it provides *job control*. This lets you run commands “in the background” while you are busy doing something else, and lets you stop and resume jobs.

To run a command in the background, end the command line with an ampersand (“&”). The shell will print a job identifier in square brackets followed by the process id of the job. It will then print your prompt and accept the next command immediately rather than waiting for the command to finish. When the background job terminates, you will be notified the next time the shell prints your prompt.⁵

To move a background job into the foreground (there can only be one foreground job), use the command

```
% fg job
```

where *job* is the number printed by the shell when the command was started. Don’t confuse the shell’s prompt with the percent sign required for job references. You can also use patterns to refer to jobs; the command

```
% fg emacs
```

will move the latest job whose command starts with `emacs` into the foreground. You can use the command

```
% jobs
```

to list your current jobs. To suspend a command while it is in the foreground, type `<Control-z>`.⁶ You can move a suspended job into the background with

⁵But see the description of the C shell special variable `notify`.

⁶This can be changed to another key if you like; see the man page for `stty`.

```
% bg job
```

where *job* defaults to the last job suspended. You can resume a suspended job in the foreground by simply typing

```
% %job
```

as a shorthand for the `fg` command. Again, you can use patterns rather than job numbers if you like. Note that the interrupt key (usually `<Control-c>`) that we described earlier works by killing the current foreground job.

Further information about job and process control can be found in the man pages for `kill`, `nice`, and especially `ps` which lets you inspect the state of processes in the system.

3.1.5 C Shell Programming

As we mentioned, the C shell is really a complete programming language and many problems can be solved more easily using shell scripts than using a C program. There is far too much to explain here, but we will mention a few of the features that get the most use.

All UNIX programs have associated with them three input/output streams called the *standard input*, the *standard output*, and the *standard error*. Figure 3.1 illustrates how these streams appear. Many UNIX programs are

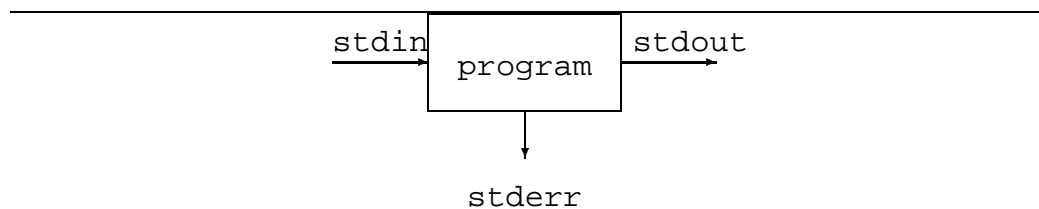


Figure 3.1: Standard I/O Streams

designed to be *filters*. That is, they generally read from files given on the command line or from their standard input if no filenames are given, write to their standard output, and send diagnostics to their standard error. The C shell provides mechanisms for you redirect these streams, which by default are connected to your terminal. This allows you to put output from commands into files that you can use later, and have commands take input

Syntax	Meaning
<i>cmd</i> > <i>file</i>	send stdout to <i>file</i>
<i>cmd</i> >> <i>file</i>	append stdout to <i>file</i>
<i>cmd</i> < <i>file</i>	take stdin from <i>file</i>
<i>cmd</i> << <i>token</i>	take stdin from script until <i>token</i>

Table 3.5: C Shell I/O Redirection

from files. Table 3.5 summarizes the redirection operators. Note that redirecting stdout by default destroys the previous contents of *file* if it existed. You can alter this behaviour with the shell variable `noclobber`.

Finally, a set of programs which follow the filter model can be connected together into a *pipeline* using the syntax

```
% cmd1 | cmd2 | ... | cmdn
```

The stdin of *cmd1* is left connected to the terminal, its stdout is connected to the stdin of *cmd2*, and so on up to *cmdn* whose stdout is left connected to the terminal.⁷

3.2 Other Shells

3.2.1 The Bourne Shell

As we mentioned, the Bourne shell `sh` was the original standard shell. It has different control structures for shell programming (in fact, nicer ones), but I/O redirection and variable substitution are substantially the same. The Bourne shell uses a file called `.profile` rather than `.cshrc`, uses the `export` command rather than `setenv`, and lacks job control or history mechanisms (see the man page concerning the C shell history mechanism). The main reason for using it is in writing scripts, for which you are referred to its man page and also the book by Kernighan and Pike.

3.2.2 CMU C Shell

This is a version of the C shell that was hacked at CMU to provide an editing facility for the command line to minimize the amount of retyping

⁷See also the `tee` program in connection with pipes.

needed. It uses certain variables to customize its behaviour which can be set to mimic the standard editors (see later sections regarding editors). Like usual, you are referred to the man page for details.

Chapter 4

The UNIX Filesystem

Well, we've now discussed quite a bit about various aspects of UNIX without describing how you can create or modify files. In fact, we haven't discussed files at all. There is a bit of a chicken-and-egg problem here, but we will first describe how the filesystem works and you can use existing files and directories to experiment with these commands. The next section will describe how you can view and edit files yourself.

The filesystem is one of the key aspects of UNIX, and one with which you will interact constantly. The filesystem is the way that permanent disk storage is organized. The two main concepts from a user's point of view are *files* and *directories*. Files are what you would expect, namely a collection of bits on disk that were typed in by a user or created by a program. UNIX does not make any special assumptions about its files in terms of formatting or contents. Directories are collections of files and can contain other directories, resulting in a tree-structured filesystem organization.

There is a special directory at the "top" called the *root* directory which is referred to by the pathname `"/`.¹ All other files and directories are referred to by their position relative to the root. This is done by separating the directory names with slashes `"/` and making the last *component* of the name the name of the file. Figure 4.1 is a schematic diagram of a typical UNIX filesystem. Valid pathnames in this filesystem include: `/bin/login`, the program which checks passwords and logs you onto the system, `/usr/staff/bin/enscript`, which formats documents for

¹Note that directory trees have their root at the top and branch downward, as opposed to "real" trees.

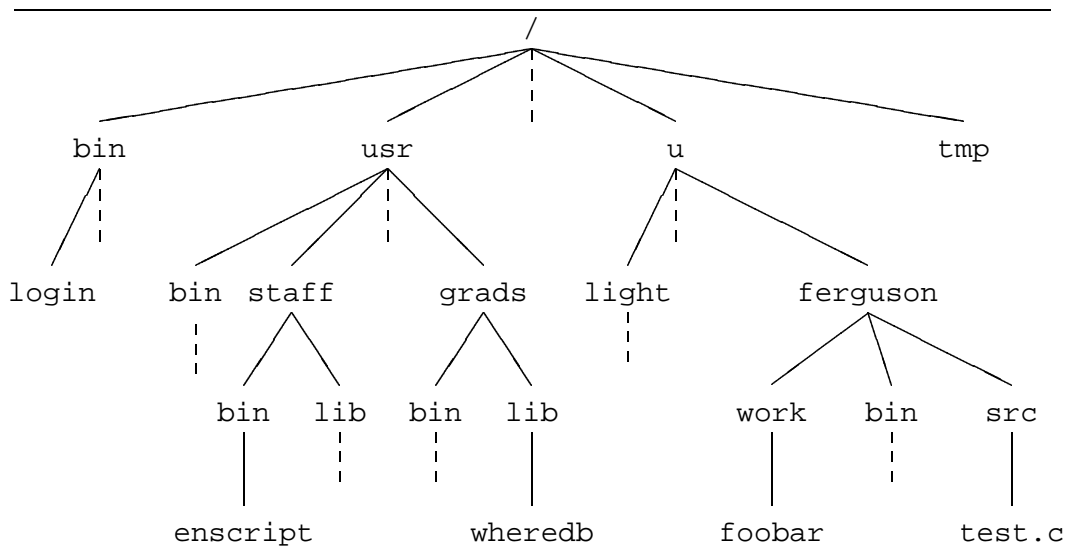


Figure 4.1: The UNIX filesystem

printing, `/tmp`, a directory where you can create temporary files, `/u/ferguson`, the user `ferguson`'s home directory, and `/u/ferguson/src/test.c`, presumably a file with some C code. You should be sure you understand how each of these files is named before going on.

There are several things to point out. First, to avoid always having to type the complete (or *fully-qualified*) pathname of a file, the system maintains a notion of your *current working directory*. If a pathname does not start with a slash, then it is a *relative* pathname and the search for matching components begins at your current working directory. When you log in, your current working directory is your *home* directory and you can change directories with the `cd` command described below. Recall that in the C shell, you can refer to your home directory with the file metacharacter tilde ("`~`").

All directories include the entries `."` and `.."` (pronounced "dot" and "dotdot", funnily enough). Dot refers to the current directory; that is, it is its own entry. Changing directory to dot doesn't do anything since you end up where you started. Dotdot refers to the directory's parent directory in the filesystem hierarchy.² Thus changing directory to dotdot

²Except for the root directory, where dot and dotdot both mean the root directory

while in `/usr/staff/bin` is the same as changing to `/usr/staff`, in the previous example. These can be used as often as needed, and dot and dotdot are always relative to wherever the search of the pathname is at that time. Thus,

```
/usr/staff/../../grads/bin/../../lib/./wheredb
```

is a convoluted way of referring to `/usr/grads/lib/wheredb`.

Finally, as mentioned above, UNIX does not enforce any relationship between a file and its name. Some programs interpret filenames in special ways (such as the C compiler assuming that files ending in `".c"` are C code), but this is up to the particular program and would be documented in its man page.

4.1 Filesystem Commands

The most used filesystem command is `ls` which lists the contents of a directory. It has a staggering array of options to display different types and arrangements of information. Table 4.1 summarizes some of the most useful ones. The `-a` option is necessary since by default `ls` does not display

Flag	Meaning
<code>-a</code>	list all files including dot files
<code>-l</code>	list in long format (more information)
<code>-R</code>	recursively list all sub-directories
<code>-F</code>	flag filenames to indicate type of file
<code>-d</code>	list information about the directory itself, not its files

Table 4.1: Useful `ls` flags

files whose names start with a dot (`"."`). These include dot and dotdot, and files which are typically not accessed very often or which are for internal use by applications. These often start with a dot to avoid cluttering the default output of `ls`.

Commands to manipulate files include `mv`, which renames or transfers a file or directory and `cp` which copies files or directories. If `bar` is a file (not a directory) or does not exist, then the command

itself.

```
% mv foo bar
```

will rename file `foo` to `bar`. If `bar` was a directory, then the command would rename `foo` to `bar/foo`, that is it would move it into the directory. You can `mv` either files or directories. The command `cp` is similar, but it leaves the original file behind rather than removing it. To copy an entire directory, you need to give the `-r` (recursive) option to `cp`.

The command `rm` removes (*i.e.*, deletes) files. Since you can use wildcards to specify the filenames for any of these commands (recall from Section 3.1.3 that they are expanded by the shell), something like

```
% rm -rf ~/*
```

is probably the most destructive thing you can do to yourself in UNIX.³ New users may want to use the `-i` option (“interactive”) to `rm` which prompts for confirmation before deleting each file. You can make `-i` the default by using a shell alias if you are using `csh` (see Section 3.1):

```
% alias rm "rm -i"
```

This can go in your `.cshrc` file so it always defined for you.

A final file manipulation command is `ln` (“link”) which creates a file which refers to another file, sort of. That is, the command

```
% ln foo bar
```

makes a file named `bar` which is linked to the original file `foo` (which must exist). All references to `bar` are “really” references to `foo`. In fact, they are the same file.⁴ If you delete `foo`, you will still be able to access the file using the name `bar`. These links (called *hard* links) have certain limitations; more typically useful are so-called *symbolic* links. To make a symbolic link, you give the `-s` option to `ln`, as in

```
% ln -s gnat fly
```

As with a hard link, references to `fly` are really references to `gnat`, but in this case the linkage is not so tight. In particular, only the original file (`gnat`) “really” exists, and deleting it will cause references to `fly` to fail with a confusing “no such file or directory” error. Table 4.2 summarizes all these commands.

³Do you see why? Don’t experiment to find out!

⁴For the technically-minded, the presence of links makes the filesystem hierarchy a directed acyclic graph, rather than a tree.

Command	Function
<code>ls</code>	list contents of current directory
<code>ls dir</code>	list contents of <i>dir</i>
<code>mv file1 file2</code>	rename <i>file1</i> to <i>file2</i>
<code>mv file dir</code>	move <i>file</i> to <i>dir</i>
<code>mv dir1/file1 dir2/file2</code>	move and rename <i>file1</i>
<code>cp file1 file2</code>	make copy of <i>file1</i> called <i>file2</i>
<code>cp file dir</code>	make copy of <i>file</i> in <i>dir</i>
<code>rm file</code>	remove (delete) <i>file</i>
<code>ln file1 file2</code>	make hard link to <i>file1</i> called <i>file2</i>
<code>ln -s file1 file2</code>	make symlink to <i>file1</i> called <i>file2</i>

Table 4.2: Filesystem Commands

4.2 Directory Commands

We mentioned the concept of working directory above; the command `pwd` prints your working directory.⁵ The `cd` command changes to another directory, `mkdir` creates a new directory and `rmdir` removes one, but only if it is empty, otherwise it prints an error message. Note that `mv` can be used to rename or move a directory. Table 4.3 summarizes these commands. The `ln` command with `-s` option also works for directories. The

Command	Function
<code>pwd</code>	print name of current directory
<code>cd dir</code>	make <i>dir</i> the current directory
<code>mkdir dir</code>	create <i>dir</i>
<code>rmdir dir</code>	remove <i>dir</i> if it is empty

Table 4.3: Directory manipulation commands

shell stores the name of the current working directory in the shell variable `cwd`. The C-Shell also allows you to maintain a stack of directories you are interested in and switch between them easily. See the `cs` man page description of the `pushd` and `popd` builtin commands.

⁵There is a C shell builtin command called `dirs` that is faster and usually better than `pwd`.

4.3 File Permissions

All files in UNIX have associated with them a set of *permissions* which control which users can perform what type of operation on the file. The most commonly used permission bits are those controlling *read*, *write*, and *execute* permission, and there are three of each of these. The first set of permissions controls what you, the owner of the file can do to it. The second set controls what users in your *group* can do, and the third applies to all other users. The `ls` command with option `-l` prints these bits as “`rw-rw-rw-`”. The meaning of these bits is different for files and directories, as Table 4.4 indicates.

File type	r	w	x
File	read,copy	write,replace	execute
Directory	access	create/delete	search

Table 4.4: Meanings of permission bits

For example, suppose if we have the following situation:

```
% ls -l
-rw-r--r-- 1 ferguson 2751 Jun 23 11:08 README
drwxr-xr-x 2 ferguson 1024 Sep 11 16:03 doc
drwxrwxrwx 2 ferguson 1024 Sep 10 13:52 pub
-rwxr-xr-x 1 ferguson 137 Jun 23 11:08 sortit
```

What this means is that `doc` and `pub` are directories, `README` and `sortit` are plain files, and `sortit` is executable (by everyone). The owner (`ferguson`) has both read and write permission on both plain files. Since directory `pub` has all permissions on, everybody else can also read (*i.e.*, access files), write (*i.e.*, create files), or search (*i.e.*, do `ls`) in it. However, while others can read and search in directory `doc`, only the owner can create and remove files there.

The permission bits (and there are a few others beyond the basic three groups of three) can be manipulated using the `chmod` command. It accepts either an octal number describing the desired bit value or a more mnemonic form such as

```
% chmod a+x file
```

which would add the execute permission to all three sets of bits for file *file*. See the man page for more details. It is crucial that you ensure that your files have the appropriate permissions. If you don't want others looking in a directory or at a file, use

```
% chmod go-rwx file
```

or (almost) equivalently

```
% chmod 600 file
```

(They would be different if bits beyond the basic nine were set.) It is a UNIX tradition that anything which is not protected against reading is available for others to read. One of the best ways to learn about UNIX is to exploit this and go prowling around the filesystem to see how others have done things and what programs are available. Of course tampering with other user's files should they have left them with the wrong permissions is not acceptable.

Finally, here are a few other filesystem commands for you to look up: `df` which prints the amount of free disk space; `spi` which looks for an executable (program), and `whereis` and `find`, which look for files more generally. In fact, `find` has a whole Boolean language of command-line options that allow you to search for files using many different attributes.

4.4 File Utilities

As we said above, almost everything you do in UNIX is associated with files, so its not surprising that there are lots of programs for searching and manipulating files. We will only mention some of them, and not in very much detail.

To compare files, the simplest method is `cmp`, which simply reports the first byte at which the files differ. Much more sophisticated is `diff` and its relative `diff3`, which indicate all differences between files. They can produce the output in a variety of forms depending on the intended use of the information. For example the `patch` program for updating software distributions uses "`diff -c`", a so-called *context diff*.

Various filters can also be applied to files (using the filter model described previously with `stdin` and `stdout`). These include `sort`, which can

sort in a variety of ways, `uniq`, which removes repeated lines of a sorted file, `tr`, which translates characters, `sed`, the stream editor, which applies editing commands to a file without you having to do it visually and by hand, and `awk` and `perl`, general purpose pattern-scanning and file processing programming languages. These last often allow you to write better text manipulation programs than the equivalent C program, and are easier to use than C.

Chapter 5

Pagers and Editors

Now that you're familiar with the filesystem and how filenames are specified, let's get on with the details of who you can view the contents of files and create new files or edit existing ones.

Like everything else in UNIX, you have several choices. We will first describe *pag*ers, programs for viewing files, and then describe the two most commonly used *edit*ors, programs for creating and changing files. You should realize that, unlike many microcomputer or specialty systems, most work on UNIX consists of creating or editing a file and then running your application using that file (which may mean compiling that source code or feeding that data to a program, for example) and repeating this until you get what you want. Very few applications are self-contained, and those that are are often scorned since they don't allow the user to choose their editor! Like much of UNIX, this is a reflection of UNIX's history as a software development environment.

5.1 Pagers

The simplest way to view a file is with the command

```
% cat file1 file2 ...
```

where `cat` comes from "concatenate", which admittedly looks funny if there is only one file. This will be fairly unpleasant though for a large file since `cat` simply reads the files and sends them to its `stdout` as fast as the

terminal can go.¹ Another use for `cat` is at the start of a shell pipeline.

A more useful way to view files is with a pager, of which the two standard ones are `more`, whose name comes from the prompt it prints after each page, and `less` whose man page synopsis reads “opposite of `more`”. In fact though, `less` has significantly more features than `more`. You can scroll forward one line by hitting `<Return>` or one page by hitting `<Space>`, scroll backward one line with `<k>` or one page with ``, go to the start of the file with `<g>` or the end with `<G>`. Hitting `<q>` will exit the program. There are lots of other operations (hit `<h>` for a list) and you can use the environment variable `LESS` to set defaults. Many programs check the environment variable `PAGER` to determine which pager to use when one is called for, so set it appropriately in your `.login` file.

5.2 Vi

To actually edit a file, you need to use an editor, of which there are two in general use. The traditional editor is `vi`, whose name comes from the fact that it is, in fact, the visual mode of the editor `ex`. That is, it allows you to move the cursor around on the screen and look at your file while making changes. In the old days of `ex` you gave commands to make changes, then other commands to see if the changes were correct.

The basic principle of `vi` is that it has two *modes*: command mode and insert mode. You start by invoking `vi` with the name of the file you want to edit (it will be created if it doesn’t exist already):

```
% vi filename
```

You will be in command mode with the first page of the current contents of the file (if any) displayed. You can move around the file using the commands summarized in Table 5.1.

To add or change text, you use the commands summarized in Table 5.2. These commands put you in insert mode: the characters you type are inserted into the file rather than being interpreted as commands. When you’ve entered all the text you want to, type `<Escape>` to return to command mode.

¹You can still use `<Control-S>` and `<Control-Q>` to pause and resume terminal I/O just like the good old days if you want to (and are fast enough).

Command	Description
<h>, <Backspace>	Move back one character
<l>, <Space>	Move forward one character
<w>	Move forward one word
	Move back one word
<e>	Move to end of word
<^>	Move to start of line
<\$>	Move to end of line
<j>, <+>	Move down one line
<k>, <->	Move up one line
<Control-u>	Scroll screen up
<Control-d>	Scroll screen down
<Control-f>	Move down one page
<Control-b>	Move up one page

Table 5.1: Vi Motion Commands

Command	Description
<i>	Insert text before cursor
<o>	Open new line below cursor
<O>	Open new line above cursor
<cw>	Change word after cursor
<cb>	Change word before cursor
<cc>	Change entire line

Table 5.2: Vi Insert Mode Commands

What about if you make a mistake, say inserting something that you didn't want? Well, if you haven't left insert mode yet (by hitting `<Escape>`), you can just use `<Backspace>` to delete it. If you've already left insert mode, you can use the undo command by typing `<u>`. You can undo just about any command, but you can only the very last thing you did. If you hit `<u>` again, it will undo the undo, leaving your text as it was. This limited undo ability is one of the key reasons to use a different editor, such as `emacs`, described in the next section.

To delete text, use the commands summarized in Table 5.3. Finally, to

Command	Description
<code><x></code>	Delete character under cursor
<code><dw></code>	Delete word forward
<code><db></code>	Delete word backwards
<code><dd></code>	Delete current line

Table 5.3: Vi Deletion Commands

save your file and exit the editor (or quit without saving, or edit another file, *etc.*, use the commands summarized in Table 5.4. You prefix each command with a colon ("`:`"), which is `vi`'s way of saying "I'd like to execute an `ex` command," which is what these are. You need to type `<Return>` at the end to actually execute the command.

Command	Description
<code><:w></code>	Write file
<code><:w> name</code>	Write file to <i>name</i>
<code><:q></code>	Quit if no changes
<code><:q!></code>	Quit without saving changes
<code><:wq></code>	Write changes then quit
<code><:r> name</code>	Read file <i>name</i> into buffer
<code><:r!> cmd</code>	Execute <i>cmd</i> and insert output

Table 5.4: Vi File Commands

For more details, you are referred to the document "Introduction to Display Editing with Vi" by Bill Joy. As for pagers, many applications

check an environment variable when an editor is needed; you should set `EDITOR` to `/usr/ucb/vi` to use `vi` by default. Many programs also use the environment variable `VISUAL` so you may want to set it also.

5.3 Emacs

A more powerful editor with many features for programmers is `emacs`. It is written in a LISP dialect and allows customization through the creation of new LISP functions. The major differences between `vi` and `emacs`, as I see them, are:

- Unlimited undo – `vi` will only allow you to undo the last command; `emacs` lets you undo right back to the last time the file was written, and even beyond. This is reason enough to use it!
- Multiple buffers and windows – `vi` let's you switch between files, but the facility is quite limited. On the other hand, `emacs` is designed to allow you to switch back and forth between several files, transferring text back and forth. It also allows you to split your screen into several parts, each of which can be looking at different files.
- Interaction modes – `emacs` is designed to be useful as a general programming tool, and so provides ways for you to interact with other programs, such as a Common Lisp system. You can use the same commands you use for editing text (and others) in these modes, providing a very useful degree of uniformity.
- Full programmability – `vi` allows you to define macros based on combinations of keystrokes; `emacs` provides a complete version of the Lisp programming language, specially tailored to editing text.

With all this and more, `emacs` can be difficult to grasp at first. As one pundit on the Net put it: "I find the idea of any feature ever being *removed* from GNU `emacs` so frightening that I don't want even the suggestion that this has occurred to go unchallenged." The following sections provide a very cursory survey of some of the main features, starting with the extensive help facility (another advantage over `vi`). I suggest that you become familiar with that first, and use it while you read the other sections.

You should set `EDITOR` to `/usr/staff/bin/emacs` to use it by default with programs that support such usage. An additional pointer is to use the program `emacsclient` to allow editing tasks from multiple jobs to use the same `emacs` process. See its manpage for details.

5.3.1 Notation

Emacs makes heavy use of the *modifier* keys Shift, Control, and Meta. Since any discussion of Emacs commands will refer to these often, we use the following shorthands: “C-x” means “Control-x” (*i.e.*, hold down Control and press `<x>`), “M-x” means “Meta-x” and, less commonly, “S-x” means “Shift-x”, but might be simply written “x”. Combinations of modifiers are possible, as in “C-M-x”, which means “hold down Control and Meta and press `<x>`.” Some other keys are often commonly used: `<Return>` (RET), `<Tab>` (TAB), `<Space>` (SPC), `<Escape>` (ESC), and `<Delete>` (DEL) are the main ones.²

Commands are Emacs-Lisp functions, and are bound to keystrokes using facilities to be described later. In what follows, we will be describing the default keybindings.

Command key sequences themselves have a conventional syntax. Any command can be executed by typing M-x and then entering the command in the *minibuffer* – the area at the bottom of the Emacs display. Most very common commands are bound to single keys, but since there aren’t enough of them, Emacs uses the concept of *prefix* keys to expand them. By default, C-x is the main prefix character. That is, typing it does not itself do anything, but opens up a new range of commands based on the next key typed. Again by default, C-c is the “mode-specific” prefix character. That is, it again introduces a new set of commands, but those commands depend on what *mode* Emacs is in (see below).

You can cancel any command or prefix character by typing C-g (which invokes the function `keyboard-quit`).

²If your keyboard doesn’t have a Meta key, you should type `<Escape>` followed by the key you want to “meta-ize”. Don’t hold down Escape the way you would hold down a modifier key like Meta.

5.3.2 The Emacs help facility

Let's start with how you go about getting help in Emacs. To get help, type C-h. (Note that C-h is also typically Backspace, which can be confusing. If you find yourself hitting Backspace and entering Help by mistake, see below regarding redefining keybindings). Emacs waits for you to enter another character indicating what type of help you want, or hit C-h again for a one-line list of choices, or hit C-g to abort. The most useful types of help are probably: b, which describes current keybindings; m, which describes the current major mode (see below); and i, which puts you in "Info" mode. You might also like to try the Emacs tutorial, available by asking for help topic t. Hitting C-h once more will give you a full-screen menu of help topics and descriptions of each.

In Info mode, you are in a hypertext-like browser, with information available on a variety of topics, including Emacs itself in gory detail. You select a topic from the initial menu (items marked with an asterisk) by typing m and then entering the item name and pressing <Return>. For example, for Info about Info, do m Info RET. Each chunk of information is called a *node*, and nodes are connected with Next, Previous, and Up pointers. The commands n, p, and u allow you to go to other nodes. Many nodes (like the directory node) have menus for which you use the m command as above. You can scroll the text of the node up and down with SPC or b respectively. To return to the top node of the current topic, use t. To return to the directory, hit d. To exit Info, hit q.

5.3.3 Basic editing

When Emacs starts, you are in a buffer called **scratch**. Buffers and files will be described presently, but first let's cover the basics of moving and editing in Emacs. You can experiment with these commands in this buffer.

Unlike vi, Emacs is *modeless*. In vi, you are either in command mode or in insert mode, and you have to give a command to enter insert mode or press <Escape> to return to command mode. In Emacs on the other hand, you can always insert text by simply typing. The location of the cursor on your screen is called the *point*, and new text will be inserted before (*i.e.*, to the left of) the point. To move the point around, you use the motion commands summarized in Table 5.5. You can get further in-

Command	Function
C-f	forward-char
C-b	backward-char
M-f	forward-word
M-b	backward-word
C-a	beginning-of-line
C-e	end-of-line
C-n	next-line
C-p	previous-line
C-v	scroll-up
M-v	scroll-down

Table 5.5: Basic Emacs motion commands

formation about these functions (indeed, about any function) by typing “C-h f *function* RET”. Emacs provides other context-dependent motion commands, for example to move over paragraphs or expressions. See the Info documentation for details.

The other basic editing capability is the ability to cut and paste text. Table 5.6 summarizes the basic cut and paste commands. To understand how Emacs implements some of these functions, you need to understand what Emacs means by the *region*. In addition to the point (the left side of the cursor), Emacs provides something called the *mark*, which you can think of as a token or marker that you can drop at any point in your text. To set the mark at the current location of the point, type C-SPC. Emacs responds with the message “Mark set.” Then, when you move the point with the motion commands, the mark stays where you left it. The region is defined as the text between the point and the mark. Remember that the point is to the left of the cursor position, so if you want to include the final end-of-line in the region, you need to position point at the start of the next line.

With that understood, what Emacs means by “kill” is that the relevant text is deleted, but is stored in what is called the *kill ring*. Thus kill-region will delete the text between point and mark and save it. You can retrieve text from the kill ring with the “yank” commands: yank retrieves the last thing you killed; yank-pop cycles through the kill ring

Command	Function
Delete	backward-delete-char
M-Delete	backward-delete-word
C-k	kill-line
C-Space	set-mark-command
C-w	kill-region
M-w	copy-region-as-kill
C-y	yank
M-y	yank-pop
C-x x c	copy-to-register
C-x y c	insert-register

Table 5.6: Emacs cut and paste commands

each time you invoke it, so you can retrieve something even though you have done other kills since killing it. Finally, a “copy” is like a kill but the text is put in the kill ring without being deleted from its original location. In addition to the kill ring, Emacs provides a set of named *registers* that you can use to save copied or killed text. The name of the register is a single character.

The final basic function we will present is how to search for patterns and possibly replace them with other text. Table 5.7 summarizes the relevant commands. The basic searching functions are what Emacs calls *incre-*

Command	Function
C-s	isearch-forward
C-r	isearch-backward
C-M-s	isearch-forward-regexp
M-%	query-replace

Table 5.7: Emacs search and replace commands

mental searches, or “isearches.” That is, as you type the characters to search for, Emacs incrementally moves to the closest match. You can either type DEL to erase a character from the search string (and Emacs will back up to

the last matching piece of text), ESC to terminate the search leaving point at the location being displayed, or C-s again to search for the next location matching the string you've got so far. If you abort with C-g, point will be left where you started the search from. There are some other possibilities, see the Info documentation. The "regexp" form of searching is identical in terms of interaction, but the search string you input incrementally is interpreted as a *regular-expression* – a more elaborate pattern than a literal string. Details of regular expressions are also in the Info documentation.

To automatically replace occurrences of one string with another, use the `query-replace` function. Emacs will prompt you for the two strings, and then for each occurrence of the target, you can type SPC or y to replace it with the substitution, Delete or n to skip to the next match, ! to perform all remaining replacements without asking, or ESC or q to exit without performing any more replacements. Like usual, there are more possibilities, see the Info documentation. To perform replacements using a regular expression, use "M-x `query-replace-regexp`" (there is no keybinding for this by default).

Now that you have all sorts of ways of changing your text, we should mention how to undo changes, since that is one of Emacs' best features. To undo the last change, type "C-x u" or C-_ (that's "Control-underscore"). As long as you don't do anything but undo, you can keep going, undoing changes right back to the last time the buffer was saved and beyond. As soon as you do anything else however, such as moving, the next undo will start again, probably by undoing your previous undos! Never fear, you can eventually get back to them by continuing the undo sequence without doing anything else.

5.3.4 Files, Buffers, and Windows

This section describes the commands that actually create and save files, and the various ways Emacs lets you view them. Table 5.8 summarizes the commands that will be described.

To start editing a file (which Emacs calls *visiting* a file), you use the function `find-file`. Emacs will prompt you for the name of the file you want to edit or create. What this does is create a new editing context, which Emacs calls a *buffer*, whose name is the name of the file. If the file exists, it will be loaded into the buffer and displayed, otherwise

Command	Function
C-x C-f	find-file
C-x C-v	find-alternate-file
C-x C-s	save-buffer
C-x k	kill-buffer
C-x s	save-some-buffers
C-x C-b	list-buffers
C-x b	switch-to-buffer
C-x 2	split-window-vertically
C-x 5	split-window-horizontally
C-x 0	delete-window
C-x 1	delete-other-windows
C-x o	other-window

Table 5.8: Emacs file, window, and buffer commands

it will be created when you save the buffer. Should you type the wrong filename, you can use `find-alternate-file` to replace the buffer with a new one. After you're done editing, you can save the file by invoking `save-buffer`. When you're completely done with the buffer, use `kill-buffer` to get rid of it. Emacs prompts you with the name of the current buffer. Hit `RET` to kill it, or type another name, or hit `C-g` to abort and not kill anything. Emacs will ask you to confirm the killing of a buffer with unsaved changes.

You can edit another file at any time by invoking `find-file` again. The default directory will be the same as that of the current buffer, and a new buffer will be created to edit the new file. If the files have the same name, the new buffer's name will have "`<2>`" appended.³ To switch between buffers, use `switch-to-buffer`. Emacs prompts you with the name of the buffer you last switched from, which makes it easy to switch back and forth between two buffers. You can get a list of all the current buffers with `list-buffers`.

Switching between buffers is useful, especially since you can kill or copy some text in one buffer, switch buffers, and yank the text into the

³You can have more than one buffer visiting the same file (as opposed to two different files with the same name), but this would be strange.

other buffer. However, sometimes it's more convenient to be able to see two or more buffers at the same time. Emacs allows you to split the display into multiple *windows*, each of which views a buffer. You can split the screen horizontally or vertically (see Table 5.8) and subsequently adjust their sizes (see the Info documentation). When you split a window, both resulting windows are viewing the same buffer. However, they scroll independently and have separate notions of point and mark. You can switch to a different buffer in one window with `switch-to-buffer`, and switch windows with `other-window`.

Be sure you understand the difference between files, buffers, and windows. Files are things on disk. Buffers are Emacs editing contexts, which can be associated with a file on disk, but only in the sense that they are initially read from the disk and can be written back to disk. In particular, the file can be changed (by another process or by saving another buffer editing the same file) without the buffer noticing it. Emacs does provide an "interlock" mechanism for detecting such conflicts and warning you if you have an outdated copy of a buffer. Finally, windows are simply views into buffers – changes to a buffer in one window are immediately reflected in all other windows viewing that buffer. Of course, the change may not be visible if the other windows have scrolled to a different point in the buffer.

5.3.5 Completion

For many commands, such as `find-file` and `switch-to-buffer`, Emacs provides a *completion* mechanism that can reduce the number of keys you have to type to respond to a prompt. The way it works is that for commands such as these, Emacs has some idea of the possible responses to the prompt. For example, for `find-file` the possible responses include the files in the current directory, and for `switch-to-buffer` the only valid response is the name of a buffer. In the minibuffer, therefore, certain keys are bound to functions that allow Emacs to complete your response if possible, or to list possible choices. Table 5.9 summarizes the minibuffer completion bindings.

If you hit `TAB`, Emacs will attempt to complete your response as far as possible. Hitting `SPC` is similar, but will only complete the next word (*i.e.*, up to a hyphen or space). Hitting `RET` will attempt to complete as for

Command	Function
TAB	<code>minibuffer-complete</code>
SPC	<code>minibuffer-complete-word</code>
RET	<code>minibuffer-complete-and-exit</code>
?	<code>minibuffer-list-completions</code>

Table 5.9: Emacs minibuffer completion bindings

TAB and, if successful, will send your response to Emacs. Note that the degree to which RET does completion depends on what Emacs is willing to accept as a response. For example, for `find-file`, since any filename is valid (whether the file exists or not), Emacs will not complete on RET. On the other hand, when executing a command with `M-x`, since only a valid command name is an acceptable response, RET will attempt to complete your input. In practice, it never hurts to hit TAB before hitting RET if you want to be sure Emacs will do completion.

5.3.6 X Mouse

Emacs was originally developed for use on ASCII terminals, not modern bitmapped displays. For this reason the basic operations are defined in terms of keystrokes. On a workstation, however, there is this handy thing called a mouse that can be very useful for moving the location of point, switching between windows, and for cutting and pasting text. Unfortunately, the complete integration of Emacs with display systems like X Windows will not be complete until the release of Emacs version 19. Meanwhile, several good packages have been developed independently that provide mouse functionality for Emacs. This section will briefly describe one of them, `x-mouse`, that is included by default with Emacs. You might also be interested in an alternate package, `x-sb-mouse`, or in looking at the Epoch version of Emacs.

The `x-mouse` package extends the Emacs keybinding system to include mouse clicks, with or without modifier keys. Table 5.10 summarizes the bindings. What these mean is that clicking the right button (Button1) will set point to where the pointer was when you clicked, selecting that window if necessary. Clicking the right button (Button3) will copy the text

Command	Function
x-button-left	x-mouse-set-point
x-button-middle	x-mouse-paste-text
x-button-right	x-mouse-cut-text
x-button-s-middle	x-mouse-cut-text
x-button-s-right	x-mouse-paste-text
x-button-c-middle	x-mouse-cut-and-wipe-text
x-button-c-right	x-mouse-select-and-split

Table 5.10: Emacs x-mouse bindings

between point and the location of the pointer into the kill ring and into the X cut buffer. Clicking the middle button (Button2) will move point to the location of the pointer and insert the contents of the X cut buffer. Text in the X cut buffer can be pasted into other X applications (like xterm), and text can be placed in the X cut buffer by other applications and pasted into Emacs. The other bindings shown in Table 5.10 indicate the effect of the Shift and Control modifiers. By holding down Shift while clicking, you can reverse the function of the middle and right buttons. By holding down Control and clicking middle you can kill (rather than copy) the text between point and the pointer or by clicking right you can split the window that the pointer is in.

5.3.7 Modes

5.3.8 Process interaction

5.3.9 Customizing Emacs

- .emacs (sample below) , keybindings, load-path, automode-alist, autoload
- Automode a-list: Sets buffer mode automatically based on filename. This is an association list (ie. a list of conses (x . y)) where the car of each cons (x) is a regular expression and the cdr (y) is the name of a function that will set the mode of a buffer. Each time a file is visited in a buffer, the filename is matched against each regexp in turn. At the first match, the corresponding function is called to set the mode. If no reg-

exp matches, then the buffer is left in the mode specified by the variable `default-major-mode`, which defaults to "Fundamental" mode.

- Autoload modules: Emacs allows you to delay the definition of functions by specifying that a file should be "autoloaded" when that function is first called. If the function is never called, then you save time and memory. Many of the functions are also bound to keys, and will be autoloaded when the key sequence is first encountered.

5.3.10 Sample .emacs file

The following is a short example of an Emacs initialization file. You would place this in a file called `.emacs` in your home directory. The discussion assumes familiarity with Lisp, although not necessarily with Emacs-Lisp.

The first thing to do is to set up some variables to hold the pathnames of our home directory and personal emacs directory, then set the `load-path` variable so Emacs knows where to find library (Emacs-Lisp) files. The code below adds a personal library directory (`$HOME/lib/emacs`) to the `load-path`. If you find useful Emacs stuff, that's where to put it so Emacs can load or autoload it. The default value of `load-path` includes the system library, which is why we add our personal library directory to the list rather than replacing the default:

```
(defvar homedir (getenv "HOME") "${HOME}")
(defvar emacslib (concat homedir "/lib/emacs/")
  "Pathname of my private library")
(setq load-path (append (list emacslib urlibdir)
  load-path))
```

This section is an example of rebinding keys. Many people find it annoying that Emacs uses `C-h` for help, since `C-h` usually means Backspace. That means that every time you go to delete something, you get into help-mode instead. The following rebindings allow you to use `M-h` for help (rather than its default `mark-paragraph`). Note that we also have to set `M-hM-h` to run `help-for-help`, and we can't change that `C-h` is required to get the help screen (since it's hard-coded in the definition of the function):

```
(global-set-key "\M-h" help-map)
(define-key help-map "\M-h" 'help-for-help)
```

```
(define-key global-map "\C-h" 'delete-backward-char)
```

Next, we decide to use the improved X mouse package `x-sb-mouse`, so we load it (from the system directory) when Emacs is running under X. We also turn off the bell and redefine a few of the bindings:

```
(cond ((eq window-system 'x)
      (load "x-sb-mouse")
      (x-set-bell 'flash)
      (setq x-mouse-1-window-click 'x-mouse-set-point)
      (setq x-mouse-1-window-drag 'x-mouse-copy-text)
      (setq x-mouse-2-window-click 'x-mouse-paste-there)
      (setq x-mouse-2-window-drag 'x-mouse-cut-text)
      (setq x-mouse-3-window-click 'x--mouse-ignore)
      (setq x-mouse-3-window-drag 'x-mouse-append-drag)))
```

In fact, these are the default `x-sb-mouse` bindings with the exception of the Button2 click action which by default pastes at the location of the pointer (like `x-mouse`), rather than at the location of point as with the above (`x-mouse-paste-there`).

Next we set the `auto-mode-alist` described above. Since file types are typically based on the extension (suffix) of the filename, these patterns all match an explicit period ("`\\.`") and the end of the string ("`$`"). See the Info documentation for regular expressions by doing "`m emacs RET m regexps RET`". Also, note that we append the default list to our customizations so we get the normal mode selections as well (unless we override them):

```
(setq auto-mode-alist
      (append
        (list
          (cons "\\.l$" 'lisp-mode)
          (cons "\\.lsp$" 'common-lisp-mode)
          (cons "\\.lisp$" 'common-lisp-mode)
          (cons "\\.scm$" 'scheme-mode)
          (cons "\\.el$" 'emacs-lisp-mode)
          (cons "\\.bib$" 'bibtex-mode)
          (cons "\.[1-8]$" 'nroff-mode)
          (cons "\\.ms$" 'nroff-mode)
          (cons "\\.man$" 'nroff-mode))
```

```

      (cons "\\ .cshrc$"      'generic-code-mode)
      (cons "\\ .login$"    'generic-code-mode)
      (cons "\\ .awk$"      'generic-code-mode)
      (cons "\\ .gp$"       'gnu-plot-mode))
  auto-mode-alist))

```

As an example of an autoload module (described above), here we specify that the `ispell` package should be loaded whenever one of its interface functions is invoked:

```

(let ((file "/usr/grads/lib/ispell"))
  (autoload 'ispell-word file "Spellcheck word." t)
  (autoload 'ispell-region file "Spellcheck region." t)
  (autoload 'ispell-buffer file "Spellcheck buffer." t))

```

Finally, some miscellaneous customizations. You can get information on most of these variables by doing “M-h v” and then typing the variable’s name followed by return (use C-h if you didn’t change the help bindings as described above).

(1) Always append a newline in case I forget, because some programs get very upset when there isn’t one:

```
(setq require-final-newline t)
```

(2) Use smooth scrolling with C-n and C-p:

```
(setq scroll-step 1)
```

(3) Whenever we’re in text mode, automatically break lines at right margin:

```
(setq text-mode-hook 'turn-on-auto-fill)
```

Chapter 6

Electronic Mail

Now that you have some idea about the fundamentals of interacting with UNIX, it's about time we looked at some of the things that you can use UNIX for. This chapter describes the basics of network addressing, how to read and send electronic mail, and how to customize your mail environment.

6.1 Electronic addressing

To start, recall that your username identifies you to the system. Since electronic mail travels between machines, usernames are not enough to uniquely identify you – there might be someone with the same username on a different machine. Thus your electronic *address* is a combination of your username and the name of your machine. In fact though, the network is so large these days that it is even possible for machine names to be non-unique.¹

To alleviate these problems, addresses are structured hierarchically just like the filesystem. Each address consists of a number of components (or *domains*), each of which identifies you at some layer of the hierarchy. Let's go through an example to make this concrete. Suppose your username is `fred`, and you're at the University of Rochester, in the Department of Computer Science, on machine `hal`. Your address might be

`fred@hal.cs.rochester.edu`

¹Did you you ever stop to think how many sites probably have a machine named "sun1", for example?

The at-sign (“@”) separates your username from the rest of your address. The periods (“.”) separate the domains of your address, just like the slashes did for filenames. The outermost domain is last in the list, in this case `edu` identifying the educational subnetwork of the United States. If you were in a foreign country, your outermost domain would indicate the country, such as `ca` for Canada or `uk` for the United Kingdom, but since the U.S. was first and is so big, they have the subnets at the outer level. Then within the U.S. educational subnet, we are in the `rochester` subnet, which corresponds to all the facilities in all the various departments of the University. Within that, we are in the `cs` subdomain, which picks out the exact department, and then finally identifies `machine hal` as being on that subnet.

If that seems like a lot to remember, it is. But it does provide a fairly uniform way to figure out someone’s address if you know where they are, and vice-versa. Two things are important to note about electronic addresses. First, many sites are set up so that mail can be read from any machine. In this case, we would omit the machine name from the address, yielding something like `fred@cs.rochester.edu`. We leave it up to the mail software at the destination site to figure out how to get it to `fred` once we get it to them. Secondly, you only have to specify as much of the address as is needed for the system to know where you want to go. So to send mail to someone in the Computer Science department the University of Rochester, we can simply use their username, since the software will use our sitename if we don’t give it one. So if our address is as above, then mailing to `barney` is the same as mailing to `barney@cs.rochester.edu`. If we want to send mail to `wilma` in the Psychology department of the same university, then we could mail to `wilma@psych` and the software would know that we mean `wilma@psych.rochester.edu`.²

In all honesty we should point out that the addressing scheme described above is referred to as Internet addressing and is only appropriate for major, well-connected sites. You may see two other types of addresses, which we will describe briefly. The first is an address of the form

`alpha!beta!gamma!dumbo`

²As reasonable as it may seem, you usually cannot omit the `edu` part of the address when mailing to another address. Details about this process can be found in the man page for `named`, but be warned that this whole subject gets very technical very quickly.

This is called a UUCP address, where UUCP stands for “Unix-to-Unix copy,” which is one way for mail to be transferred between sites (see the man page for `uucp`). Such addresses are a throwback to the days where you had to specify the exact path your mail would travel to reach its destination. These days, *routers* do most of the work and we can use Internet addresses and let the software figure out how to get the mail there. Still, less well-connected sites will still use UUCP addresses. The interpretation of the above address is “First send this message to machine `alpha`, get them to send it to machine `beta`, then on to machine `gamma` where it should be delivered to user `dumbo`.” Some of the machine names might be in Internet style.

The other type of address you might see is

```
betty%obscure@wellknown.edu
```

This means that the message is for user `betty` at site `obscure`, but the local router doesn’t know how to reach that site. However, we know that site `wellknown.edu` does know how to reach it, so we are passing the message to them to send on to `obscure`. This type of address is typically seen where `obscure` is overseas or otherwise remote, and `wellknown.edu` is big, well-connected site. Again, modern routers almost always insulate the user from these issues.

6.2 Sending Mail

Okay, so now you know your address and the address of the person you want to send mail to. How do you actually get the job done? The simplest way to send mail is to give the command

```
% Mail recipient(s)
```

where *recipient(s)* is (are) where you want the message sent. Note that case is important: `mail` is a different program than `Mail` (or used to be, compare their man pages). You will be prompted to enter the subject of your message (a short descriptive phrase), and then will be allowed to compose the text of your message. There is no prompt for this; you just type until you’re done, hitting `<Return>` to separate lines. When you’re done and are ready to send the message, type `<Control-d>` (or two periods)

on a line by itself. You will be prompted for any people who should get “carbon-copies” of the message (ie. they also get the message, but they can tell that they aren’t the prime recipient). You should type their addresses or just hit `<Return>` to not have any carbon-copies. Your message is now on its way.

What? You made a mistake? The Mail program provides a set of *escapes*, commands that allow you to stop entering text and do something else, and then resume composing your message. However, once you’ve sent the message (i.e., hit `<Control-d>`), there’s no way to cancel or change it! Anyway, these commands are called *tilde escapes* since they are indicated by typing a tilde (“~”) and a key letter on a line by themselves, then hitting `<Return>`. The most common tilde escapes are summarized in Table 6.1. If you just want to change the Subject line, use `~s` and enter the

<code>~s text</code>	Change subject line
<code>~h</code>	Change To:, Subject:, Cc:, and Bcc: lines
<code>~p</code>	Page message using \$PAGER
<code>~e</code>	Edit message using \$EDITOR
<code>~v</code>	Edit message using \$VISUAL
<code>~?</code>	Summarize tilde escapes
<code>~~</code>	Insert tilde at start of line

Table 6.1: Common Mail tilde escapes

new text before hitting `<Return>`. To change the Subject or addresses, use `~h` (for the “header” of your message). To invoke the editor to change the text of your message use `~e` or `~v`. When you quit the editor you can continue and add more text or type `<Control-d>` to end the message. There are many other tilde escapes; they are described in the help message obtained from the `~?` escape and in the man page.

To abort your message without sending it, you can either interrupt the Mail program with `<Control-c>` or use the `~q` escape. You can, of course, also suspend the process by typing `<Control-z>` and do something else (assuming you’re using a shell with job control, see Section 3.1.4).

6.3 Reading Mail

Now, you've sent mail and the person has sent a message back to you. How do you read it and what can you do after that? To read your mail, give the `Mail` command with no arguments, *i.e.*,

```
% Mail
```

The program will display a summary of your messages (the contents of your mailbox). If there is no mail to read, it will print a message to that effect and exit. Otherwise, you can look at a message with the `print` command, where you can specify a message number as in

```
print 2
```

to print the second message or specify message range as in

```
print 1-3
```

to print the first three messages. If you don't give a message number, most commands use the *current message*, indicated by a ">" symbol in the list of headers. Messages can also be identified by patterns, such as "all messages from Fred"; see the man page. If a message is very short, it will simply be printed. If it is longer than a few characters, `Mail` will use a pager to let you view it one page at a time (see Section 5.1). You can give the `headers` command to view the list of messages again. It also takes an optional message range as argument. Note that if you more than about 36 messages, `Mail` will by default only list the first ones. You must give a message range to see the rest. Commands can in general be shortened to their first letter. Thus `print` can be `p` and `headers` can be `h`. You can get help by typing `help`.

6.4 Replying To Mail

To reply to a message that you have received, use the `reply` command or type `r`. If no message specifier is given, you will be replying to the current message. By default, the `reply` command only replies to the original sender of the message, even if it was addressed to several people. To send your reply to all the original recipients as well as the original sender, use

Reply with a capital “R”.³ You should also look at the description of the Mail variable `metoo` in this regard (see below).

To send a message to another user, you can use the `mail` command (or `m`). This is the same as invoking Mail from the command line; you should specify the recipient(s) as arguments. To forward a message to another person, use `reply` or `mail` to send them a message, and use the `~f` escape to include the forwarded message.

6.5 Saving messages and exiting

By default, when you exit Mail with the `quit` command, any messages that you read are transferred to your personal mailbox, a file called `mbox` in your home directory.⁴ Any messages that were marked for deletion with the `delete` command will be deleted. If you mark a message for deletion and then decide you’d rather not delete it, you can give the `undelete` command before quitting. Once you’ve quit the deleted messages are gone forever. You can also use the `save` command to save a message in a file. Messages saved like this, as well as messages in your personal mailbox, can be viewed using the `-f` option to Mail, ie.

```
% Mail -f filename
```

which indicates that *filename* should be used for this session rather your system mailbox. Subsequent `save`’s append to the mail file. Saved messages are also deleted when you `quit`.

Finally, if you get confused about the state of your mail, or if you do something and can’t figure out to undo it, you can use the `exit` command (or `x`) to leave Mail without deleting anything.

6.6 Customizing Mail

You can customize the behaviour of Mail by setting various things in your `.mailrc` file, which is read each time Mail starts. There are many variables that can be set; boolean variables are set with a command “`set var`”

³Note that this is a reversal of the standard Berkeley Mail behaviour.

⁴See the Mail variable `hold` if you would rather keep read messages in your system mailbox.

or “unset *var*”, other variables are set with “set *var=value*”. Some of the more useful variables are shown in Table 6.2. The man page has a

Variable	Type	Meaning
askcc	Boolean	Ask for CC: list
hold	Boolean	Keep undeleted messages in system mailbox
metoo	Boolean	Allow mail to yourself
prompt	String	Prompt used by Mail

Table 6.2: Selected Mail variables

complete list.

The other important way to customize your mail is through the use of *aliases*. These allow you to use a short, easy to remember name for someone and have Mail automatically expand it into the person’s address. Aliases go in your `.mailrc` file, using the syntax

```
alias name address1 address2,...
```

You can create a “group alias” by separating the addresses of the members of the group with spaces. Then every time you send mail to *name*, it will go all the members of the group. For example, you can use aliases if you don’t want to remember someone’s whole address, as in:

```
alias gf ferguson@cs.rochester.edu
```

An example of a group alias might be

```
alias bugs fred barney joe@widget.com
```

In this last, the first two members of “bugs” have local addresses, the third member doesn’t. Also, alias can refer to other aliases, *etc.* Using aliases properly is key to managing large quantities of mail.

6.7 Other Mail Tools

The following briefly describes some of the programs available that have to do with electronic mail. Generally they will just be mentioned and you can look them up in the manual. A good way to find out about these things is to do

```
% man -k mail
```

and look up some of the items that are listed.

There is an X Windows interface to Berkeley Mail called `xmail` (see also the section on X Windows).

To be informed when mail messages arrive, you can use `biff` (from “be informed”). A somewhat nicer program that does the same thing is `mailwatch`. There is also an X Windows version called `xbiff`.

There is a completely separate mail handling system called MH, developed at Rand Corporation. Rather than a single program that does it all, `mh` follows the UNIX tradition and provides a set of tools that allow you to send, read, and sort mail, among other things. You should note that if you use `mh` the preceding discussion of mailbox files does not apply. In particular, it will copy files from your system mailbox automatically, which might bother you if you also wanted to use Mail. There is an X Windows interface to this system called `xmh`.

Finally, you can process your mail from within `emacs`, if you use it as your editor. The Emacs mail system is called RMAIL, and full details are available from the `emacs` help facility. As with `mh`, the resulting mail files are not quite compatible with Mail. However, there are programs to allow you to convert back and forth between the “BABYL” format used by `emacs` and the standard Berkeley format.

Chapter 7

Network News

Electronic mail is one of the main uses of computer networks, the other is electronic news. Netnews began as an informal way to use electronic mail to connect people who wanted to discuss topics of common interest. It has since evolved into a huge distributed system of information exchange and retrieval called the Usenet. We will first describe the structure of Usenet, and then describe how you can use it.

The metaphor often used to describe netnews is that of a bulletin board. Users can *post* messages (or *articles*) on the board that others can then read. Since there are so many users with so many diverse interests, Usenet is broken down hierarchically (like the filesystem and network addresses) into topics and subtopics. A particular topic area is called a *newsgroup*. At the top level of the hierarchy are broad divisions; the standard toplevel classifications are summarized in Table 7.1. In addition to these standard

comp	Computers and computation
sci	Research and application of established sciences
misc	Things that don't fit anywhere else
soc	Social issues and socializing
talk	Debates without appreciable amounts of useful information
news	USENET and news software
rec	Hobbies and recreational activities
alt	Alternative anarchic stuff

Table 7.1: Toplevel USENET classifications

toplevel divisions, there are also divisions of local interest, such as `cs`, which contains groups related to the Computer Science department. Note that the `alt` hierarchy is not really a part of Usenet proper, and has different rules.

As with network addresses, newsgroups are named by separating the components of their name with a dot (“.”). Unlike addresses though, where the toplevel domain is on the right, newsgroup names are read from left to right. Thus `cs.general` contains articles of general interest to members of the Computer Science department, while `rec.sport.hockey` is about, from left to right, recreational activities, in particular organized sports, in particular hockey.

7.1 Reading News

To read news, you use the `rn` program.¹ This program uses a file called `.newsrc` to keep track of which newsgroups you are interested in, and which articles you’ve read in those groups. When you first start `rn`, it will offer you a long list of newsgroups from which you are supposed to select the ones you want. Another possibility is to use the `newsetup` command before starting `rn`. At any time in `rn`, you can type `h` to get help about your options at that point.

When you decide to follow the articles in a newsgroup, this is called *subscribing* to the group. You can either subscribe to a group by answering yes when `rn` first offers it to you, or by using the `g` command to “goto” the group, and then subscribing when prompted. You should probably subscribe to most of the local newsgroups since they contain potentially important news and announcements. Another important group to subscribe to, at least at first, is `news.announce.newusers`. You should read and understand the articles in this group before getting involved in Usenet. If a group becomes uninteresting, you can *unsubscribe* by giving the `u` command. You will be unsubscribed from whatever group is current at the time you give the command, so be careful. If you accidentally unsubscribe, use `g` to re-subscribe.

Now, assuming you’re subscribed to some newsgroups, `rn` begins by listing groups that have unread articles in them. This is called being in

¹There are several Emacs packages for reading news, such as GNUUS.

“newsgroup selection mode.” You can move through this list using the `n` and `p` (“next” and “previous”) commands, or the `g` command described previously. When you get to a group that you would like to read, answer `y` to do so. At this point `rn` will present the first unread article in the group if there are any. You are now in “pager mode”, and you can see more of the article by pressing `<Space>` or go back with ``. When you reach the end of an article (or if there were no unread articles in the newsgroup you selected), you are in “article selection mode”. At this point you can go to the next or previous unread article in the group by typing `n` or `p`, respectively. If you want to look at articles that you’ve already read, use `N` and `P` (*i.e.*, capital “N” and “P”), respectively. You can also go to the next or previous article on the same subject with `<Control-n>` and `<Control-p>`, respectively. As always in `rn`, help is available by pressing `h` at any point.

There are myriad other commands you can use to handle the ever-increasing flood of news. For example, you can use `s` to save a message to a file, or `|` (vertical bar) to pipe the contents of an article through a program. For example, to print an article from article selection mode, use “`| enscript`”, that is, “pipe into enscript”. Finally, if someone gets on your nerves, or if a particular subject gets boring, you can *kill* articles matching a pattern using the `k` or `<Control-k>` commands. See the man page for more details.

7.2 Posting News

Once you’ve read some netnews, in particular the introductory and “netiquette” articles in `news.announce.newusers`, you’re ready to jump into the fray with your own opinions. If you want to reply to an article, use the `f` (for “followup”) from `rn`. This will invoke the program `Pnews`, which will allow you to compose a message and then post it. You can specify which editor to use by setting the `EDITOR` environment variable. To include the text of the followed-up-to message, use `F` (capital “F”). The newsgroups and subject lines will be set appropriately. If your reply to an article is probably not of general interest, or if the original poster requested replies by email, you can use the `r` or `R` commands to reply directly to them. As before, the capitalized version of the command includes the original text for you to refer to.

If you want to post an article on a new subject, then you can use the

`Pnews` command from your shell. You can either pass the newsgroup(s) and article title on the command line, or wait and `Pnews` will prompt you for them. Multiple newsgroups should be separated by commas. Note that posting simultaneously to several newsgroups (called *cross-posting*) will increase the coverage of your article, but may upset people who feel that your message was inappropriately directed into “their” newsgroup. You’ll find out soon enough about such things. As with following-up, `Pnews` will invoke the editor to let you compose the posting, and then allow you to post or abort it.

If you post an article and subsequently find it to be in error, or if the question you asked has been answered, you might consider *cancelling* your article. To do this, select the article in `rn` and use the `C` command. You should understand a bit about how cancelling works in order to use it appropriately. First, there’s no way to cancel the original message, since the news software has already propagated it at least throughout your local site, and probably far beyond (depending on the newsgroup’s *distribution*). Instead, another message is sent by the news software to inform other sites that your article is to be deleted. Once they get the second message, your article will no longer be available to readers at that site. Due to the latency involved in this process, and the fact that even sending the cancel message uses network bandwidth, you probably only want to cancel local articles (article posted to local newsgroups) or articles in which a particularly bad error was caught relatively quickly. Articles expire by themselves within at most a few weeks anyway.

Finally, some newsgroups are *moderated*. This means that people are not allowed to post articles to the group directly, but rather must mail them to a moderator who will determine their suitability and post them if appropriate. Ideally, this decreases the amount of noise in the group, but in practice it is entirely dependent on the efforts of the moderator. Moderated groups are indicated as such by `rn`, and an attempt to post to such a group is transformed automatically into a mailing to the moderator. The details of moderation are in the netiquette articles in `news . announce . newusers`.

Chapter 8

Document Preparation

Note: Much of the material that is supposed to be in this section is covered in my Department Guide 12, “Document Formatting using \LaTeX .”

8.1 Printers

- lpr, lp
- lpq, lpstat
- lprm, cancel

8.2 Printing Programs and Files

- enscript
- indent, cb

8.3 Troff

- description and principles
- usage: roff
- preprocessors: eqn, tbl, pic

- bibliographies: bib, invert, lookbib
- previewers: xtroff

8.4 **T_EX and L^AT_EX**

- description and principles
- usage
- bibtex
- previewers: xdvi, xtex

8.5 **Figures and Graphs**

- xfig, fig2dev
- gnuplot, grap
- psfig for troff and TeX

8.6 **Other utilities**

- spell, ispell
- wwv
- Island Tools

Chapter 9

Network Communications

9.1 Your Machine

- who, w, whoami
- date, uptime, ps
- hostname, mach

9.2 The Local Network

- rwho, ruptime, ru, rh, rf
- rlogin, rsh

9.3 The Internet

- firewall issues
- finger
- telnet
- ftp
- www

Chapter 10

X Windows

Note: The following information, while still technically accurate, does not reflect the now-standard use of the Common Desktop Environment (CDE) on department workstations. Maybe some day...

In the old days of UNIX the only way to talk to the system was via an ASCII terminal connected to a serial port on the host you worked on. These days, and especially in our department, much of your work is on workstations which can support much more sophisticated forms of interaction.

The **X Window System**, a joint project of MIT, DEC, and others, provides a portable platform for development of graphically-oriented programs. In particular, it supports the notion of you having multiple *windows*, each devoted to different task or different views of the same task. The source code for X is freely redistributed (although it is copyrighted) and many of the programs and tools have been developed by individuals and contributed via the network.

The following sections provide a very brief overview of the user-level view of X. The intention, as always, is to provide enough information to get you started and to point you towards further sources of information. Details for programmers wishing to write X applications are available in the manual set.

10.1 Your X Environment

In order to start an X session on your workstation, several things must happen. These are typically taken care of by two shell scripts which we will refer to as `start-x` and `.xinitrc` (you might prefer other names).

The `start-x` program must:

1. set the `DISPLAY` environment variable;
2. start the X server;
3. clean up after the server exits.

The `DISPLAY` variable tells the server where the graphics for this session should be drawn. It is of the form `hostname:0.0`, where `hostname` can be `unix` to refer to the current host or the name of your workstation. A typical C shell command to set the variable would be

```
% setenv DISPLAY unix:0.0
```

You then want to invoke the program `xinit`, which will start the X server and run a script (called `.xinitrc` from your home directory by default, or you can give a filename on the command line). When the script terminates, `xinit` shuts down the server. To clean up after the server, you should run the command

```
% kbd_mode -a
```

which will ensure that the console keyboard is working normally.

The script that `xinit` runs determines what your X session looks like when it starts up. It should list any X applications which you want started automatically; see the list below for some ideas. You should ensure that each of these is run in the background (i.e. the command ends with an ampersand) so that `.xinitrc` does not block waiting for it to end. The last command in the script should start your window manager, typically `twm`. This is the program which controls opening, closing, positioning, and resizing windows, as well as a host of other features (see the next section). It should not be run in the background, since otherwise `.xinitrc` would finish and `xinit` would shut down the server and exit. Instead, `.xinitrc` will exit when `twm` exits, which is typically done from a root-menu entry. A typical `.xinitrc` therefore, might start an `xclock` and

several `xterm`'s (one of which would be a console window), and then invoke `twm`.

Further details on the startup behaviour of the X server are available in the man pages for `X` and for `xinit`. The former also describes something of the X approach and the way well-behaved applications accept a standard set of command-line arguments to customize behaviour.

10.2 Window Manager

As mentioned above, under X your session is managed by a window manager, typically `twm`. The window manager is just another user program though, which distinguishes X from other window environments where the window manager is really the whole graphics kernel (such as Suntools). In fact, you needn't have a window manager at all!¹

It would take far too long to describe all the possible ways of customizing `twm`. These customizations are performed through the `.twmrc` file which should be in your home directory. A sample file is provided showing one way of customizing the interaction. There is also the default configuration (provided if no `.twmrc` file is found) and of course, you can refer to the man page for the gory details.

10.3 Applications

The following is a list of just some of the many programs available under X.

`xterm` This is the terminal emulator, which provides you with a shell running in a window. You can have several of these running simultaneously, with input going to whichever the mouse pointer is in. The `-C` option makes the window a console window, which captures messages sent to `/dev/console`, and the `-e` command can be used to simply execute a command in a window and then exit.

`xclock` This is a simple analog clock, which you would typically place in the corner of the screen by a command in your `.xinitrc`.

¹Try running `xinit` without having a `.xinitrc` file in your home directory.

`xload` Monitors the system load average on a bar graph.

`xbiff` Puts up a mailbox flag and beeps when you receive new mail.
Click on it to lower the flag again.

`xmh` A tool for processing electronic mail, based on the Rand MH system
(see the section on Electronic Mail).

`xclipboard` A place for cutting and pasting selections.

`xpostit` Those little yellow sticky notes for X. They are reloaded each
time you start up X.

`xcalc` A calculator tool.

`xeyes` The eyes follow you everywhere...

`xwd` Dump an X window for printing.

`xgrabsc` A better program for grabbing parts of the screen.

The programs `xset` and `xmodmap` are used to tailor the X environment
once the server is running.

Chapter 11

Program Development

11.1 The C Programming Language

- cc, gcc (K&&R)
- CC, g++ (Soustroup)
- lint
- codecenter, objectcenter

11.2 Debugging

- dbx, xdbx, gdb, gdb+

11.3 Project Management

- make
- imake, xmkmf
- rcs, sccs, cvs

11.4 Other Languages

- Lisp: cl (ref: Steele)
- Prolog: prolog (Quintus) (ref: Clocksin&Mellish)
- Java: javac, java

Chapter 12

Archiving Files

12.1 Shell Archives

- shar, unshar, makekit

12.2 Tape archives

- tar, mt

12.3 Compression and Encoding

- compress, uncompress, zcat
- gzip, gunzip, gzcat
- crypt, decrypt
- uuencode/uudecode, btoa/atob/tarmail