Berkeley Unix Primer

C R

C S
Computer Science Facilities Group
C I

L S

RUTGERS
The State University of New Jersey
Center for Computers and Information Services
Laboratory for Computer Science Research

19 December 1989

This document describes Unix on the Pyramid and Sun computers, as used at Rutgers University.

Table of Contents

This is an introduction to Berkeley 4.2BSD and 4.3BSD, as used on the Pyramid and Sun computers at Rutgers University. Unix has very good online documentation. Thus this manual does not try to be a complete reference manual. Instead, the goal is to give you enough general information that you can use the other documentation intelligently. Thus it emphasizes system conventions and overall approaches.

Our primary audience is students attempting to develop programs. However even if you are not going to be writing a program, you will find most of this manual useful. If you are not interested in programming, you should skip section 8 (Compiling and Running Your Program). The most critical sections are 3 to 6. These describe the how to type commands to the system, the file system, and how to get help.

This document describes Unix for people who are using conventional terminals. Those of you using Sun workstations have additional graphical and window-oriented facilities available. These are described in separate documentation for the Sun. However you'll still spend most of your time using the commands described here. So we recommend that you start with this manual, get used to Unix in its simple form, and then start playing with windows and other fancy features.

The overall design of the document is

1	Setting up your terminal, logging in to the system. Even if the rest of this seems too elementary for you, look at sections 1.6 and 1.7.
2	An example session.
3	How to type commands to the system, including how to fix typo's, and how to stop programs.
4	The file system: How files are named, as well as certain commonly-used attributes of files.
5	Commands: The overall syntax of commands.
6	How to use the documentation facilities in Unix. Unix is moderately helpful, but only if you know how to take advantage of it. This chapter tries to help with that.
7	How to create and change files. If you are using certain canned programs, you may not need to do this, but almost all users will eventually have to use an editor.
8	This section describes how to compile and execute programs. Everyone who is going to use one of the conventional compiled languages (not LISP) should read this. People who are just using mail or canned packages do not need it.
9	The mail system.
10	The Unix news system, which is used for both network newsgroups and local purposes, such as communication between faculty and students in a course.

If you are going to do make serious use of Unix, you will probably also want to read documentation on the shell carefully. Since the shell is what reads and executes your commands, you will spend a lot of time talking to it. You will also want to go through the EMACS tutorial, by typing the command "teach-emacs".

Finally, you should glance over the Unix reference card. This document does not give you all of the commands. It is more of a conceptual introduction. The Unix reference card does attempt to list all of the commands that you are likely to use. If something looks useful, you are encouraged to use the "man" command to find out more about it. If it involves terminology or concepts that you haven't seen before, you may want to look in this document's index.

1. Logistics

1.1. Getting Connected

In order to use Unix, you will need to use a terminal. Most often, you will use a terminal located at Rutgers. These terminals are permanently connected to the computer, or to a terminal server that will let you get to any computer on campus. In that case, you can simply walk up to it and hit the carriage return key, and it will be ready for you to type commands.

However when you want to work at home, things are more complex. The rest of this section discusses what you have to do to connect to our systems from home. You will need to have your own terminal, and provisions to contact Rutgers via a telephone line. In this case, you will need the following pieces:

- a terminal or a microcomputer with terminal emulation software (often referred to as "communications software"). We will use the word "terminal" to include a micro running a terminal emulator. See section 1.4 for more information about using terminal emulators.
- a modem. This is a small box that converts signals between a form that the terminal understands and sounds that can be sent through a telephone. Normally it has two small rubber cups into which the telephone handset fits.
- a special wire running between the modem and the terminal

If your terminal and modem are always used with Unix, then things should be easy. However if you switch back and forth between several systems, or if someone else has been using it before you and you don't know what they were doing, you may have to set switches or other options. Since different brands of terminals and modems are different, it is hard to tell you exactly what to do. But here are some choice that you often have to make:

baud rate - Most of our modems will accept 300, 1200, or 2400 baud, although some older modems do not support 2400. Normally you will want to use the fastest speed that your modem will allow. Generally terminals will handle almost any baud rate, and the modem is the limiting factor in speed. You may have to set your modem to the speed that you want. If you are getting garbage characters (typically random lower case and punctuation), this setting may be wrong.

parity - (This applies to the terminal.) Typically this is a choice among odd, even, mark, space, or none. Unix does not seem to care much about this. We normally recommend "space" or "none" if it is available, or "even" if it is not. If you have a parity problem, typically you will see half of your characters missing or displaying some special symbol (like "PE" or a big blob).

data bits - (terminal) Normally this is a choice between 7 and 8. Typically this should be set to 7. However sometimes this will interact with the parity setting, and with parity none, you might have to set data bits to 8.

stop bits - (terminal) Traditionally this is set to 2 for speeds of 300 baud or slower, and 1 for higher speeds. With modern equipment, you may be able to use 1 at all speeds.

duplex - (both terminal and modem) Typically this is a choice between half-duplex and full-duplex. Sometimes you will also see the term "local copy". You want full-duplex, or local copy turned off. If you are getting two or more copies of each character you type, this setting is probably wrong.

other settings - Most modern terminals have lots of strange options. Generally you want to turn them off. Unix wants to control your terminal. So anything that causes the terminal to generate extra characters automatically, ring bells in odd situations, etc., should be disabled. The one exception to this is "auto newline" or "auto wrap". This is an option, available on some terminals, that cause it to go to a new line when an existing line is filled with data. You should turn this on when using Unix. You do not want anything whose description includes terms like "block mode".

Generally the same settings should work on Unix and on VAX/VMS timesharing systems. If you use the same terminal with an IBM system, you will almost certainly have to change duplex when going back and forth, and you may have to change parity.

Assuming that your terminal is properly set, here is what you have to do to connect it:

- Make sure that the terminal and modem are connected to each other, that they are both plugged in, and that they are both turned on.
- Dial the telephone number of one of our terminal servers. You can find these numbers by typing "man dialups" on any of our Unix systems. Be sure to use the correct group of lines. In general New Brunswick Computer Science faculty and students have separate numbers, and then there are general numbers for other users. While you can access any machine from any terminal server, response will be better and service more reliable if you use the one intended for you.
- When the terminal server answers, the phone will stop ringing, and you will instead hear a whine or tone of some sort. At this point you you connect the phone to your terminal. With modern modems, this may happen automatically, or you may have to press a button to put your terminal online. Some older modems have rubber cups for the telephone handset. In that case, you have to place the handset in the cups.
- Generally some sort of light will light on your modem and/or terminal to indicate that the terminal, your modem, and our modem, are all connected. This may take 15 seconds or so. Some newer modems are full of options, and the two modems have to negotiate sets of options that they can both handle.
- At this point, hit the carriage return key every second or so until you get a greeting message. (An example greeting message is shown in the next section.) The reason you have to hit the carriage return key several times is that the terminal server has to figure out what speed your modem is using. It may have to try several speeds before it is able to understand you.

1.2. Using the terminal servers

Most of our computers do not have terminals and modems of their own. They are connected to the Rutgers campus network. This gives them access to a set of "terminal servers". A terminal server is a special-purpose machine that allows you to choose what computer you want to talk to. By using terminal servers, many computers can share a common set of terminals and modems. This saves us money, and makes it more likely that you'll be able to find a free terminal or phone line.

To get the terminal server's attention, hit the carriage return key. When you do, you'll see greeting message that looks something like this:

```
      cisco ASM-64, Rutgers LCSR Computer Science Network, Node Hilltop.
    Please type name of machine you wish to connect to, followed by a <CR>.
hilltop>
```

Hilltop is the name of the particular terminal server. The one you are using may have a different name. The name doesn't matter. All of our terminal servers can reach all of our computers.

Once you are talking to the terminal server, simply type the name of the computer you want to log in to. Your faculty member or TA should have given you this name. If you try to log into the wrong computer, it probably won't recognize you. Here's what it looks like to log in. In this case we're trying to log in to a system called "topaz".

```
hilltop>topaz
Trying TOPAZ.RUTGERS.EDU (128.6.4.194)... Open
OSx 4.1 UNIX (topaz.rutgers.edu)
login:
```

All we typed was the name of the system we want to use, "topaz". If the first line doesn't say "Open", the computer you wanted can't be reached. Either you typed the wrong name, or something is wrong with the equipment somewhere.

Once you see the word "Open" you are now talking to the computer, in this case topaz. You'll get a greeting message from it, which may be longer than what is shown here. Finally you will see the "login:" prompt. At this point you have to identify yourself with a user name and password. However before going through this, we want to say a little bit about how to type things to Unix.

1.3. Keyboard

You will talk to Unix using your terminal's keyboard. Unix talks to you through your terminal's CRT screen. Notice that the keyboard looks much like a standard typewriter. In particular, many keys are labelled as they are on a typewriter. However there are some differences:

- The Rubout key. When you make a mistake typing a character, you can delete it with the rubout key. In most cases, the character will actually disappear from the screen when you delete it. Different terminals have different labels for this, including "rubout", "rub", "delete", and "del". On some terminals you have to hold down the shift key in order to produce this. The rubout key is often located at the right edge of the keyboard. If you are using a terminal emulator running on a microcomputer, its documentation should tell you how to produce a rubout. (In some cases it is much more convenient to use a key labelled "backspace". It is possible to tell Unix to accept the backspace key instead of the rubout key.)
- The Control key. This key is not meant to be typed by itself. When it is held down, it changes the meaning of any other key that is typed. For example, if you hold down the control key and then type "s", this sends a special character called a control S, normally denoted by "^s". This key is normally labelled "control", "ctrl", or "ctl". If you are using a terminal emulator running on a microcomputer, its documentation should tell you how to produce control characters. The control key may be labelled as something else completely (such as "option"). On an Apple computer, it may have a picture of an apple on it.
- The Return key. The symbol for this "character" is <CR>. This key is typically used to terminate a command or input specification.

Other important keys are described in sections where the software that uses them is described.

1.4. Using a Microcomputer as a Terminal

This section contains special advice for people who are using microcomputers as terminals. In order for this to work, you will need two things: a microcomputer with an "asynchronous port" (most computers have these), and software for terminal emulation. The software is often called a "communications package," because it combines terminal emulation with other communications functions. There are so many kinds of microcomputers and terminal emulators that we can't tell you exactly what to do. However we do have some general advice.

It is important for you to choose terminal emulation software that implements a terminal that Unix knows about. Otherwise Unix won't know what commands to send to your program to cause it to clear the screen, move the cursor, etc. So before you buy terminal emulation software, make sure you know what kinds of terminals it emulates. The most common one is "VT100". However there is a long list of acceptable terminal types, including VT100, VT52, and various Televideo, Wyse and Hewlett-Packard terminals.

The Rutgers Microcomputer Lab can give you public domain terminal emulation software for the IBM PC, Macintosh, and certain other common kinds of microcomputer.

Once you have the right software, you will have to connect your modem to your computer's asynchronous port, and then start the terminal emulator software. Most software has complex sets of commands and options to set various communications parameters. The major parameters are discussed above, in section 1.1. Often, microcomputers can be set up so that the computer automatically controls your modem. In this case the entire process of dialing our phone number, and making the connection to our modem, can be done by a single command. Unfortunately, it may take some trouble to get this set up properly. The Microlab is a good source of advice on this, if you are using software that they know.

1.5. Getting on the System

In order to use Unix, the system administrator or a TA must authorize you. As part of the authorization process, they will assign you a "user name" and a "password". Normally your user name is simply your last name. However there is a limit of 8 characters in a user name. So if you have a long name, it may be abbreviated. Also, if you have a common last name, it may have to be modified to give you your own user name. If you are John Q. Smith, your user name will probably by "jsmith", or even "jqsmith". Your password is a sequence of characters which the system uses to make sure that you are really you. You should keep this password secret, so that others can't use your user name.

The instructions above should have allowed you to get connected to the system. Normally you will go through a terminal server. Once you connect to your computer, you will get a greeting message, which will end with the prompt "login: ". (In a few cases, there will be terminals connected directly to the computer. In this case, when you hit the carriage return on the terminal, you'll see "login: " rather than a terminal server greeting.)

You should login by typing your user name and then hitting carriage return. The system will then ask "password: ". You should now type your password, again ending with a carriage return. Your password will not appear on the screen as you type it. This is to prevent other people from seeing it, in case someone is watching over your shoulder.

It is a good idea to change your password now and then. If you suspect that someone else is using your user name, you should certainly do so. The command "passwd" will ask you for your old password, and then allow you to type a new password. Since it does not echo the passwords, it will ask you for the new one twice. This is to make sure that you really typed what you intended to type.

1.6. Getting Out

When you are finished working on Unix, type the "logout" command. In order to do this, you will have to be at the main Unix prompt (normally "%" or "topaz>"). Note: when you are at the main prompt, a single ^D (control-D) will also log you out. Once you have logged out, you will be returned to your terminal server. E.g.

```
topaz> logout
[Connection to TOPAZ closed by foreign host]
hilltop>
```

You can now connect to a different computer, or type "quit" to disconnect from the terminal server. If you are connected via a phone line, it is safe simply to hang up at this point.

1.7. Software Settings

In order to help you do tell Unix about yourself, we have created a command called "newuser". We recommend that every new user should execute this command the first time he logs in. It does 3 things:

- It sets things up so the system will ask you what terminal type you are using, if it doesn't already know. To find out the legal terminal types, you can browse through the file /etc/termcap. The most commonly used types are:

```
dm2500    Datamedia 2500
heath-19  Heathkit (or Zenith) 19
v200      Visual 200
v55       Visual 55
i400      Infoton (or GTC) 400
gvt       Concept GVT or AVT (at Rutgers these are labelled
              "INFOCON")
vt52      Digital Equipment Corporation (DEC) VT52.  Use
              this also for DEC VK100 ("gigi").
vt100     Digital Equipment Corporation (DEC) VT100.  Use
              this also for DEC VT2xx series.
```

- It will set things up so that you get the prompt "topaz> " instead of "% " when the system is waiting for a command. We recommend using the system name as a prompt so that you always know what system you are talking to. We do so much with networks around here that it is easy to forget. Also, the prompt will change to "topaz 2> " if you are talking to a subsidiary command parser. This will help you realize that you are in an unusual context.

- It runs the program chfn, which will ask you for your name, address, and phone number. Since most users have their last name as their user name, there is a special abbreviation. "&" means to use your user name. Suppose I am Charles Hedrick, and my user name is hedrick. I would use "Charles &" as my Name. The system will automatically replace the & with my user name, appropriately capitalized. Other users can find out this information by using the command "finger". "finger hedrick" will print detailed information about the user hedrick, or "finger" alone will print briefer information about everyone who is logged in on the system. (The "who" command is a somewhat faster way to do this if you don't want the personal information.)

We strongly recommend that every new user type the command "newuser" to do these things.

The rest of this section contains details about how to do these setups yourself. You probably won't understand this until you have read the rest of this manual, and parts of the C Shell reference manual. Note that newuser sets things up so you don't have to do these things yourself. The rest of this section is for people who want to know how to do it themselves.

The most important thing to tell Unix is what kind of terminal you are using. If you don't do this, some programs may refuse to run. If you are on a terminal that is permanently connected to one of our terminal servers, it will get the terminal type from a file. However if you have dialed the system, or connected from another computer, the system has no way of knowing what kind of terminal you are using. Here is how to tell Unix your terminal type:

```
set term = v55
```

where you use the name of your terminal instead of "v55". Note that you must put spaces around the =.

You can type the "set term" command yourself after you have logged in. However many users prefer to do this automatically. Any command that you put in a file called .login will be done automatically when you log in. We recommend that you start out by creating a .login file that looks like this:

```
if ($term == "dialup" || $term == "plugboard" || $term == "network") then
  echo -n "Terminal type: "
  set term = $<
endif
```

This checks for the "dummy" terminal types, i.e. the names that the system uses if it doesn't really know your terminal type. If it finds one of them, then it asks you for your real terminal type. If you always use the same terminal type, then of course you can replace the two indented lines by "set term = TYPE", where TYPE is the type that you always use. The procedure "newuser" sets up a file .login as shown above.

There is a second setup file that you may want to use. This is a file called ".cshrc". This file sets up parameters for the command scanner, or "shell". It is possible for various programs to start shells for you. This file is used each time that happens, not just at login. The most common thing to put in this file is

```
set history = 20
```

This tells the system to remember the last 20 commands you have typed. As we will see below, this allows the history system to work. Some users also like to change the prompt. Normally the system shows a "% " when it is ready for you to type a command. Many users prefer to have a prompt that shows other information. We recommend a prompt that includes the name of the system. That way if you connect to another system using the network, you will always know what system you are talking to. We also recommend including a "level number". That way if a program starts up a subsidiary shell, you will know that you are talking to a lower-level shell, and not to your main shell. The following lines in your .cshrc will do that:

```
if ($?level == 0) set level = 0
 level = $level + 1
setenv level $level
if ($level == 1) then
  set prompt = "topaz> "
else
  set prompt = "topaz $level> "
endif
```

The "newuser" command sets up a .cshrc that includes both this and "set history = 20".

Some users don't like the idea that typing a single ^D (control-D) will log them out. If you don't like this, you should add a line "set ignoreeof" to your ".cshrc". If you have that, you will have to use the command "logout" to log out, and "exit" to get out of shells created by other programs.

2. An Introduction to Programming with Unix

Here is an outline of the process you will follow to write and debug a program:

- Log in.
- Use an editor (probably EMACS) to create a file that contains your program. Later, you will also use the editor to make changes in the program.
- Use a compiler to process your program so that the computer can execute it. The result is normally put in a file called "a.out".
- Run "a.out", probably under control of the debugger, "dbx". Using dbx, find any bugs that may exist.
- Repeat the last three steps until your program works.
- Log out by typing the command "logout".

Here is an example of a complete session. What you type is underlined. Things in the right column are explanatory comments.

4.2 BSD UNIX (topaz)

```
login: hedrick                                   my user name
Password:                                        it doesn't show
Last login: Mon Nov 19 12:00:09 on ttyi30
% ls *.p                                         see what Pascal files I have
test.p     trygraph.p
% emacs demo.p                                   create a new one

<<<<I am now in a video editor, creating a new file.  This
<<<<is impossible to show in a hardcopy document.  EMACS
<<<<is described below

% pascal -g demo.p                               try to compile it
        1  begin
        1  ^
demo.p : 1  *** ERROR *** syntax error
% emacs demo.p                                   I goofed.  Fix it.

<<<<I am now in EMACS again, changing the program to be
<<<<correct.  I can't show that here.

% pascal -g demo.p                               try compiling again
% a.out                                          compilation OK.  Run it

wrong answer

% dbx                                            use dbx to debug it
dbx version 145 of 8/11/85 11:56 (topaz).
Type 'help' for help.
reading symbolic information ...

(dbx) list
    1   program foo(output);
    2     var
...                                              first 10 lines of the program.
(dbx) stop at 7                                  place a breakpoint
[1] stop at 7
(dbx) run                                        start it
[1] stopped in main at line 7
    7   i := 1;
(dbx) step                                       execute one line
stopped in main at line 8
    8   j := 2;
...                                              continue looking around
stopped in main at line 9
    9   k[3] := 4;
(dbx) q                                          get out of dbx
% emacs demo.p                                   make another change
 ....
% pascal -g demo.p                               compile again
% a.out                                          run it again

correct answer

% ls                                             see what files I have created
a.out    demo.p    test.p    testdata   trygraph.p
```

% rm a.out testdata   get rid of what I don't need
% logout

Notice that Unix types out a prompt (normally "% " or "topaz>") and expects you to type commands. These commands are normally English words or abbreviations of English words, followed by arguments. Here are some typical Unix commands:

ls	LiSt the files on your directory
rm a.out	ReMove the file named "a.out" from your directory

Generally the Unix philosophy is to say as little as possible. If a command works, Unix just gives you another prompt. If something is wrong, you will get an error message. For example, if you say "rm assign1.p" and there is no such file, you will see an error message such as "rm: assign1.p nonexistent".

3. How to Type Commands to Unix

Before we go into the commands in more detail, let's look at how you type things. You type commands just as you would on a typewriter. Nothing actually happens to the command until you hit the "return" key. Until then, you can change your mind, or correct typographical errors. The "rubout" key will delete the most recent letter you typed. It will remove the letter from the screen, so you can see its effect. Each time you hit rubout, the cursor on your terminal will move back one letter. If you hit it enough times, the command will go away completely, and you can type a different one. However if you decide that you want to do a completely different command, there is an easier way: ^U will abort the whole line. It will move the cursor to a new line, and wait for you to start typing again.

^U is one example of a "control character". The problem is that there are not enough keys on the keyboard to do everything you want to do. So in order to do some things, you hold down a special key and type a letter or other symbol. This key is called the "control key". On many terminals, the control key is at the lower left of the keyboard. Note that the control key works much like the shift key on a normal typewriter: The control key alone doesn't do anything. You have to hold it down and hit another key at the same time. If you hold down the control key and hit the letter "U", this is referred to as typing "control U". More commonly this is abbreviated "^U".

There are two other control keys which are used commonly on Unix: ^C and ^D. ^C is referred to as the "interrupt" key. It interrupts whatever command or program is currently running. Shortly after typing it, you should see the "%" prompt again, and Unix should be ready for you to type another command. Certain programs process ^C themselves. For example in the debugger, dbx, ^C will interrupt whatever is going on. But it will return you to the command level of dbx, rather than the normal Unix command level. That is, you can now type another dbx command.

^D is the "end of file" signal. When a program wants you to type several lines, there has to be some way for you to tell it that you are finished. ^D does this. One of the more common cases where you use ^D is when you are sending mail. You might type the command "mail hedrick" to send mail to the user hedrick. At that point, the mail program would expect you to type a message. You can keep typing as many lines as you want. When you have finished the message, you type a ^D at the beginning of a line.

For your convenience, here is a summary of the special keys that are most commonly used with Unix:

return - causes the command you are typing to be executed
rubout - delete a character
^U - delete a whole line
^C - interrupt the current process
^D - end of file

One final word of warning: Unix is "case sensitive". That is, "rm" and "RM" are different words. Most commands should be typed in lower case. The command names themselves are all lower case. And most people use lower case for their file names. Of course you can call your file "AsSiGnMeNt1" if you want to, but if you do, you have to remember which letters are in upper case. The one exception to this is the so-called "options". There are a few cases where commands have options such as "-S". In general, the rule is simple: type things exactly as they appear in the manual. If the manual says the "rm" is the command for deleting files, type "rm", not "RM". If it says that "ls -R" is used to get a certain kind of directory listing, type the "ls" in lower case and the "-R" in upper case.

4. Files and Commands that Deal with Files

Most programs and data you use will be stored in "files". A file is an area of storage to which you assign a name. You can put anything you want into a file: a program, a set of data, or a paper you are writing. You usually decide what you want to call a file when you first create it and start putting data into it. However you can change the name later if you want to (with "mv").

Normally files are stored on "disks". A disk is a set of metal platters covered with the same material used to make common tape cassettes. Most Unix systems have disk drives permanently attached to them. Data is stored on the disk as magnetic patterns, much as music is stored on an audio tape or cassette. However a single disk will store many thousands of different files, owned by hundreds of different users. The trick is to keep track of them all.

Each file has a name. Normally the person who creates the file chooses the name. Names are from 1 to 255 characters, normally letters, digits, and some special characters such as "." and "_". There is a naming convention that you will do well to follow. Although it is not strictly necessary, it will help the system programs to figure out what you are doing. Under this convention, file names have at most one "." in them. The part of the name after the "." is called the "extension". It is used to describe what sort of thing is in the file. For example, "prog.p" is probably a Pascal source program. since ".p" is the extension normally used for Pascal. Similarly "prog.o" is probably a relocatable binary file, produced by compiling prog.p. Normally executable programs are given names without dots at all.

A collection of files on a particular disk belonging to a particular user, is called a "directory". The login procedure connects you to your own directory. When you refer to file names with no special qualification, you will get a file by that name in your directory. In order to refer to a file in another directory, you must specify the directory name in front of the file name. Directory names normally have "/"'s around them. For example, the file name "project.p" would be a file in your own directory. "/bin/rm" is a file in the directory "/bin". "/usr/ucb/mail" is a file in the directory "/usr/ucb". Normally you will be using files only in your own directory, except for systems programs that you run.

Most commands let you specify more than one file at a time. For example, to remove files "x" and "y", you could say "rm x y". There is also a shorthand that makes it easier for you to specify several files. The characters "*" and "?" in a filename act as "wildcards". A filename that includes one of them is actually a pattern. It expands to include a list of all files that match the pattern. For example, "rm *.s" would remove all files whose name ends in

".s". That's because * matches any character or sequence of characters. There can be more than one * in a name. E.g. "*pro*" will match all files with "pro" somewhere in their name. Since * will match the null string, this includes names that begin or end with "pro". The character "?" is used less often. It match any one character. So "rm ?.s" matches only files whose names have exactly 3 characters, the last of which are ".s"

In Unix, everthing on the system is considered to be a file. The things that aren't really files are typically allocated pseudo-filenames in the directory "/dev". Thus your terminal is "/dev/tty", and the tape drive "/dev/mt0".

4.1. Common Commands Dealing with Files

Here are the most common commands that deal with files. Note that the following list does not show how to create or edit a file. This is done with an editor, which is the subject of a separate section.

cp oldfile newfile
    CoPy a file

diff file1 file2
    shows the DIFFerences between 2 files

du   shows the Disk Usage on the current directory

grep string files
    search all the specified files for the specified string. (Actually, you can specify a fairly complicated search pattern, but you will have to do "man grep" for details on that.)

lpr file   print file on the Line PRinter

ls   LiSt all the files in your directory

more file   type a file on the terminal, stopping when the screen is full. Hit the space bar to get it to continue. (It is called "more" because when the screen is full, it says "more: " to tell you there is more to come.) This is the way that most people type out files.

mv oldfile newname
    MoVe a file (i.e. give it a new name)

rm files   ReMove files

In order to keep things simple, we have not shown you the options for most of the commands. However the "ls" command is used so much that it is worth taking a bit of time to explain it in more detail. If you just type "ls", it will list all of the files in your directory. The first option you have is to give it a filename to look for. "ls *.p" will show you all files than end in ".p". "ls /bin/*" will show you all files in directory /bin. So far all you will have seen are lists of filenames. It is also possible to ask "ls" to give you more information about a file. That's where the options come in. Options give extra details about how you want the command to be executed. Normally the options come right after the name of the command, with a hyphen in front of it. For example, "ls -l" will give you a "long-form" directory listing. For each file, this will show its size, creation date, owner, etc. If you wanted to see this information for all files ending in ".p", you would put the option before the file names, i.e. "ls -l *.p". If you want to specify more than one option, put all of the letters after the hyphen, e.g. "ls -lt" will give you a long-form listing, sorted by creation time. This is useful for reviewing the files you have created recently, since those will show up first. Here are the most common options for ls:

```
ls [files]      LiSt files [default: all files in directory]
  -a              all files [normally files starting . not shown]
  -l              long form - give lots of information
  -t              time-sorted - most recent first
  -R              Recursively look into directories
```

The "-a" option requires some explanation. Normally filenames starting with "." are for files that you don't normally want to see. These include .login and .logout, which list commands that get done automatically every time you log in or out, and files giving default options for various other programs. The designers of Unix figured you would want to set up these options and then forget about them. Thus a normal "ls" will not show files whose names start with ".". "ls -a" will show all files, including those that start with ".".

4.2. Disk quotas

In order to prevent one user from hogging the whole disk, we have disk quotas on many of our systems. You are not allowed to use more space on the disk than your disk quota. To find out what your quota is, type the command "quota". Here is an example of what you will see:

```
Disc quota for hedrick (uid 171):
  Filsys current quota limit #warns files quota limit #warns
    /u2    398    400   2000          91    0     0
```

(Note that the system uses the Australian spelling of "disc". This is because the disk quota code was implemented Down Under.)

Filsys is simply the name of the disk system you happen to be on. It will probably be either /u1 or /u2.

Current is the amount of disk space you are using now, in KBytes (1024 bytes). It should be roughly the sum of the file sizes shown by "ls -l". However files are allocated in units of 2 KBytes, so there will be some rounding up.

Quota is the amount of disk space you are supposed to use. Whenever your current usage goes over this number, you will get a message "WARNING: exceeding disc quota on /u2". We realize that you often need more space than your quota temporarily, so this is just a warning. Your program will continue.

Limit is your absolute limit. If you go over that you will get a message like "DISC LIMIT REACHED (/u2) - WRITE FAILED". Your program will probably blow up at this point. The limit is normally 5 times the quota.

#warnings will be explained below.

The second set of numbers (starting with files) refers to a separate quota on the number of files you can create. We do not use this, so those quotas and limits will always be 0.

In order to keep people honest, we have to make sure that you don't stay over your quota too much. The idea is that you are supposed to get your usage down to the quota when you log out. In login, this is checked. If you are over quota when you log in, you will get a warning message. It will say something like "Warning: too much disc space on /u2, 2 warnings left." If you use the quota command, you will now see a 2 in the warnings column. Each time you log in over quota, the number of warnings will be decremented. If it goes to zero, the system will conclude that you are a hopeless pig, and prevent you from creating any more files until you fix the problem. At that point, you will have to delete enough files to bring your usage down to the quota. You will then have to log out and log in again.

There is a possible problem with quotas. Current usage refers to all files that you own. That is, all files that show your name in the owner column when you do "ls -l". You own every file that you created. You also own your own mail file. So if someone sends you a lot of mail, this can put you over your quota. Also, if you copy a file into someone else's directory, you still own it, and it is still counted against your quota. If you want to give someone else a file, you should let them copy it. Note that current usage also counts files that have not yet been closed. If a program is still running, it may be using disk space that does not show in an "ls -l" command.

************ NOTE ***************

The rest of this chapter describes more details about the file system. On your first reading, you may want to skip to the beginning of the next chapter. Then again, you might at least look at the section headings you are skipping, just to see what you're missing.

4.3. More about Directories, Subdirectories, and "cd"

Every user has a "home directory". This is a directory that is associated with the user name. Normally it has the same name as the user name, although there may be another directory or two above it. For example, the user hedrick might have a home directory "/u2/hedrick".

Most users end up with so many files that it is hard to keep track of them all. They end up wanting to create more directories. For example, they may have separate directories for each course they are working on. Normally these additional directories are put within the user's home directory. Thus if hedrick is taking courses 431 and 435, he might create directories /u2/hedrick/cs431 and /u2/hedrick/cs435.

To create a directory, use the command "mkdir". For example, "mkdir /u2/hedrick/cs435". As with file names, you often will not bother to type out the entire directory name. If you are currently connected to /u2/hedrick, it would be easier to type just "mkdir cs435".

At any moment you are always connected to a current directory. If you mention a file name but don't specify a directory, it refers to this current directory. That is, if you are connected to "/u2/hedrick", then the filename "prog.p" means "/u2/hedrick/prog.p". You connect to a directory by using the command "cd". The full form would involve naming the whole directory, e.g. "cd /u2/hedrick/cs431". As usual, you will often not bother to do this. For example, if you were already connected to /u2/hedrick, then "cd cs431" would connect you to "/u2/hedrick/cs431".

The "pwd" command prints the current directory, in case you have forgotten. (pwd stands for Print Working Directory.)

Be sure you know where you are before you do mkdir. Otherwise you can have unintended effects. For example, suppose you were connected to /u2/hedrick/cs435 and you did "mkdir cs431". You would end up with /u2/hedrick/cs435/cs431. Of course if you want to be safe, you can use the full form: "mkdir /u2/hedrick/cs431".

The abbreviation ".." means the directory above the current one. This can be handy for getting back up to your home directory. If you were in /u2/hedrick/cs431, then "cd .." would get you to /u2/hedrick. You can use .. with other things, e.g. "cd ../cs435" would take you up to /u2/hedrick and then back down to /u2/hedrick/cs435.

The abbreviation "." means the current directory. This isn't normally needed, since this is the default. However a few commands require you to specify a directory names, and "." can be useful then.

4.4. More About Specifying Other Users' Files

It would be nice if the way to access Smith's files was always to give a file name starting with /smith. Unfortunately, things are not so pleasant. For efficiency reasons, Unix system managers do not generally put everything on the same disk or disk partition. Thus you are likely to find one more level of directory than you would expect. Typically users' directories have names like /u1/smith and /u2/jones. The top level directories actually represent separate disk drives or at least separate physical parts of the same disk drive. On most large Unix systems, there are at least 3 separate disks or disk partitions: /, /usr, and at least one more partition for user directories. None of this matters particularly to you, except that it makes it harder for you to figure out where to look for a given user's files. For this reason, there is a special syntax: ~user. ~jones means the home directory for the user named "jones". It expands to /u1/jones, /u2/jones, or whereever jones may happen to be. So to refer to the file prog.p on hedrick's directory, you would type "~hedrick/prog.p".

4.5. Some Useful Details: Ownership, Protection, Privacy

Unix allows you to control who can access your files. If you don't do anything special, anyone on the system can read or execute any of your files. However you can do things to exercize more control over file access.

You can control access at two levels: the individual file, or the whole directory. In order to get to one of your files, someone else must have the authorization to look in your directory. If he can't look in your directory, then nothing else matters. Once he can look into your directory, then each individual file has its own code to say what he can do. So you can lock out users from individual files by changing the file permissions. Or you can lock them out from the whole directory by changing the directory's permissions.

Some aspects of file protection depend upon "groups". A group is a set of users. It has a group id, which is a number, and a group name. The groups are set up by the system administrator. Normally they represent classes and research projects. So there is a group "cs431-a", which contains all students in Computer Science 431, section A. There is also a group "vlsi-design", which contains all users in the VLSI design project. One of the things you can do is to say that everyone in a certain group can access your files. If you find that you need a group that is not already defined, you should contact your system administrator. The groups are listed in the file /etc/group.

In order to provide you with maximum flexibility, Unix allows each file to have three attributes that affect who can access it.

- owner - this person can normally delete or modify the file, and can change its protection and the group that is associated with it. The owner is normally the user that created the file. This means that if someone else puts a file into your directory, he owns it! (As a special convenience, you can always delete a file that is in your directory, even if its permissions would not normally allow you to.)
- group - different files can have different groups associated with them. You would normally choose a group that you want to have more access to the file than the rest of the world. For example, you might decide that members of your research project should be able to change the file, and everybody else should just be able to read it. Then you would set the group to your project's group. If you don't specify a group, it defaults to the group associated with the directory. The owner can change a file to any group that he belongs to.
- mode - the mode is a number that controls who can do what. It has separate fields for the owner, the group, and others. Thus the owner could allow himself to change the file, members of his group to read it, an other to do nothing at all. Or any other combination...

The commands "chown", "chgrp", and "chmod" are used to change the owner, group, and mode of a file. (However only a superuser can use chown.)

Although the mode is actually a number, you normally look at it by using the "ls -l" command. It displays the mode in a form that looks like the first line below.

```
-rwxr-x---
 xxx       owner
    xxx    group
       xxx others
 r   read
 w   write
 x   execute (search for directory)
```

There are three groups of three letters. The first group applies to the owner, the second to the group, and the third to all others. The letters mean read, write, and execute. So the code "-rwxr-x---" means that the owner can do anything; others in his group can read or execute the program, but not change it; and others can't do anything with it at all.

As mentioned above, in order to be able to access one of your files, a user must have the appropriate permission in the file mode, but he must also have access to the directory itself. The protection bits have a slightly different meaning for directories. Read allows the user to read the directory as a file. In practice, this controls whether he can do wildcard operations such as listing all the files in the directory. Write controls whether he can do operations that involve changing the directory, i.e. create or remove files. Execute controls whether he can open existing files. If execute is missing, the user cannot open any file in the directory, even if the files themselves have permissions that would allow him to do so. If the execute bit is on, then the user can do operations on individual files according to their permissions.

Typically directories are set up to have modes "-rwxr-xr-x", which means that the owner can do anything, and everyone else can access files according to their individual modes. The "r-x" allows other users to look into the directory and at the files in the directory. Because "w" is missing, the other users can't create new files in the directory. If you wanted to have a totally private directory, you would use "-rwx------". In this case, no one else could look at anything in it. If you wanted members of your group to be able to create files in your directory, you might use "-rwxrwxr-x".

4.6. I-nodes and Links

In Unix, file names and files are conceptually separate. It is perfectly possible to have two different names for the same file. In that case, there are simply two different directory entries that point to the same file. In order to avoid confusion, the term "i-node" is used to refer to the file as a physical entity. So we would say that a file with two names is an "i-node" having two directory entries that point to it. We refer to the connection between a directory entry and an i-node as a "link".

The "ln" command can be used to link a new name to an i-node. The format is "ln old-name new-name". Note that it is perfectly possible for the two names to be in different directories. So you and your friend could share a file by using the "ln" command to create a second link to it. Note that the ownership information is part of the i-node. So the one that created the file initially would still own it. But it would appear identically in both of your directories. The file will continue to exist until both of you delete it. That is, the "rm" command really only removes the link from a name to an i-node. The i-node goes away when there are no more links to it.

There is an additional kind of link, a "symbolic link". This is a link from one name to another name, instead of from a name directly to an i-node. Suppose you wanted to make "myprog" in your directory point to your friend's version: "~smith/myprog". You could of course say "ln ~smith/myprog myprog". However if your friend ever deleted myprog and created a new one, your link would continue to point to the same i-node, namely the old file. So you might prefer to use a symbolic link: "ln -s ~smith/myprog myprog". This says that myprog in your directory is linked to ~smith/myprog, whatever it may happen to be at the moment. That is, instead of being linked to a particular physical file, it is linked to the name. Most people do not use symbolic links. A normal link (technically called a "hard link") will normally do the right thing. There are special provisions in the editors so that when you edit a file that has more than one link, all of the links are maintained. But there are some situations where a symbolic link may make sense.

Unless you are a super-user, you can't create a hard link between two directories. So one of the more common uses of symbolic links is for directories. Suppose you have a friend who is working with you on files in the directory /u2/smith/cs431/source. You may not want to type all of that every time you want to access one of his files. So you might do "ln -s /u2/smith/cs431/source smith". Then if you refer to "smith/foo" you will really get "/u2/smith/cs431/source/foo".

5. Commands

The commands that you type to the system are processed by a program called the "shell". The shell reads your command, and then calls a program to execute it. The shell can't do very much on its own. But it is an expert at coordinating other programs. There are several different shells used on Unix. They all understand the same basic commands, but they have different additional features. The shell that we use at Rutgers is the C Shell. Users from Bell Labs or ATT may be used to the Bourne shell. If you want to use that, you can use the command "chsh yourname /bin/sh". chsh changes your default shell.

You will normally be talking to the "top-level" shell, that is the shell that is created when you log in. However some programs have commands that create another shell. You would normally use one of these commands if you are in the middle of something and suddenly need to do an "ls" or some other command. When you are in one of these subsidiary shells, the "exit" command will get you back to the program that called the shell. If you have not set ignoreeof, ^D will also exit from a shell.

Normally what you type to the shell is the name of a program followed by arguments. For example, when you type the command "rm *.p" (which deletes all files ending in .p), this is really a request to run the program "rm". Normally, the shell looks on your own directory first to see whether you have a program called "rm". If not, it then looks on a set of system directories where commands are normally kept. That allows you to create your own version of commands, which you will get instead of the normal ones. If you want to execute a program from someone else's directory, you can give a name that specifies the directory, e.g. "/u1/hedrick/rm *.p" would run a program from the directory "/u1/hedrick". The exact list of directories where the shell looks is defined by a variable called your "path". To change it, you can use a command such as

```
    set path = (. /usr/local/bin /usr/ucb /usr/bin /bin /etc)
```

"." means the current directory.

The shell processes the wildcards, * and ?. So if you type a command such as "rm *.p", the shell looks in your directory for all files ending in ".p". It then run the program "rm", passing it an actual list of files, such as "lastprog.p prog1.p prog2.p". Thus means that individual programs don't have to have special provisions to deal with wildcards.

You can tell the shell to direct a program's primary input and output to something other than the terminal. This can be useful in case you want to put output on a file. For example, suppose you want to put a list of your files into "filelist". You could use the command

```
ls >filelist
```

The ">filelist" tells the shell to redirect the output from the "ls" program to the file "filelist". You can also redirect input, by using "<".

You can direct one program’s primary output directly into another program’s primary input. (The device used for doing this is called a "pipe".) For example, suppose you have so many files that the "ls" command fills up your screen 3 times. You might do the following:

```
ls | more
```

This directs the output from "ls" into the program "more". More is a program that prints its input, stopping every N lines (where N is the length of your terminal screen). You can construct long chains of programs in this way, e.g.

```
ls | sort | more
```

which passes output from "ls" through "sort", and then to "more".

Many commands let you specify options. Normally options begin with a "-". For example, the "ls" command has options "-l" (long form listing) and "-t" (sort in order of creation time). You normally put options before any other arguments. Thus the following commands would be legal:

```
ls -l
ls -l *.p
```

There are two different ways of specifying more than one option. Either you can specify the options separately, e.g. "-l -t", or you can combine them into a single option "-lt". Many commands accept either one. However you may run into occasional commands that require one or the other syntax. The documentation for each command describes the exact syntax that it is prepared to accept.

The following sections describe some more sophisticated features of the C shell. You may not want to read them the first time through. If not, skip to the next chapter.

5.1. Job control

The C shell allows you to run and keep track of more than one process at a time. For reasons known only to the folks at Berkeley, the processes running under control of the shell are known as "jobs". This terminology is potentially confusing, since the term job is more commonly used to refer to an entire session, i.e. to everything that you are doing.

When you type a normal command, the shell starts the program you have requested, and waits until it is finished. If this were the only tool available to you, you could only do one thing at a time. You can interrupt a process by typing ^C, but this kills the process. So this still doesn’t let you do more than one thing at a time.

The first, and most common, way to do more than one thing is to put a "&" at the end of a command. E.g. suppose you want to get a list of all the files on your directories, and you don’t want to wait for this to finish. You might type

```
ls -R >files &
```

The & specifies that the shell should start the command but not wait for it to finish. You can now issue more commands, and forget that the "ls" is still happening. A program that is running this way is "in the background". That is, it does not interfere with anything else you might be doing.

There is some question about what should happen to the terminal when a background job is running. In the example I just showed, the "ls" command sent its output to a file. But suppose you did "ls -R &". This command would still run in the background, and you could still proceed to do other things, but all the output from the command would come to the terminal. Since "ls" produces output continuously, it wouldn't be very practical for you to try to do anything else while it was running. The situation is even more interesting if the program reads from the terminal. Suppose a background program tries to read from the terminal at the same time that the shell is waiting for a command. You would have no way to know where your input is going to go. For this reason, the system will normally stop a background program when it tries to read from the terminal. The assumption is that you will eventually notice that the program is waiting for input, and will continue it as a normal "foreground" program.

The other way to get more than one thing happening at a time is the ^Z command. This stops the currently running job. But unlike ^C, it leaves the job around. You will be able to continue it later. Suppose you are typing a mail message and realize that you have to look something up in order to include it in the message. You can type ^Z. This will stop the mail program, but leave it around. You then look at whatever you need. You can come back to the mail program by typing "%mail". You can also continue a job in the background by doing "%name &". In the case of mail that wouldn't be too useful, since mail is going to need input from the terminal. But some programs ask a few questions and then go off and compute. In that case, you might run the program, answer the questions, and then type ^Z followed by "%name &" to get it into the background.

Before you can really use all of these commands, you need to know how to refer to a job. To get the terminology straight, let's start by looking at the output of the command "jobs":

```
% jobs
[1]    Running              ls -R /
[2]  - Stopped              mail hedrick
[3]  + Stopped              emacs
```

Here we see a situation where there are 2 jobs stopped by ^Z and one running in the background. (It must be in the background, since otherwise I couldn't ever have typed the "jobs" command.) There are four different ways to refer to one of these jobs:

by job number, in this case %1, %2 or %3. A % is always used to indicate this type of notation.

by job name. This is the name of the program that is running, again with a %. So the job names would be %ls, %mail, and %emacs.

by system-wide process id. You would have to do a "ps" command to find out the pid for these processes. Even then, it wouldn't be all that useful, since only certain commands (most commonly, the "kill" command) take pids as arguments.

by default. Some of the commands default to the "current job". This is the one with the + next to it. The - is the previous job. This would become current if the current one finishes.

There are actually two different ways to continue a job. You can simply use the job name as a command, e.g. "%3" or "%emacs". Or you can use the "fg" or "bg" commands, which continue a specified job in the foreground or the background. "fg" and "bg" default to the current job. If you have just stopped a job, "fg" is probably the most convenient way to continue it, since you don't have to think about what to call it. But if you want to continue some previous job, it is probably easier to type a command like "%emacs", rather than "fg %emacs".

Two other useful commands commands that deal with processes are "kill" and "stop". Both of these allow either %name or %number. Kill also allows a system-wide pid. In general, kill is equivalent to ^C. That is, it gets rid of a job permanently. You would normally use it to kill a job that is stopped or running in the background. (If the job is running normally, you would just type ^C.) Some programs will trap the kill command. If you want to get rid of them, you have to use "kill -9", e.g. "kill -9 %emacs". This can be slightly dangerous. Normally processes trap a kill because they want to be able to clean up after themselves. If you use kill -9, they may leave temporary files lying around, etc.

The stop command is equivalent to ^Z. You would normally use it for a job that is running in the background. (It wouldn't make any sense for a job that is already stopped, and for a normal job, you would just type ^Z.)

6. Using the Documentation and Help Facilities

Unix has fairly complete documentation online. That is, for each command or program, there is supposed to be a "manual page" that describes it. To access these manual pages, you use the "man" command. For example, "man ls" prints the manual page describing the "ls" command.

In order to use the man command, there are some things you should know. First, for substantial programs, the manual pages are not complete. The intent is that they should tell you how to run things, but they should be short enough to read on your terminal. So "man pascal" will tell you how to run the Pascal compiler, and list all of the options. But it will not print out a complete Pascal language reference manual. There are separate manuals for the languages, the mail system, the document processors, and certain other complex pieces of software. Generally the manual page will refer to this separate documentation if it exists. Larger documents are normally put on the directory /usr/doc.

Second, the manual pages are organized into chapters. It is sometimes helpful to know what the chapters mean. For example if you ask "apropos" what documentation there is on "unlink" it will offer you man pages from sections 2 and 3. It is useful to know that section 2 is system calls and section 3 is subroutines. So if you were looking for the Fortran subroutine, you would want the one in section 3. Here are the chapters:

1	Commands
2	System calls
3	Subroutines
4	Special files [these are wierd devices, and are unlikely to be useful to normal people]
5	File formats and conventions
6	Games
7	Macro packages and language conventions
8	Items of interest to system managers

So if you are looking for commands and programs, you will only be interested in documentation from section 1.

If you want to get an overview of what is in a given section, you can use a command like "man 3 intro", which will print the introduction to chapter 3. Note that a few sections (particulary 3) have subsections which are described by letters:

3F	Fortran library
3M	Math library
3N	Internet network library
3S	C standard I/O routines
3X	other libraries

You can get an overview of one of these subsections by typing a command like "man 3F intro".

The "man" command is useful for finding a description when you know what something is called. But suppose you want to remove a file, but you can't remember the name of the command that does that. The "man" command can't help you, because you don’t know what to look up. "apropos" is designed for these situations. apropos searches an index containing the titles of all of the man pages. It prints all title lines containing its argument. So "apropos remove" would print all title lines that contain the word "remove". Generally the title lines are written so that they contain most of the keywords people would use for looking them up. Of course you do still need to know something about Unix terminology. (E.g. you have to know to look for "remove" instead of "delete".)

By a judicious combination of "apropos" and "man", you should be able to find anything you want to about the system. What these will not do is give you overall system conventions, and the context you need to make sense out of these pages. (That is what this document is for.)

7. Editors: Creating and Changing Files

When you want to create a new program or change an existing one, you will use an editor. An editor is a program that reads in a file, lets you modify it, and then writes out an updated copy of it. The editors that we use are "full-screen" editors. This means that they display a piece of your file on the screen. In order to change or add to the file, you move the cursor around the screen until it is pointing to the place where you want to make the change. Then you use editor commands to delete and/or insert things. The changes that you make show up immediately on the screen. Of course it is also possible to move the "window", i.e. to change which portion of the file is showing on your screen.

When you are in an editor, you are actually working with a copy of the file. That is, the changes that you make do not show up in the file itself, but rather in a copy of that file that the editor maintains in memory. When you are satisfied with your changes, you instruct the editor to make the changes in the permanent copy of the file. Sometimes something horrible will happen while you are editing. If you want to forget all of your changes, you can exit from the editor without writing the changes back into the file. In this case, the file will not be changed at all.

Editors are very different in their commands from the rest of Unix. In most programs, commands are words or abbreviations. They are designed to be easy to remember and read. However in an editor, commands are usually single characters. Initially, this makes most editors look somewhat cryptic. However there is a good reason. You will be using the editor so much that you would quickly become tired of long commands. For example, if you want to move the cursor down 4 lines, it is very easy to hit the downarrow key 4 times. How would you like to have to type the word "down" 4 times? Normally commands are typed "in the dark". That is, you don’t see the command on the screen as you are typing. Instead, you see its effect. For example, when you type a delete command, the character you are deleting disappears from the screen. You don’t see the delete command itself.

Most of our users will use the editor EMACS. So the following section describes it in more detail. There are several other text editors available on the system. You can use any of them that you prefer. One advantage of EMACS is that it is available on almost every system at the University, although there are slight differences in the different versions.

7.1. EMACS

To edit a file using EMACS, just type "emacs filename". This works whether the file exists or not. If the file does not exist, it will be created the first time you use a command that updates the file.

You may get an error message "emacs: terminal "xxx" isn't powerful enough to run Emacs." Normally this means that the system does not know what kind of terminal you are using. See section 1.7 for instructions on how to set up your terminal type. (That section recommends that you execute the command "newuser". If you do, you will have to logout and log back in before your terminal type will get set.)

The best introduction to EMACS is the EMACS tutorial program, "teach-emacs". To enter the tutorial, type the command teach-emacs from Unix command level. The following sections will give you a quick introduction, which should be enough to let you do some useful work.

In EMACS, there is no distinction between "command mode" and "insert mode", as there is in most editors. Whenever you type a normal printable character, it is inserted in the file immediately. The commands are all control characters or something equally unusual. Since you normally don't want to put control characters into your text, it is easy enough to tell what things are commands and what are text that you want to go into your file. So in EMACS, to add some new text, you just get to the right place and start typing. To execute a command, just type it.

As it turns out, there weren't enough different control characters to satisfy the designers of EMACS. They wanted more commands. So there are also some prefix characters that are used to make up 2-character commands. The most common ones are "escape" and "^x". "escape" following by anything is a command. So is "^x" followed by anything. (Escape is a special key, normally labelled "ESCAPE" or "ESC". On some older terminals it is labelled "ALTMODE" or "ALT".) Actually escape has one other use: escape followed by a number sets up a count for the next command. For example, "^d" deletes the next character. "escape 5 ^d" deletes the next 5 characters. [It's hard to write these key sequences. "escape 5 ^d" represents three keypresses: escape, 5, and control-d.]

The most common commands are put on a separate "keypad". This is a special set of keys to the right of the normal keyboard. On most of our terminals, the keys are labelled with the digits, and look like the keyboard of a desk calculator. When you use them in EMACS, you should ignore the labels. The most important keys are used to move the cursor around the screen.

```
                what they do                how they are labelled
                     up                                8
        left                 right                4             6
                    down                               2
```

Here are enough other commands to get you started doing useful work.

^v	move forward a screenful
escape v	move back a screenful
^s	search for a specified string; asks for string
^d	delete a character
rubout	delete previous character
^k	kill (delete) the rest of the current line
^a	go to the beginning of the line
^e	go to the end of the line
escape <	go to the beginning of the file
escape >	go to the end of the file
^o	open up a new line
^y	put back the most recently deleted lines
^x^s	make the changes in the file

^x^c exit

It is probably useful to give some additional information about some of these commands. There are two reasonable ways to add a new line. One is to go to the end of the line before the place where you want to put it and hit the carriage return to create the new line. The other is to go to the beginning of the line after it and hit "^o". In either case, the cursor will be left on the new line, ready for you to type in new text.

^k has a somewhat odd but useful definition. If you are in the middle of a line, it deletes the rest of the line. If you are at the end of the line, it deletes the newline at the end. That is, it joins the next line to the end of the current one. If you want to kill the whole line, this means that you should get to the beginning of it and do "^k" twice. The first time clears the line. The second time gets rid of the newline at the end.

^k can be used with ^y to move text around. When you kill lines with ^k, they are remembered in a "kill buffer". You can then to somewhere else and do ^y. This will cause the lines that you killed to be resurrected at the current location. You can put the same lines several places by doing ^y in several places. NB: If you do a few ^k's, then move the cursor, then do a few more ^k's, only the last set of lines killed will be in the kill buffer. Of course if you know how many lines you want to move, it is easier to do "escape 5 ^k" than ^k 10 times. (10 times because you have to type ^k twice to kill a single line.)

To exit from EMACS, first write out the updated version of the file by typing "^x^s", and then leave by typing "^x^c".

EMACS protects you against various kinds of disaster by keeping backup copies of your file. These have the same name as the original file, with .BCK or .CKP added to the end. If may find it a good idea to do "rm *.BCK *.CKP" now and then.

8. Compiling and Running Your Program

You will start by using EMACS to create a file that has your program in it. You should probably give it a name that shows the language it is written in (.p for Pascal, .f for Fortran, .c for C). To compile the program, you then use the appropriate compiler command:

```
language    extension   compiler
 Pascal       .p         pascal
 C            .c         cc
 Fortran      .f         f77
```

The compilers all act in roughly the same way. The rest of this section will use C for examples, but the same instructions should apply to any of the other languages, using the information from the table above.

Suppose you have a program in a file "prog.c". Before executing it, you have to call the C compiler. Use the command

cc -g prog.c

This will translate the program from source into a binary executable file called "a.out". (This is simply a default. You can choose your own file name if you want to.) The option "-g" tells the compiler that you want debugging information put into the program. This allows you to use the debugger "dbx". If you omit "-g" the program will still run, but you will not be able to use dbx with it. (Some compilers on the Pyramid want "-gx" rather than "-g". This changes over time and from machine to machine. Use the "man" command to find out which option your compiler wants.)

Now that you have the executable file, you can run it either directly or under the control of dbx. To run a program directly you simply type its file name, in this case

```
a.out
```

To run it under the control of dbx, you use the dbx command. Normally you would have to tell dbx the name of the program. But it uses a.out as a default, so in this case, all you need is

```
dbx
```

If you simply run your program, there is a reasonable chance things will really end this way:

```
% cc prog.c
% a.out
segmentation error - core dumped
```

It seems worth mentioning some of the things that can go wrong, and what to do about them.

8.1. Errors while compiling and loading

When you type "cc prog.c", several things are happening. First, the C compiler reads your program and produces an assembly-language translation. Then the assembler reads that and produces a relocatable binary. Finally, the loader reads that, merges it with any libraries that your program needs, and produces an executable binary. Any one of these stages can produce error messages.

Error messages from the compilers usually give you line numbers in your file. Here is a typical error from the Pascal compiler:

```
pascal demo.p
          5  begin
          5  ^
demo.p : 5  *** ERROR *** syntax error
```

Note that this shows you the name of the file and the line number within the file. Of course this is just the place where the compiler noticed the error. Often the actual mistake is on the previous line. (In this particular case, the error is that the user left out the PROGRAM statement. This error is not detected until the BEGIN at the start of the main block.)

The assembler should never produce an error message. If it does, something is wrong with the compiler. An error message from the assembler will begin with "as: ".

Error messages from the loader normally indicate that you called a procedure that you didn't supply. Here is a typical error message:

```
ld:
  _exp undefined
```

This may indicate that you typed the name incorrectly, or that there was a library that you forgot to include. For example, in C if you use a math routine (e.g. SIN or COS), you have to tell cc that you need the math library:

```
cc -g prog.c -lm
```

"-lxxx" indicates that you want to use the library libxxx. The math library is libm, so you specify "-lm". Pascal and Fortran automatically include the math library if necessary, but there are other libraries which are optional for all 3 languages (e.g. curses).

In Fortran, a typo in an array name can also result in an error from the loader. If you type

```
x = myaray(i,j)
```

and myaray doesn't match an array declared in a dimension statement, Fortran assumes you are trying to call a subroutine. If there is no such subroutine, then you will get an error from the loader.

8.2. Errors at runtime

Under Unix, most errors eventually result in some sort of memory protection error. This may show up as "segmentation fault", "bus timeout", or various other things. These problems occur because the program attempts to reference a memory location that is not part of the program. Typically this occurs because a register that you use for a base register or for indirection has an illegal value in it. This can be the result of having garbage where you expected a pointer, or having an index out of range.

Generally the only practical way to find out what is going on is to use the debugger "dbx". You can use it in either of two ways. First, you can run your program under the control of "dbx". To do this, use the command "dbx" to run your program. You could also say "dbx a.out", but "a.out" is the default filename for "dbx", so you don't need to mention it. "dbx" will load your program and await a command from you. You may put breakpoints in and look around before starting your program, or just run it. When you get ready to run it, type the command "run" to dbx. If something goes wrong, dbx will trap the error, and will show you the offending line. At that point, you should use the examine command to look at the values of variables. You will probably find a pointer with a 0 value, an index that is out of range, or something of the kind.

For actual dbx commands, see the dbx manual, included with this document. One thing that the manual doesn't mention is the abbreviations. From the manual it looks like you have to type "step" every time you want to execute a line of your program in single-step mode. In fact you only need "s". The most common commands have one-letter abbreviations (usually the first letter of the command).

If you are having trouble figuring out why your program is behaving as it is, you have two major tools available: breakpoints and single-stepping. A breakpoint is a place where the program will stop. You must set up a breakpoint before you start the program. To do so, use the b command, e.g. "stop at 100" will set a breakpoint at line 100. If your program ever reaches that location, it will stop, and "dbx" will get control. You can then look around and do anything else you want. To continue the program, type "continue".

If you get really desperate, it is also possible to ask dbx to run your program line by line, showing you each one as it is about to be executed. To do this, use "step". Each time you type "step" one line will be executed. You must be at a breakpoint before "step" will work. If you want to do it from the beginning, put a breakpoint at the beginning and then start the program with "run".

One last thing: To get out of "dbx", use the "quit" command. ^C will not get you out.

9. Mail

The system has a facility for "computer mail". This lets you leave messages for other users and read those for you. Before sending mail, you should know what computer mail addresses look like. Most of the time you are sending mail to another user on the same computer. In this case, you simply use the person's user name. For example "mail smith" will send mail to the user smith on the same computer system that you are currently using. (The user doesn't have to be logged in at the moment. But he must eventually log in on this computer.) The rest of this section describes how to address mail to people on different computers, and to set up mail forwarding, etc. The first time you read this, you may want to skip directly to the section on using the mail program.

If the person uses a different computer, his computer mail address will involve both his user name and the name of the computer that he is on. For example, user "hedrick" on the machine "topaz.rutgers.edu" will have the mailing address "hedrick@topaz.rutgers.edu". All computers at Rutgers have names that end in .rutgers.edu. If someone asks you for your computer mail address, you can use the command "hostname" to find the name of your computer. If you are using a diskless Sun workstation, mail will actually go to one of the file servers, not the individual workstations. However this redirection is performed automatically. (It is probably slightly preferable to find out the name of your mail server and use that as the address.)

If you want to send mail to a user on another Rutgers machine, things are fairly easy. You just use an address like user@machine.rutgers.edu. (In fact, you can actually omit the .rutgers.edu, and just say user@machine) However machines at other universities can be a problem. To start with, ask the person for his computer mail address. If he gives you an address that looks like ours, you can probably just use it. Most universities will have addresses ending in ".edu". Other kinds of lab may end in ".gov", ".com", or ".org". If the computer name (the part after the @) does not have a dot in it, it is out of date, and will not work. You should ask your correspondent to check with his support staff, and get his "domain-based" address. If the address ends in .bitnet, .csnet, or .uucp, then his machine is on a different computer mail network. However our software should handle the routing automatically. If the address has a "!" or "%" in it, it is an old format address, and may or may not work. For help in getting mail to an address, send mail to postmaster. (Every machine should have a user called postmaster.)

Note that names with ! in them need special care. ! has special meaning if you type it to the shell. Thus you must quote it by putting a "\" in front of it. That is, to send mail to "seismo!rick", you would type

```
mail seismo\!rick
```

This problem happens only when you are typing a command to the shell. If you are inside a program, e.g. mail or Emacs, you do not need the \. \ is not an actual part of the address. It is simply a way of getting ! past the C shell. If you forget to use \, you will get the error "event not found".

There are three additional forms of mail addressing: forwarding addresses, aliases, and mailing lists. Forwarding is something you set up if you want your mail to be forwarded to a different address. For example, suppose you have accounts on several different computers, but you only want to read your mail on one of them. You can set up forwarding on the other computers. To set up forwarding, you create a file called ".forward". This file contains exactly one line. On that line is the address to which you want your mail forwarded. Note that the file .forward must be created in your home directory, i.e. the directory you are in when you first log in.

An alias is something that looks like a user name but isn't. For example, every system is required to have an alias called "postmaster". There is no real user called postmaster. Mail sent to this alias is automatically routed to a system staff member who is responsible for handling problems having to do with mail. A mailing list is like an alias, but mail sent there is forwarded to more than one person. Actually there is no sharp line between an alias and a mailing list. They are defined in the same way. It's just a question of how they are used. You can set up your own private aliases and mailing lists by putting lines into a file called ".mailrc", which must be created in your home directory. Here is an example of a .mailrc file:

```
alias roy marantz@klinzhai
alias project ron joseph@elbereth hedrick@sumex.stanford.edu
```

This file sets up two aliases: "roy" and "project". If you send mail to roy, it goes to marantz@klinzhai. If you send it to project, it goes to the people listed. Note that your aliases have special meaning only when you are sending mail. Each user who wants to use an alias must have it in his own .mailrc file. The system administrator can set up aliases and mailing lists that work for everyone. These are refered to as "system-wide" mailing lists. (Note that aliases can be defined in .mailrc even if you use Emacs rather than Mail to send mail.)

9.1. The mail program

There are two ways to read your mail and send mail to other users. One is a special program called "mail". The other is the mail and rmail commands within Emacs. We now recommend using Emacs under most circumstances. However if you are on a slow dialup, or system response is slow, the mail program may be more convenient. This section describes the mail program. The next section describes the mail facilities within Emacs. You may want to skip to the next section.

The program used to implement computer mail is simply called "mail". The "mail" program is used in two somewhat different ways: to send mail to other users, and to read mail others have sent to you. To send a message, you type mail with the name of the user you want to send mail to, e.g. "mail smith". If you type "mail" with no argument, it will read your mail.

To send a message, you use the command "mail" with the addresses. Then you type the message, ending in ^D. For example:

```
mail smith jones@topaz.rutgers.edu peters
Have you finished your parts of our assignment yet?
I am ready to test mine.
^D
```

This message will go to 3 people, smith and peters on this computer and jones on a different computer called "topaz.rutgers.edu". The message contains 2 lines of text. Notice that you type ^D (control-D) at the beginning of a line to say that you have finished typing the message.

The next time these people log in, they will see a line saying "you have mail" as part of the greeting message. This tells them that they should run "mail" to read your message.

See the "man mail" for more details about sending mail. If you make a mistake while typing a message, it is possible to call EMACS to edit it. (Type ~e at the beginning of a line to do that.) There are also other useful things you can do while sending a message.

To read your mail, type "mail". You will now see something like this:

```
% mail
Mail version 2.18 5/19/83.  Type ? for help.
"/usr/spool/mail/hedrick": 2 messages 2 new
>N  1 hedrick  Mon Nov 19 22:25  9/223
 N  2 hedrick  Mon Nov 19 22:25  9/223
& t
Message  1:
... contents of message...
& d
& t
Message  2:
... contents of message...
& d
& t
No applicable messages
& q
```

As you can see, mail begins by showing you "header lines" for your messages. Then it prompts you with "& ", waiting for you to type commands. The most common commands are shown in the example. "t" types the current message. "d" deletes the current message. Once you have deleted a message, the next one becomes current, so a simple "t" will get the next message. When you have finished, a "q" will exit. There are other things you can do to messages, including moving them into a file or responding to them with a message of your own. See the manual page for mail.

If you don't delete a message, mail will normally move it into a separate file, called "mbox". The next time you run mail, you will see only new messages. The ones that you saw last time will no longer be in your "in basket". They will now be in the file mbox. If you want to leave a message in your in basket, you can use the command "hold". Note that only messages you actually looked at will be moved from the in basket to mbox. If you run mail and only look at a few of your messages, the ones you didn't look at will still be there the next time you run mail. The ones you looked at will have been moved to mbox (unless you deleted them). We recommend that you delete messages after typing them, unless you really need to look at them again. Messages take up space, and will eventually cause you to go over your disk quota.

If you want to review the messages in mbox, instead of messages in your in basket, you can run mail as follows:

```
mail -f mbox
```

There are a couple of options that we think most users will want to set up for the mail program. To set up an option, create a file called ".mailrc". This file normally consists of commands to mail that set up various options. Commands that we think most users will want are

```
set ask
set crt=24
```

The first causes mail to prompt you for a subject line. It's considered good form to supply subjects with your message, and it is much easier to let mail ask you for one than to supply it yourself. The set crt command says that if a message is longer than 24 lines, it should be displayed using "more". If you don't do this, mail will have a tendency to scroll off the top of your screen.

9.2. Using Emacs to read and send mail

It is also possible to use Emacs as a mail system. We think many users will prefer the Emacs user interface. There are two Emacs commands: "m-x rmail" and "m-x mail". (M-x means "meta-X". It is typed by hitting the "escape" key and then the letter "x".) M-x rmail gets you into the mail reading system. M-x mail simply sends a single message.

Emacs mail uses a file called RMAIL. (The file name really uses upper-case letters.) This file is put in your home directory. It contains all of your mail, in a format that only Emacs understands. (This is the same format used by the mail facilities in TOPS-20 Emacs. It is called "Babyl format".) Emacs takes mail from both your Unix "in-basket", and from the file "mbox" in your directory. (Mbox is created by the Unix mail program.) It adds this mail to RMAIL automatically. Unlike the Unix "mail" program, Emacs is designed so that you can stay in it for days. Now and then you should use the command "g" to check for new mail, and the command "s" to save any changes to RMAIL.

M-x rmail puts you into a mode for reviewing incoming mail and sending messages. When you type the command "m-x rmail", you will go into a new buffer, called RMAIL. It will show a mail message. In the mode line, you will see an indication like "(RMAIL 66/67 Narrow)" This says that you are currently reading message 66, and that there are a total of 67 in your mail file. When you are in the RMAIL buffer, there are a special set of commands, designed to make it easier to read mail. To see a complete list of them, use the facility for getting help on the current mode, i.e. "c-h m". That is, type control-H and then the letter "m". (Control-H is the general-purpose help character. "m" asks for help on the current mode.) The most commonly-used commands are

```
space    scroll to next screen of this message
delete   scroll to previous screen of this message
n        next message
p        previous message
d        delete this message
u        undelete this message
e        expunge deleted messages
s        expunge, and save the message file
q        quit: expunge, save, then switch to another buffer
g        get new mail; checks for new mail and reads it in
m        mail a message
r        reply to this message
f        forward this message to another user
o        output this message to a file
```

When you use the f, r, or m commands, Emacs will create a second buffer, which you can use to compose a new message. The three commands differ by what they put in this buffer. "m" is the simplest. It will give you a blank message form. You will see

```
To:
Subject:
--text follows this line--
```

You should fill in the To: line with the address of the person you want to send mail to. To send to more than one person, list them separated by commas. Fill in a one-line description of the subject on the Subject: line. Put the actual text of the message below the line with the --'s in it. You use normal Emacs editing commands in doing this. When the message is finished, type "c-c c-c" (that is, two control-C's). That will cause Emacs to send the message.

You can add extra fields to the header, using normal Emacs commands. For example, you can add a line starting with "cc: " if you want to send copies of the message to someone. There is little difference between listing someone in the To: list and the Cc: list. The message goes to both. The distinction is up to you. Normally it is used to show that you intend one person to handle the matter, but want another one to see it. You can also add a line beginning with "bcc: ". This is for "blind copies". The message will be sent to these people also, but the to: and cc: people will not see a list of the bcc: people.

The commands "f" and "r" are similar to "m", but they fill in some fields for you. "r" is used to reply to a message. It sets up the "to: " and "cc: " list so that your message goes back to the person who sent you this message, with cc's to everyone else who got a copy. You should look carefully at the list of people. If you don't want the reply to go to all of them, then use normal Emacs editing commands to remove the unwanted people. "f" is used to forward a message. It puts a copy of the message into the body of the new message. Otherwise, it behaves very much like the "m" command.

The Emacs command "m-x mail" is used to mail a message when you are not already in rmail. It acts much like the "m" command inside "rmail". That is, it creates a buffer with To:, Subject:, and a place for you to type the message. Type "c-c c-c" (two control-C's) to send the message.

If you normally stay in Emacs a lot, the commands m-x rmail and m-x mail will be fine for you. But some people find it convenient to define shell commands that start up emacs and put them into rmail or mail, all in one command. To do that, put in your .cshrc file the commands

```
alias rmail emacs -f rmail
alias mail  emacs -f mail
```

Then rmail will start Emacs and go into rmail, and mail will start emacs and go into mail. If you want to be able to use normal Unix mail, you may prefer to use the command name smail instead of mail.

10. News

The Unix news system is used to distribute information to large groups of people. There are international newsgroups that distribute technical information about Unix and other computer-related topics, as well as hobbyist and cultural information of various sorts. Newsgroups are created for most computer science courses, to allow faculty to post notices for students. It is possible to set up newsgroups for research projects and other groups. (To do this, see your system administrator.)

The news readers are conceptually very simple. They keep track of what messages you have seen, and show you all of the messages you haven't seen yet, one by one. You can select which newsgroups you are interested in scanning. Most users do this by exclusion. That is, they start out looking at every newsgroup, and then turn off groups that they don't want to see. This means that if a new group is created, you see messages from it, and can disable it if you decide you aren't interested. (Use the "ug" command to "unsubscribe" from a group you aren't interested in seeing.) To operate in this mode, create a file in your home directory called ".newsrc". In this file, put one line, containing "options -n all". If you want to see a specific set of groups, you can list them instead of saying "all".

There are several different news readers. The systems staff all use "vnews", so that is what we recommend. However tastes are strong in this area, and some users would rather die than switch from "rn" or "readnews". Vnews is likely to be maintained the best. Normally, you invoke vnews without any arguments. It scans new news, and presents it to you, one screen at a time. You hit the space bar to go to the next screen. There are a number of commands to vnews. You can see them by typing "?" when you are in it. The most common are

```
space    go to next screen
n        go to next article
b        go to previous article
^N       go forward one line
^P       go back one line
r        reply to this article (send mail to author)
f        post followup (post article to the whole world)
ug       unsubscribe to this newsgroup (no longer look at
           articles in this newsgroup)
q        quit
```

Note the difference between the "r" and "f" commands. "r" sends a reply to the person who wrote the posting. "f" sends a response that goes to the entire world as a normal posting. Normally you should use "r" to answer questions and comment on a posting. When someone asks a question, generally he will collect all the answers, and then send a summary to the net. "f" should be used only if you are sure everyone will be interested in the answer. See the comments below on posting, where the cost of making a posting is discussed.

One special feature of vnews is the article index. In this mode, you see one line for each new article in the current newsgroup. The n and b commands move the cursor up and down one line. To look at the current article, hit CR. To get back to the article index, use the "a" command. If you always want to go to the index when you move to a new newsgroup, use the -A option. The easiest way to do this is to put the following line as the first line in .newsrc: "options -A -n all".

If you do not create a .newsrc file, you will see only a small set of "mandatory" groups. These are groups where announcements of a general nature are put. This would be appropriate for someone who wanted to know about important software changes, but not to follow the general news. Currently, this includes the following groups: general, announce, and all.announce. See below for the significance of these groups.

There are too many groups to list them all here. However here are some general guidelines. Groups with no dots in them are local to the particular machine. Other groups have prefixes that indicate their nature. Currently most machines have local groups called general and announce. Announce is for official announcements, such as software changes. It is moderated, so that people can’t put things on it without permission. General is more of a general bulletin board, for user postings which the author believes should be seen by everyone on the computer. There are University-wide equivalents of these groups, called ru.announce and ru.general. They are intended for software and policy announcements that are likely to involve all systems (ru.announce) and University-wide announcements by users (ru.general).

Here are the other major prefixes:

```
    qa          - questions and answers from users.  This is the
                  way to communicate with the systems staff.
    caip, dcs   - items of interest to specific departments
                  Newgroups for classes are under dcs
    comp        - technical newsgroups
    misc        - general-interest groups that are hard to
                  classify
    mod         - moderated newsgroups [obsolete]
    na          - groups forwarded from the Arpanet [obsolete]
    net         - [obsolete]
    nj, ny      - cultural events in these areas
    ru, princeton  - University-wide announcements
    rec         - recreational or hobbyist
    sci         - non-computer scientific groups [generally
                  at a hobbyist level -- serious research work
                  is not published here]
    soc         - social interactions
    talk        - religion, politics, etc
```

For a complete list of groups, see the file /usr/lib/news/newsgroups.

To post a message to a newsgroup, use the program "postnews". It is generally self-explanatory. Except for groups with obvious limitations, e.g. ru or dcs, postings go around the country. Much of the distribution is done by long-distance phone calls. Thus each message costs money, probably on the order of $100. While there are many people on the network who use it as if it were free, this is very anti-social. The network is currently in the middle of a crisis, caused by too many people posting messages. We would like for Rutgers users not to contribute to this crisis. If you enjoy political discussions or other voluminous postings, please use the ru groups. These groups are restricted to Rutgers. Distribution is done by Ethernet, so they don’t cost much.

Index