#62

Some practical matters. First, the UNIX system has become very popular, and there are a number of versions in wide use. For example, the 7th Edition comes from the original source of the UNIX system, the Computing Science Research Center at Bell Labs. System III and System V are the official Bell Labs-supported versions. The University of California at Berkeley distributes systems derived from the 7th Edition, usually known as UCB 4.xBSD. In addition, there are numerous variants, particularly on small computers, that are derived from the 7th Edition.

We have tried to cope with this diversity by sticking closely to those aspects that are likely to be the same everywhere. Although the lessons that we want to teach are independent of any particular version, for specific details we have chosen to present things as they were in the 7th Edition, since it forms the basis of most of the UNIX systems in widespread use. We have also run the examples on Bell Labs' System V and on Berkeley 4.1BSD; only trivial changes were required, and only in a few examples. Regardless of the version your machine runs, the differences you find should be minor.

Second, although there is a lot of material in this book, it is not a reference manual. We feel it is more important to teach an approach and a style of use than just details. The *UNIX Programmer's Manual* is the standard source of information. You will need it to resolve points that we did not cover, or to determine how your system differs from others.

Third, we believe that the best way to learn something is by doing it. This book should be read at a terminal, so that you can experiment, verify or contradict what we say, explore the limits and the variations. Read a bit, try it out, then come back and read some more.

We believe that the UNIX system, though certainly not perfect, is a marvelous computing environment. We hope that reading this book will help you to reach that conclusion too.

We are grateful to many people for constructive comments and criticisms, and for their help in improving our code. In particular, Jon Bentley, John Linderman, Doug McIlroy, and Peter Weinberger read multiple drafts with great care. We are indebted to Al Aho, Ed Bradford, Bob Flandrena, Dave Hanson, Ron Hardin, Marion Harris, Gerard Holzmann, Steve Johnson, Nico Lomuto, Bob Martin, Larry Rosler, Chris Van Wyk, and Jim Weythman for their comments on the first draft. We also thank Mike Bianchi, Elizabeth Bimmler, Joe Carfagno, Don Carter, Tom De Marco, Tom Duff, David Gay, Steve Mahaney, Ron Pinter, Dennis Ritchie, Ed Sitar, Ken Thompson, Mike Tilson, Paul Tukey, and Larry Wehr for valuable suggestions.

Brian Kernighan

Rob Pike

CHAPTER 1: UNIX FOR BEGINNERS

What is "UNIX"? In the narrowest sense, it is a time-sharing operating system *kernel*: a program that controls the resources of a computer and allocates them among its users. It lets users run their programs; it controls the peripheral devices (discs, terminals, printers, and the like) connected to the machine; and it provides a file system that manages the long-term storage of information such as programs, data, and documents.

In a broader sense, "UNIX" is often taken to include not only the kernel, but also essential programs like compilers, editors, command languages, programs for copying and printing files, and so on.

Still more broadly, "UNIX" may even include programs developed by you or other users to be run on your system, such as tools for document preparation, routines for statistical analysis, and graphics packages.

Which of these uses of the name "UNIX" is correct depends on which level of the system you are considering. When we use "UNIX" in the rest of this book, the context should indicate which meaning is implied.

The UNIX system sometimes looks more difficult than it is — it's hard for a newcomer to know how to make the best use of the facilities available. But fortunately it's not hard to get started — knowledge of only a few programs should get you off the ground. This chapter is meant to help you to start using the system as quickly as possible. It's an overview, not a manual; we'll cover most of the material again in more detail in later chapters. We'll talk about these major areas:

- basics — logging in and out, simple commands, correcting typing mistakes, mail, inter-terminal communication.
- day-to-day use — files and the file system, printing files, directories, commonly-used commands.
- the command interpreter or *shell* — filename shorthands, redirecting input and output, pipes, setting erase and kill characters, and defining your own search path for commands.

If you've used a UNIX system before, most of this chapter should be familiar; you might want to skip straight to Chapter 2.

some work too. The rest of this section will discuss the session above, plus other programs that make it possible to do useful things.

Logging in

You must have a login name and password, which you can get from your system administrator. The UNIX system is capable of dealing with a wide variety of terminals, but it is strongly oriented towards devices with *lower case*: case distinctions matter! If your terminal produces only upper case (like some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure the switches are set appropriately on your device: upper and lower case, full duplex, and any other settings that local experts advise, such as the speed, or *baud rate*. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone or merely flipping a switch. In either case, the system should type

```
login:
```

If it types garbage, you may be at the wrong speed; check the speed setting and other switches. If that fails, press the BREAK or INTERRUPT key a few times, slowly. If nothing produces a login message, you will have to get help.

When you get the `login:` message, type your login name *in lower case*. Follow it by pressing RETURN. If a password is required, you will be asked for it, and printing will be turned off while you type it.

The culmination of your login efforts is a *prompt*, usually a single character, indicating that the system is ready to accept commands from you. The prompt is most likely to be a dollar sign `$` or a percent sign `%`, but you can change it to anything you like; we'll show you how a little later. The prompt is actually printed by a program called the *command interpreter* or *shell*, which is your main interface to the system.

There may be a message of the day just before the prompt, or a notification that you have mail. You may also be asked what kind of terminal you are using; your answer helps the system to use any special properties the terminal might have.

Typing commands

Once you receive the prompt, you can type *commands*, which are requests that the system do something. We will use *program* as a synonym for command. When you see the prompt (let's assume it's `$`), type `date` and press RETURN. The system should reply with the date and time, then print another prompt, so the whole transaction will look like this on your terminal:

```
$ date
Mon Sep 26 12:20:57 EDT 1983
$
```

Don't forget RETURN, and don't type the `$`. If you think you're being

ignored, press RETURN; something should happen. RETURN won't be mentioned again, but you need it at the end of every line.

The next command to try is `who`, which tells you everyone who is currently logged in:

```
$ who
rlm      tty0    Sep 26 11:17
pjw      tty4    Sep 26 11:30
gerard   tty7    Sep 26 10:27
mark     tty9    Sep 26 07:59
you      ttya    Sep 26 12:20
$
```

The first column is the user name. The second is the system's name for the connection being used ("tty" stands for "teletype," an archaic synonym for "terminal"). The rest tells when the user logged on. You might also try

```
$ who am i
you      ttya    Sep 26 12:20
$
```

If you make a mistake typing the name of a command, and refer to a nonexistent command, you will be told that no command of that name can be found:

```
$ whom                          Misspelled command name ...
whom: not found                   ... so system didn't know how to run it
$
```

Of course, if you inadvertently type the name of an actual command, it will run, perhaps with mysterious results.

Strange terminal behavior

Sometimes your terminal will act strangely, for example, each letter may be typed twice, or RETURN may not put the cursor at the first column of the next line. You can usually fix this by turning the terminal off and on, or by logging out and logging back in. Or you can read the description of the command `stty` ("set terminal options") in Section 1 of the manual. To get intelligent treatment of tab characters if your terminal doesn't have tabs, type the command

```
$ stty -tabs
```

and the system will convert tabs into the right number of spaces. If your terminal does have computer-settable tab stops, the command `tabs` will set them correctly for you. (You may actually have to say

```
$ tabs terminal-type
```

to make it work — see the `tabs` command description in the manual.)

hanging up the phone will stop most programs.

If you just want output to pause, for example to keep something critical from disappearing off the screen, type *ctl*-s. The output will stop almost immediately; your program is suspended until you start it again. When you want to resume, type *ctl*-q.

Logging out

The proper way to log out is to type *ctl*-d instead of a command; this tells the shell that there is no more input. (How this actually works will be explained in the next chapter.) You can usually just turn off the terminal or hang up the phone, but whether this really logs you out depends on your system.

Mail

The system provides a postal system for communicating with other users, so some day when you log in, you will see the message

```
You have mail.
```

before the first prompt. To read your mail, type

```
$ mail
```

Your mail will be printed, one message at a time, most recent first. After each item, `mail` waits for you to say what to do with it. The two basic responses are d, which deletes the message, and RETURN, which does not (so it will still be there the next time you read your mail). Other responses include p to reprint a message, s *filename* to save it in the file you named, and q to quit from `mail`. (If you don't know what a file is, think of it as a place where you can store information under a name of your choice, and retrieve it later. Files are the topic of Section 1.2 and indeed of much of this book.)

`mail` is one of those programs that is likely to differ from what we describe here; there are many variants. Look in your manual for details.

Sending mail to someone is straightforward. Suppose it is to go to the person with the login name `nico`. The easiest way is this:

```
$ mail nico
Now type in the text of the letter
on as many lines as you like...
After the last line of the letter
type a control-d:
ctl-d
$
```

The *ctl*-d signals the end of the letter by telling the `mail` command that there is no more input. If you change your mind half-way through composing the letter, press DELETE instead of *ctl*-d. The half-formed letter will be stored in a file called `dead.letter` instead of being sent.

For practice, send mail to yourself, then type `mail` to read it. (This isn't as aberrant as it might sound -- it's a handy reminder mechanism.)

There are other ways to send mail — you can send a previously prepared letter, you can mail to a number of people all at once, and you may be able to send mail to people on other machines. For more details see the description of the `mail` command in Section 1 of the *UNIX Programmer's Manual*. Henceforth we'll use the notation `mail(1)` to mean the page describing `mail` in Section 1 of the manual). All of the commands discussed in this chapter are found in Section 1.

There may also be a calendar service (see `calendar(1)`); we'll show you in Chapter 4 how to set one up if it hasn't been done already.

Writing to other users

If your UNIX system has multiple users, someday, out of the blue, your terminal will print something like

```
Message from mary tty7...
```

accompanied by a startling beep. Mary wants to write to you, but unless you take explicit action you won't be able to write back. To respond, type

```
$ write mary
```

This establishes a two-way communication path. Now the lines that Mary types on her terminal will appear on yours and vice versa, although the path is slow, rather like talking to the moon.

If you are in the middle of something, you have to get to a state where you can type a command. Normally, whatever program you are running has to stop or be stopped, but some programs, such as the editor and `write` itself, have a '!' command to escape temporarily to the shell — see Table 2 in Appendix 1.

The `write` command imposes no rules, so a protocol is needed to keep what you type from getting garbled up with what Mary types. One convention is to take turns, ending each turn with (o), which stands for "over," and to signal your intent to quit with (oo), for "over and out."

won't describe any specific screen editor here, however, partly because of typographic limitations, and partly because there is no standard one.

There is, however, an older editor called ed that is certain to be available on your system. It takes no advantage of special terminal features, so it will work on any terminal. It also forms the basis of other essential programs (including some screen editors), so it's worth learning eventually. Appendix 1 contains a concise description.

No matter what editor you prefer, you'll have to learn it well enough to be able to create files. We'll use ed here to make the discussion concrete, and to ensure that you can make our examples run on your system, but by all means use whatever editor you like best.

To use ed to create a file called junk with some text in it, do the following:

```
$ ed                          Invokes the text editor
a                             ed command to add text
now type in
whatever text you want ...
.                             Type a '.' by itself to stop adding text
w junk                        Write your text into a file called junk
39                            ed prints number of characters written
q                             Quit ed
$
```

The command a ("append") tells ed to start collecting text. The "." that signals the end of the text must be typed at the beginning of a line by itself. Don't forget it, for until it is typed, no other ed commands will be recognized — everything you type will be treated as text to be added.

The editor command w ("write") stores the information that you typed; "w junk" stores it in a file called junk. The filename can be any word you like; we picked junk to suggest that this file isn't very important.

ed responds with the number of characters it put in the file. Until the w command, nothing is stored permanently, so if you hang up and go home the information is not stored in the file. (If you hang up while editing, the data you were working on is saved in a file called ed.hup, which you can continue with at your next session.) If the system crashes (i.e., stops unexpectedly because of software or hardware failure) while you are editing, your file will contain only what the last write command placed there. But after w the information is recorded permanently; you can access it again later by typing

```
$ ed junk
```

Of course, you can edit the text you typed in, to correct spelling mistakes, change wording, rearrange paragraphs and the like. When you're done, the q command ("quit") leaves the editor.

What files are out there?

Let's create two files, junk and temp, so we know what we have:

```
$ ed
a
To be or not to be
.
w junk
19
q
$ ed
a
That is the question.
.
w temp
22
q
$
```

The character counts from ed include the character at the end of each line, called *newline*, which is how the system represents RETURN.

The ls command lists the names (not contents) of files:

```
$ ls
junk
temp
$
```

which are indeed the two files just created. (There might be others as well that you didn't create yourself.) The names are sorted into alphabetical order automatically.

ls, like most commands, has *options* that may be used to alter its default behavior. Options follow the command name on the command line, and are usually made up of an initial minus sign '-' and a single letter meant to suggest the meaning. For example, ls -t causes the files to be listed in "time" order: the order in which they were last changed, most recent first.

```
$ ls -t
temp
junk
$
```

The -l option gives a "long" listing that provides more information about each file:

```
$ ls -l
total 2
-rw-r--r-- 1 you           19 Sep 26 16:25 junk
-rw-r--r-- 1 you           22 Sep 26 16:26 temp
$
```

```
$ pr junk temp

Sep 26 16:25 1983  junk Page 1


To be or not to be
        (60 more blank lines)


Sep 26 16:26 1983  temp Page 1


That is the question.
        (60 more blank lines)
$
```

pr can also produce multi-column output:

```
$ pr -3 filenames
```

prints each file in 3-column format. You can use any reasonable number in place of "3" and pr will do its best. (The word *filenames* is a place-holder for a list of names of files.) pr -m will print a set of files in parallel columns. See pr(1).

It should be noted that pr is *not* a formatting program in the sense of rearranging lines and justifying margins. The true formatters are nroff and troff, which are discussed in Chapter 9.

There are also commands that print files on a high-speed printer. Look in your manual under names like lp and lpr, or look up "printer" in the permuted index. Which to use depends on what equipment is attached to your machine. pr and lpr are often used together; after pr formats the information properly, lpr handles the mechanics of getting it to the line printer. We will return to this a little later.

Moving, copying, removing files — mv, cp, rm

Let's look at some other commands. The first thing is to change the name of a file. Renaming a file is done by "moving" it from one name to another, like this:

```
$ mv junk precious
```

This means that the file that used to be called junk is now called precious; the contents are unchanged. If you run ls now, you will see a different list: junk is not there but precious is.


```
$ ls
precious
temp
$ cat junk
cat: can't open junk
$
```

Beware that if you move a file to another one that already exists, the target file is replaced.

To make a *copy* of a file (that is, to have two versions of something), use the cp command:

```
$ cp precious precious.save
```

makes a duplicate copy of precious in precious.save.

Finally, when you get tired of creating and moving files, the rm command removes all the files you name:

```
$ rm temp junk
rm: junk nonexistent
$
```

You will get a warning if one of the files to be removed wasn't there, but otherwise rm, like most UNIX commands, does its work silently. There is no prompting or chatter, and error messages are curt and sometimes unhelpful. Brevity can be disconcerting to newcomers, but experienced users find talkative commands annoying.

What's in a filename?

So far we have used filenames without ever saying what a legal name is, so it's time for a couple of rules. First, filenames are limited to 14 characters. Second, although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should avoid characters that might be used with other meanings. We have already seen, for example, that in the ls command, ls -t means to list in time order. So if you had a file whose name was -t, you would have a tough time listing it by name. (How would you do it?) Besides the minus sign as a first character, there are other characters with special meaning. To avoid pitfalls, you would do well to use only letters, numbers, the period and the underscore until you're familiar with the situation. (The period and the underscore are conventionally used to divide filenames into chunks, as in precious.save above.) Finally, don't forget that case distinctions matter — junk, Junk, and JUNK are three different names.

A handful of useful commands

Now that you have the rudiments of creating files, listing their names, and printing their contents, we can look at a half-dozen file-processing commands.

```
$ tail -1 poem
  and greater still, and so on.
$
```

`tail` can also be used to print a file starting at a specified line:

```
$ tail +3 filename
```

starts printing with the 3rd line. (Notice the natural inversion of the minus sign convention for arguments.)

The final pair of commands is for comparing files. Suppose that we have a variant of poem in the file new_poem:

```
$ cat poem
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
  and so ad infinitum.
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
$ cat new_poem
Great fleas have little fleas
  upon their backs to bite them,
And little fleas have lesser fleas,
  and so on ad infinitum.
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
$
```

There's not much difference between the two files; in fact you'll have to look hard to find it. This is where file comparison commands come in handy. `cmp` finds the first place where two files differ:

```
$ cmp poem new_poem
poem new_poem differ: char 58, line 2
$
```

This says that the files are different in the second line, which is true enough, but it doesn't say what the difference is, nor does it identify any differences beyond the first.

The other file comparison command is `diff`, which reports on all lines that are changed, added or deleted:


```
$ diff poem new_poem
2c2
<   upon their backs to bite 'em,
---
>   upon their backs to bite them,
4c4
<   and so ad infinitum.
---
>   and so on ad infinitum.
$
```

This says that line 2 in the first file (poem) has to be changed into line 2 of the second file (new_poem), and similarly for line 4.

Generally speaking, `cmp` is used when you want to be sure that two files really have the same contents. It's fast and it works on any kind of file, not just text. `diff` is used when the files are expected to be somewhat different, and you want to know exactly which lines differ. `diff` works only on files of text.

A summary of file system commands

Table 1.1 is a brief summary of the commands we've seen so far that deal with files.

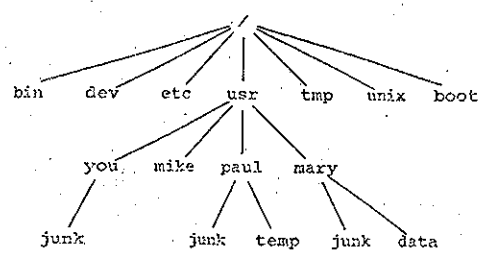
1.3 More about files: directories

The system distinguishes your file called junk from anyone else's of the same name. The distinction is made by grouping files into *directories*, rather in the way that books are placed on shelves in a library, so files in different directories can have the same name without any conflict.

Generally each user has a personal or *home directory*, sometimes called login directory, that contains only the files that belong to him or her. When you log in, you are "in" your home directory. You may change the directory you are working in — often called your working or *current directory* — but your home directory is always the same. Unless you take special action, when you create a new file it is made in your current directory. Since this is initially your home directory, the file is unrelated to a file of the same name that might exist in someone else's directory.

A directory can contain other directories as well as ordinary files ("Great directories have lesser directories ..."). The natural way to picture this organization is as a tree of directories and files. It is possible to move around within this tree, and to find any file in the system by starting at the root of the tree and moving along the proper branches. Conversely, you can start where you are and move toward the root.

Let's try the latter first. Our basic tool is the command `pwd` ("print working directory"), which prints the name of the directory you are currently in:



Your file named junk is unrelated to Paul's or to Mary's.

Pathnames aren't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, they become handy indeed. For example, your friends can print your junk by saying

```
$ cat /usr/you/junk
```

Similarly, you can find out what files Mary has by saying

```
$ ls /usr/mary
data
junk
$
```

or make your own copy of one of her files by

```
$ cp /usr/mary/data data
```

or edit her file:

```
$ ed /usr/mary/data
```

If Mary doesn't want you poking around in her files, or vice versa, privacy can be arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which can be used to control access. (Recall `ls -l`.) In our local systems, most users most of the time find openness of more benefit than privacy, but policy may be different on your system, so we'll get back to this in Chapter 2.

As a final set of experiments with pathnames, try

```
$ ls /bin /usr/bin
```

Do some of the names look familiar? When you run a command by typing its name after the prompt, the system looks for a file of that name. It normally looks first in your current directory (where it probably doesn't find it), then in `/bin`, and finally in `/usr/bin`. There is nothing special about commands

like `cat` or `ls`, except that they have been collected into a couple of directories to be easy to find and administer. To verify this, try to execute some of these programs by using their full pathnames:

```
$ /bin/date
Mon Sep 26 23:29:32 EDT 1983
$ /bin/who
srm      tty1    Sep 26 22:20
cvw      tty4    Sep 26 22:40
you      tty5    Sep 26 23:04
$
```

Exercise 1-3. Try

```
$ ls /usr/games
```

and do whatever comes naturally. Things might be more fun outside of normal working hours. □

Changing directory — `cd`

If you work regularly with Mary on information in her directory, you can say "I want to work on Mary's files instead of my own." This is done by changing your current directory with the `cd` command:

```
$ cd /usr/mary
```

Now when you use a filename (without /'s) as an argument to `cat` or `pr`, it refers to the file in Mary's directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, if you want to write a book, you might want to keep all the text in a directory called `book`. The command `mkdir` makes a new directory.

```
$ mkdir book                    Make a directory
$ cd book                       Go to it
$ pwd                           Make sure you're in the right place
/usr/you/book
...                             Write the book (several minutes pass)
$ cd ..                         Move up one level in file system
$ pwd
/usr/you
$
```

'..' refers to the parent of whatever directory you are currently in, the directory one level closer to the root. '.' is a synonym for the current directory.

```
$ cd                            Return to home directory
```

```
$ rm *
```

removes *all files* in your current directory. (You had better be *very* sure that's what you wanted to say!)

The * is not limited to the last position in a filename — *'s can be anywhere and can occur several times. Thus

```
$ rm *.save
```

removes all files that end with `.save`.

Notice that the filenames are sorted alphabetically, which is not the same as numerically. If your book has ten chapters, the order might not be what you intended, since `ch10` comes before `ch2`:

```
$ echo *
ch1.1 ch1.2 ... ch10.1 ch10.2 ... ch2.1 ch2.2 ...
$
```

The * is not the only pattern-matching feature provided by the shell, although it's by far the most frequently used. The pattern [...] matches any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated:

```
$ pr ch[12346789]*   Print chapters 1,2,3,4,6,7,8,9 but not 5
$ pr ch[1-46-9]*     Same thing
$ rm temp[a-z]       Remove any of tempa, ..., tempz that exist
```

The ? pattern matches any single character:

```
$ ls ?               List files with single-character names
$ ls -l ch?.1        List ch1.1 ch2.1 ch3.1, etc. but not ch10.1
$ rm temp?           Remove files temp1, ..., tempa, etc.
```

Note that the patterns match only *existing* filenames. In particular, you cannot make up new filenames by using patterns. For example, if you want to expand `ch` to `chapter` in each filename, you cannot do it this way:

```
$ mv ch.* chapter.*      Doesn't work!
```

because `chapter.*` matches no existing filenames.

Pattern characters like * can be used in pathnames as well as simple filenames; the match is done for each component of the path that contains a special character. Thus `/usr/mary/*` performs the match in `/usr/mary`, and `/usr/*/calendar` generates a list of pathnames of all user `calendar` files.

If you should ever have to turn off the special meaning of *, ?, etc., enclose the entire argument in single quotes, as in

```
$ ls '?'
```

You can also precede a special character with a backslash:


```
$ ls \?
```

(Remember that because ? is not the erase or line kill character, this backslash is interpreted by the shell, not by the kernel.) Quoting is treated at length in Chapter 3.

Exercise 1-4. What are the differences among these commands?

```
$ ls junk          $ echo junk
$ ls /             $ echo /
$ ls               $ echo
$ ls *             $ echo *
$ ls '*'           $ echo '*'
```

□

Input-output redirection

Most of the commands we have seen so far produce output on the terminal: some, like the editor, also take their input from the terminal. It is nearly universal that the terminal can be replaced by a file for either or both of input and output. As one example,

```
$ ls
```

makes a list of filenames on your terminal. But if you say

```
$ ls >filelist
```

that same list of filenames will be placed in the file `filelist` instead. The symbol > means "put the output in the following file, rather than on the terminal." The file will be created if it doesn't already exist, or the previous contents overwritten if it does. Nothing is produced on your terminal. As another example, you can combine several files into one by capturing the output of `cat` in a file:

```
$ cat f1 f2 f3 >temp
```

The symbol >> operates much as > does, except that it means "add to the end of." That is,

```
$ cat f1 f2 f3 >>temp
```

copies the contents of `f1`, `f2` and `f3` onto the end of whatever is already in `temp`, instead of overwriting the existing contents. As with >, if `temp` doesn't exist, it will be created initially empty for you.

In a similar way, the symbol < means to take the input for a program from the following file, instead of from the terminal. Thus, you can prepare a letter in file `let`, then send it to several people with

```
$ mail mary joe tom bob <let
```

In all of these examples, blanks are optional on either side of > or <, but our formatting is traditional.

```
$ ls | pr -3 | lpr
```

creates a 3-column list of filenames on the line printer, and

```
$ who | grep mary | wc -l
```

counts how many times Mary is logged in.

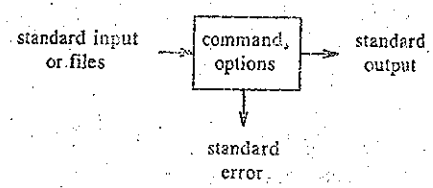
The programs in a pipeline actually run at the same time, not one after another. This means that the programs in a pipeline can be interactive; the kernel looks after whatever scheduling and synchronization is needed to make it all work.

As you probably suspect by now, the shell arranges things when you ask for a pipe; the individual programs are oblivious to the redirection. Of course, programs have to operate sensibly if they are to be combined this way. Most commands follow a common design, so they will fit properly into pipelines at any position. Normally a command invocation looks like

command optional-arguments optional-filenames

If no filenames are given, the command reads its standard input, which is by default the terminal (handy for experimenting) but which can be redirected to come from a file or a pipe. At the same time, on the output side, most commands write their output on the *standard output*, which is by default sent to the terminal. But it too can be redirected to a file or a pipe.

Error messages from commands have to be handled differently, however, or they might disappear into a file or down a pipe. So each command has a *standard error* output as well, which is normally directed to your terminal. Or, as a picture:



Almost all of the commands we have talked about so far fit this model; the only exceptions are commands like `date` and `who` that read no input, and a few like `cmp` and `diff` that have a fixed number of file inputs. (But look at the '-' option on these.)

Exercise 1-7. Explain the difference between

```
$ who | sort
```

and


```
$ who >sort
```

□

Processes

The shell does quite a few things besides setting up pipes. Let us turn briefly to the basics of running more than one program at a time, since we have already seen a bit of that with pipes. For example, you can run two programs with one command line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands:

```
$ date; who
Tue Sep 27 01:03:17 EDT 1983
kenl        tty0    Sep 27 00:43
dmr         tty1    Sep 26 23:45
rob         tty2    Sep 26 23:59
bwk         pty3    Sep 27 00:06
jj          tty4    Sep 26 23:31
you         tty5    Sep 26 23:04
ber         tty7    Sep 26 23:34
$
```

Both commands are executed (in sequence) before the shell returns with a prompt character.

You can also have more than one program running simultaneously if you wish. For example, suppose you want to do something time-consuming like counting the words in your book, but you don't want to wait for wc to finish before you start something else. Then you can say

```
$ wc ch* >wc.out &
6944                        Process-id printed by the shell
$
```

The ampersand & at the end of a command line says to the shell "start this command running, then take further commands from the terminal immediately," that is, don't wait for it to complete. Thus the command will begin, but you can do something else while it's running. Directing the output into the file `wc.out` keeps it from interfering with whatever you're doing at the same time.

An instance of a running program is called a *process*. The number printed by the shell for a command initiated with & is called the *process-id*; you can use it in other commands to refer to a specific running program.

It's important to distinguish between programs and processes. `wc` is a program; each time you run the program `wc`, that creates a new process. If several instances of the same program are running at the same time, each is a separate process with a different process-id.

If a pipeline is initiated with &, as in

a bother to have to type this every time you log in.

The shell comes to the rescue. If there is a file named `.profile` in your login directory, the shell will execute the commands in it when you log in, before printing the first prompt. So you can put commands into `.profile` to set up your environment as you like it, and they will be executed every time you log in.

The first thing most people put in their `.profile` is

```
stty erase ←
```

We're using ← here so you can see it, but you could put a literal backspace in your `.profile`. `stty` also understands the notation `^x` for *ctl-x*, so you can get the same effect with

```
stty erase '^h'
```

because ctl-h is backspace. (The `^` character is an obsolete synonym for the pipe operator `|`, so you must protect it with quotes.)

If your terminal doesn't have sensible tab stops, you can add `-tabs` to the `stty` line:

```
stty erase '^h' -tabs
```

If you like to see how busy the system is when you log in, add

```
who | wc -l
```

to count the users. If there's a news service, you can add `news`. Some people like a fortune cookie:

```
/usr/games/fortune
```

After a while you may decide that it is taking too long to log in, and cut your `.profile` back to the bare necessities.

Some of the properties of the shell are actually controlled by so-called *shell variables*, with values that you can access and set yourself. For example, the prompt string, which we have been showing as `$`, is actually stored in a shell variable called `PS1`, and you can set it to anything you like, like this:

```
PS1='Yes dear? '
```

The quotes are necessary since there are spaces in the prompt string. Spaces are not permitted around the `=` in this construction.

The shell also treats the variables `HOME` and `MAIL` specially. `HOME` is the name of your home directory; it is normally set properly without having to be in `.profile`. The variable `MAIL` names the standard file where your mail is kept. If you define it for the shell, you will be notified after each command if new mail has arrived:†

† This is implemented badly in the shell. Looking at the file after every command adds perceptibly to the system load. Also, if you are working in an editor for a long time you won't learn about new mail because you aren't running new commands with your login shell. A better design is to look every few minutes, instead of after every command. Chapters 5 and 7 show how to implement this kind of mail checker. A third possibility, not available to everyone, is to have the `mail` program notify you itself: it certainly knows when mail comes for you.


```
MAIL=/usr/spool/mail/you
```

(The mail file may be different on your system; `/usr/mail/you` is also common.)

Probably the most useful shell variable is the one that controls where the shell looks for commands. Recall that when you type the name of a command, the shell normally looks for it first in the current directory, then in `/bin`, and then in `/usr/bin`. This sequence of directories is called the *search path*, and is stored in a shell variable called `PATH`. If the default search path isn't what you want, you can change it, again usually in your `.profile`. For example, this line sets the path to the standard one plus `/usr/games`:

```
PATH=.:/bin:/usr/bin:/usr/games          One way ...
```

The syntax is a bit strange: a sequence of directory names separated by colons. Remember that `.` is the current directory. You can omit the `.`; a null component in `PATH` means the current directory.

An alternate way to set `PATH` in this specific case is simply to augment the previous value:

```
PATH=$PATH:/usr/games                    ... Another way
```

You can obtain the value of any shell variable by prefixing its name with a `$`. In the example above, the expression `$PATH` retrieves the current value, to which the new part is added, and the result is assigned back to `PATH`. You can verify this with `echo`:

```
$ echo PATH is $PATH
PATH is :/bin:/usr/bin:/usr/games
$ echo $HOME              Your login directory
/usr/you
$
```

If you have some of your own commands, you might want to collect them in a directory of your own and add that to your search path as well. In that case, your `PATH` might look like this:

```
PATH=:$HOME/bin:/bin:/usr/bin:/usr/games
```

We'll talk about writing your own commands in Chapter 3.

Another variable, often used by text editors fancier than `ed`, is `TERM`, which names the kind of terminal you are using. That information may make it possible for programs to manage your screen more effectively. Thus you might add something like