UNIX For Beginners — Second Edition

Brian W. Kernighan

(Updated for 4.3BSD by Mark Seiden)

ABSTRACT

This paper is meant to help new users get started on the UNIX† operating system. It includes:

- basics needed for day-to-day use of the system — typing commands, correcting typing mistakes, logging in and out, mail, inter-terminal communication, the file system, printing files, redirecting I/O, pipes, and the shell.
- document preparation — a brief discussion of the major formatting programs and macro packages, hints on preparing documents, and capsule descriptions of some supporting software.
- UNIX programming — using the editor, programming the shell, programming in C, other languages and tools.
- An annotated UNIX bibliography.

INTRODUCTION

From the user's point of view, the UNIX operating system is easy to learn and use, and presents few of the usual impediments to getting the job done. It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this introduction is to help new users get used to the main ideas of the UNIX system and start making effective use of it quickly.

You should have a couple of other documents with you for easy reference as you read this one. The most important is *The UNIX Programmer's Manual*; it's often easier to tell you to read about something in the manual than to repeat its contents here. The other useful document is *A Tutorial Introduction to the UNIX Text Editor*, which will tell you how to use the editor to get text — programs, data, documents — into the computer.

A word of warning: the UNIX system has become quite popular, and there are several major variants in widespread use. Of course details also change with time. So although the basic structure of UNIX and how to use it is common to all versions, there will certainly be a few things which are different on your system from what is described here. We have tried to minimize the problem, but be aware of it. In cases of doubt, this paper describes Version 7 UNIX.

This paper has five sections:

1. Getting Started: How to log in, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which system you log into (phone numbers, for example) and what terminal you use, so this section must necessarily be supplemented by local information.
2. Day-to-day Use: Things you need every day to use the system effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX systems. This section contains advice, but not extensive instructions on any of the formatting tools.
4. Writing Programs: UNIX is an excellent system for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages provided by the system.
5. A UNIX Reading List. An annotated bibliography of documents that new users should be aware of.

---

† UNIX is a trademark of Bell Laboratories.

I. GETTING STARTED

Logging In

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number, unless your system uses permanently connected terminals. The UNIX system is capable of dealing with a wide variety of terminals: Terminet 300's; Execuport, TI and similar portables; video (CRT) terminals like the HP2640, etc.; high-priced graphics terminals like the Tektronix 4014; plotting terminals like those from GSI and DASI; and even the venerable Teletype in its various forms. But note: UNIX is strongly oriented towards devices with *lower case.* If your terminal produces only upper case (e.g., model 33 Teletype, some video and portable terminals), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device. Switches that might need to be adjusted include the speed, upper/lower case mode, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal; this may involve dialing a telephone call or merely flipping a switch. In either case, UNIX should type ''**login:**'' at you. If it types garbage, you may be at the wrong speed; check the switches. If that fails, push the ''break'' or ''interrupt'' key a few times, slowly. If that fails to produce a login message, consult a guru.

When you get a **login:** message, type your login name *in lower case.* Follow it by a RETURN; the system will not do anything until you type a RETURN. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it. Don't forget RETURN.

The culmination of your login efforts is a ''prompt character,'' a single character that indicates that the system is ready to accept commands from you. The prompt character is usually a dollar sign **$** or a percent sign **%**. (You may also get a message of the day just before the prompt character, or a notification that you have mail.)

Typing Commands

Once you've seen the prompt character, you can type commands, which are requests that the system do something. Try typing

date

followed by RETURN. You should get back something like

Mon Jan 16 14:17:10 EST 1978

Don't forget the RETURN after the command, or nothing will happen. If you think you're being ignored, type a RETURN; something should happen. RETURN won't be mentioned again, but don't forget it — it has to be there at the end of each line.

Another command you might try is **who**, which tells you everyone who is currently logged in:

who

gives something like

mb	**tty01**	**Jan 16**	**09:11**
ski	**tty05**	**Jan 16**	**09:33**
gam	**tty11**	**Jan 16**	**13:07**

The time is when the user logged in; ''ttyxx'' is the system's idea of what terminal the user is on.

If you make a mistake typing the command name, and refer to a non-existent command, you will be told. For example, if you type

whom

you will be told

whom: not found

Of course, if you inadvertently type the name of some other command, it will run, with more or less mysterious results.

Strange Terminal Behavior

Sometimes you can get into a state where your terminal acts strangely. For example, each letter may be typed twice, or the RETURN may not cause a line feed or a return to the left margin. You can often fix this by logging out and logging back in.†
Or you can read the description of the command **stty** in section 1 of the manual. To get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs, type the command

stty –tabs

and the system will convert each tab into the right number of blanks for you. If your terminal does have computer-settable tabs, the command **tabs** will set the stops correctly for you.

Mistakes in Typing

If you make a typing mistake, and see it before RETURN has been typed, there are two ways to recover. The sharp-character **#** erases the last character typed; in fact successive uses of **#** erase characters back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

† In Berkeley Unix, the command "reset<control-j>" will often reset a terminal apparently in a strange state because a fullscreen editor crashed.

dd#atte##e

is the same as **date**.‡

The at-sign **@** erases all of the characters typed so far on the current input line, so if the line is irretrievably fouled up, type an **@** and start the line over.

What if you must enter a sharp or at-sign as part of the text? If you precede either **#** or **@** by a backslash \, it loses its erase meaning. So to enter a sharp or at-sign in something, type \# or \@. The system will always echo a newline at you after your at-sign, even if preceded by a backslash. Don't worry — the at-sign has been recorded.

To erase a backslash, you have to type two sharps or two at-signs, as in \##. The backslash is used extensively in UNIX to indicate that the following character is in some way special.

Read-ahead

UNIX has full read-ahead, which means that you can type as fast as you want, whenever you want, even when some command is typing at you. If you type during output, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So you can type several commands one after another without waiting for the first to finish or even begin.

Stopping a Program

You can stop most programs by typing the character ''DEL'' (perhaps called ''delete'' or ''rubout'' on your terminal). The ''interrupt'' or ''break'' key found on most terminals can also be used.† In a few programs, like the text editor, DEL stops whatever the program is doing but leaves you in that program. Hanging up the phone will stop most programs.‡

Logging Out

The easiest way to log out is to hang up the phone. You can also type

login

and let someone else use the terminal you were on.* It is usually not sufficient just to turn off the terminal. Most UNIX systems do not use a time-out mechanism, so you'll be there forever unless you hang up.

‡ Many installations set the erase character for display terminals to the delete or backspace key. "stty all" tells you what it actually is.

† In Berkeley Unix, "control-c" is the usual way to stop programs. "stty all" tells you the value of your "intr" key.

‡ If you use the c shell, programs running in the background continue running even if you hang up.

* "control-d" and "logout" are other alternatives.

Mail

When you log in, you may sometimes get the message

You have mail.

UNIX provides a postal system so you can communicate with other users of the system. To read your mail, type the command

mail

Your mail will be printed, one message at a time, most recent message first.‡ After each message, **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 mailbox). Other responses are described in the manual. (Earlier versions of **mail** do not process one message at a time, but are otherwise similar.)

How do you send mail to someone else? Suppose it is to go to ''joe'' (assuming ''joe'' is someone's login name). The easiest way is this:

mail joe
now type in the text of the letter
on as many lines as you like ...
After the last line of the letter
type the character ''control–d'',
that is, hold down ''control'' and type
a letter ''d''.

And that's it. The ''control-d'' sequence, often called ''EOF'' for end-of-file, is used throughout the system to mark the end of input from a terminal, so you might as well get used to it.

For practice, send mail to yourself. (This isn't as strange as it might sound — mail to oneself is a handy reminder mechanism.)

There are other ways to send mail — you can send a previously prepared letter, and you can mail to a number of people all at once. For more details see **mail**(1). (The notation **mail**(1) means the command **mail** in section 1 of the *UNIX Programmer's Manual*.)

Writing to other users†

At some point, out of the blue will come a message like

Message from joe tty07...

accompanied by a startling beep. It means that Joe wants to talk to you, but unless you take explicit action you won't be able to talk back. To respond, type the

‡ The Berkeley mail program lists the headers of some number of unread pieces of mail in the order of their receipt.

† Although "write" works on Berkeley UNIX, there is a much nicer way of communicating using display-terminals — "talk" splits the screen into two sections, and both of you can type simultaneously (see talk(1)).

command

write joe

This establishes a two-way communication path. Now whatever Joe types on his terminal will appear on yours and vice versa. 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 terminate or be terminated. If you're editing, you can escape temporarily from the editor — read the editor tutorial.)

A protocol is needed to keep what you type from getting garbled up with what Joe types. Typically it's like this:

> Joe types **write smith** and waits.
> Smith types **write joe** and waits.
> Joe now types his message (as many lines as he likes). When he's ready for a reply, he signals it by typing **(o)**, which stands for ''over''.
> Now Smith types a reply, also terminated by **(o)**.
> This cycle repeats until someone gets tired; he then signals his intent to quit with **(oo)**, for ''over and out''.
> To terminate the conversation, each side must type a ''control-d'' character alone on a line. (''Delete'' also works.) When the other person types his ''control-d'', you will get the message **EOF** on your terminal.

If you write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, simply type ''control-d''.

On-line Manual

The *UNIX Programmer's Manual* is typically kept on-line. If you get stuck on something, and can't find an expert to assist you, you can print on your terminal some manual section that might help. This is also useful for getting the most up-to-date information on a command. To print a manual section, type ''man command-name''. Thus to read up on the **who** command, type

man who

and, of course,

man man

tells all about the **man** command.

Computer Aided Instruction

Your UNIX system may have available a program called **learn**, which provides computer aided instruction on the file system and basic commands, the editor, document preparation, and even C programming. Try typing the command

learn

If **learn** exists on your system, it will tell you what to do from there.

II. DAY-TO-DAY USE

Creating Files — The Editor

If you have to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with the UNIX ''text editor'' **ed**. Since **ed** is thoroughly documented in **ed**(1) and explained in *A Tutorial Introduction to the UNIX Text Editor*, we won't spend any time here describing how to use it. All we want it for right now is to make some *files*. (A file is just a collection of information stored in the machine, a simplistic but adequate definition.)

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

ed junk	(invokes the text editor)
a	(command to ''ed'', to add text)
now type in	
whatever text you want ...	
.	(signals the end of adding text)

The ''**.**'' that signals the end of adding text must be 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.

At this point you can do various editing operations on the text you typed in, such as correcting spelling mistakes, rearranging paragraphs and the like. Finally, you must write the information you have typed into a file with the editor command **w**:

w

ed will respond with the number of characters it wrote into the file **junk**.

Until the **w** command, nothing is stored permanently, so if you hang up and go home the information is lost.† But after **w** the information is there permanently; you can re-access it any time by typing

ed junk

Type a **q** command to quit the editor. (If you try to quit without writing, **ed** will print a **?** to remind you. A second **q** gets you out regardless.)

Now create a second file called **temp** in the same manner. You should now have two files, **junk** and **temp**.

---

† This is not strictly true — 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.

What files are out there?

The **ls** (for ‘‘list’’) command lists the names (not contents) of any of the files that UNIX knows about. If you type

ls

the response will be

junk
temp

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

ls –t

causes the files to be listed in the order in which they were last changed, most recent first. The **–l** option gives a ‘‘long’’ listing:

ls –l

will produce something like

–rw–rw–rw– 1 bwk 41 Jul 22 2:56 junk
–rw–rw–rw– 1 bwk 78 Jul 22 2:57 temp

The date and time are of the last change to the file. The 41 and 78 are the number of characters (which should agree with the numbers you got from **ed**). **bwk** is the owner of the file, that is, the person who created it. The **–rw–rw–rw–** tells who has permission to read and write the file, in this case everyone.

Options can be combined: **ls –lt** gives the same thing as **ls –l**, but sorted into time order. You can also name the files you’re interested in, and **ls** will list the information about them only. More details can be found in **ls**(1).

The use of optional arguments that begin with a minus sign, like **–t** and **–lt**, is a common convention for UNIX programs. In general, if a program accepts such optional arguments, they precede any filename arguments. It is also vital that you separate the various arguments with spaces: **ls–l** is not the same as **ls –l**.

Printing Files

Now that you’ve got a file of text, how do you print it so people can look at it? There are a host of programs that do that, probably more than are needed.

One simple thing is to use the editor, since printing is often done just before making changes anyway. You can say

ed junk
1,$p

ed will reply with the count of the characters in **junk** and then print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it’s not feasible to use the editor for printing. For example, there is a limit on how big a file **ed** can handle (several thousand lines). Secondly, it will only print one file at a time, and sometimes you want to print several, one after another. So here are a couple of alternatives.

First is **cat**, the simplest of all the printing programs. **cat** simply prints on the terminal the contents of all the files named in a list. Thus

cat junk

prints one file, and

cat junk temp

prints two. The files are simply concatenated (hence the name ‘‘**cat**’’) onto the terminal.

pr produces formatted printouts of files. As with **cat**, **pr** prints all the files named in a list. The difference is that it produces headings with date, time, page number and file name at the top of each page, and extra lines to skip over the fold in the paper. Thus,

pr junk temp

will print **junk** neatly, then skip to the top of a new page and print **temp** neatly.

pr can also produce multi-column output:

pr –3 junk

prints **junk** in 3-column format. You can use any reasonable number in place of ‘‘3’’ and **pr** will do its best. **pr** has other capabilities as well; see **pr**(1).

It should be noted that **pr** is *not* a formatting program in the sense of shuffling lines around and justifying margins. The true formatters are **nroff** and **troff**, which we will get to in the section on document preparation.

There are also programs that print files on a high-speed printer. Look in your manual under **opr** and **lpr**. Which to use depends on what equipment is attached to your machine.

Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving it a new name), like this:

mv junk precious

This means that what used to be ‘‘junk’’ is now ‘‘precious’’. If you do an **ls** command now, you will get

precious
temp

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

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

cp precious temp1

makes a duplicate copy of **precious** in **temp1**.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called **rm**.

rm temp temp1

will remove both of the files named.

You will get a warning message if one of the named files wasn't there, but otherwise **rm**, like most UNIX commands, does its work silently. There is no prompting or chatter, and error messages are occasionally curt. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

What's in a Filename

So far we have used filenames without ever saying what's a legal name, so it's time for a couple of rules. First, filenames are limited to 14 characters, which is enough to be descriptive.† 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 probably 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. Besides the minus sign, there are other characters which have special meaning. To avoid pitfalls, you would do well to use only letters, numbers and the period until you're familiar with the situation.

On to some more positive suggestions. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it must be divided too, for **ed** will not handle really big files. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

chap1
chap2
etc...

Or, if each chapter were broken into several files, you might have

chap1.1
chap1.2
chap1.3
...
chap2.1
chap2.2
...

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

pr chap1.1 chap1.2 chap1.3 .....

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

pr chap*

The ***** means ''anything at all,'' so this translates into ''print all files whose names begin with **chap**'', listed in alphabetical order.

This shorthand notation is not a property of the **pr** command, by the way. It is system-wide, a service of the program that interprets commands (the ''shell,'' **sh**(1)). Using that fact, you can see how to list the names of the files in the book:

ls chap*

produces

chap1.1
chap1.2
chap1.3
...

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

rm *junk* *temp*

removes all files that contain **junk** or **temp** as any part of their name. As a special case, ***** by itself matches every filename, so

pr *

prints all your files (alphabetical order), and

rm *

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

The ***** is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9. Then you can say

pr chap[12349]*

The **[...]** means to match any of the characters inside the brackets. A range of consecutive letters or digits

† In 4.2 BSD the limit was extended to 255 characters.

can be abbreviated, so you can also do this with

pr chap[1–49]*

Letters can also be used within brackets: **[a–z]** matches any character in the range **a** through **z**.

The **?** pattern matches any single character, so

ls ?

lists all files which have single-character names, and

ls –l chap?.1

lists information about the first file of each chapter (**chap1.1**, **chap2.1**, etc.).

Of these niceties, ***** is certainly the most useful, and you should get used to it. The others are frills, but worth knowing.

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

ls '?'

We'll see some more examples of this shortly.

What's in a Filename, Continued

When you first made that file called **junk**, how did the system know that there wasn't another **junk** somewhere else, especially since the person in the next office is also reading this tutorial? The answer is that generally each user has a private *directory*, which contains only the files that belong to him. When you log in, you are ''in'' your directory. Unless you take special action, when you create a new file, it is made in the directory that you are currently in; this is most often your own directory, and thus the file is unrelated to any other file of the same name that might exist in someone else's directory.

The set of all files is organized into a (usually big) tree, with your files located several branches into the tree. It is possible for you to ''walk'' around this tree, and to find any file in the system, by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start where you are and walk toward the root.

Let's try the latter first. The basic tools is the command **pwd** (''print working directory''), which prints the name of the directory you are currently in.

Although the details will vary according to the system you are on, if you give the command **pwd**, it will print something like

/usr/your-name

This says that you are currently in the directory **your-name**, which is in turn in the directory **/usr**, which is in turn in the root directory called by convention just /. (Even if it's not called **/usr** on your system, you will get something analogous. Make the corresponding mental adjustment and read on.)

If you now type

ls /usr/your-name

you should get exactly the same list of file names as you get from a plain **ls**: with no arguments, **ls** lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Next, try

ls /usr

This should print a long series of names, among which is your own login name **your-name**. On many systems, **usr** is a directory that contains the directories of all the normal users of the system, like you.

The next step is to try

ls /

You should get a response something like this (although again the details may be different):

bin
dev
etc
lib
tmp
usr

This is a collection of the basic directories of files that the system knows about; we are at the root of the tree.

Now try

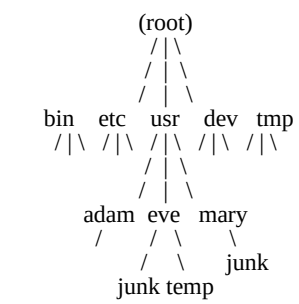
cat /usr/your-name/junk

(if **junk** is still around in your directory). The name

/usr/your-name/junk

is called the **pathname** of the file that you normally think of as ''junk''. ''Pathname'' has an obvious meaning: it represents the full name of the path you have to follow from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX system that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:



Notice that Mary's **junk** is unrelated to Eve's.

This isn'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, it becomes

handy indeed. For example, your friends can print your book by saying

pr /usr/your-name/chap*

Similarly, you can find out what files your neighbor has by saying

ls /usr/neighbor-name

or make your own copy of one of his files by

cp /usr/your-neighbor/his-file yourfile

If your neighbor doesn't want you poking around in his 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 set to control access. See **ls**(1) and **chmod**(1) for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy.

As a final experiment with pathnames, try

ls /bin /usr/bin

Do some of the names look familiar? When you run a program, by typing its name after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory (where it typically doesn't find it), then in **/bin** and finally in **/usr/bin**. There is nothing magic about commands like **cat** or **ls**, except that they have been collected into a couple of places to be easy to find and administer.

What if you work regularly with someone else on common information in his directory? You could just log in as your friend each time you want to, but you can also say ''I want to work on his files instead of my own''. This is done by changing the directory that you are currently in:

cd /usr/your-friend

(On some systems, **cd** is spelled **chdir**.) Now when you use a filename in something like **cat** or **pr**, it refers to the file in your friend'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. Of course, if you forget what directory you're in, type

pwd

to find out.

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, when you write your book, you might want to keep all the text in a directory called **book**. So make one with

mkdir book

then go to it with

cd book

then start typing chapters. The book is now found in (presumably)

/usr/your-name/book

To remove the directory **book**, type

rm book/*
rmdir book

The first command removes all files from the directory; the second removes the empty directory.

You can go up one level in the tree of files by saying

cd ..

''**..**'' is the name of the parent of whatever directory you are currently in. For completeness, ''**.**'' is an alternate name for the directory you are in.

Using Files instead of the Terminal

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 universal in UNIX systems 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 files on your terminal. But if you say

ls >filelist

a list of your files will be placed in the file **filelist** (which will be created if it doesn't already exist, or overwritten if it does). The symbol > means ''put the output on the following file, rather than on the terminal.'' Nothing is produced on the terminal. As another example, you could combine several files into one by capturing the output of **cat** in a file:

cat f1 f2 f3 >temp

The symbol >> operates very much like > does, except that it means ''add to the end of.'' That is,

cat f1 f2 f3 >>temp

means to concatenate **f1**, **f2** and **f3** to 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 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 could make up a script of commonly used editing commands and put them into a file called **script**. Then you can run the script on a file by saying

ed file <script

As another example, you can use **ed** to prepare a letter

in file **let**, then send it to several people with

mail adam eve mary joe <let

Pipes

One of the novel contributions of the UNIX system is the idea of a *pipe.* A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipeline.

For example,

pr f g h

will print the files **f**, **g**, and **h**, beginning each on a new page. Suppose you want them run together instead. You could say

cat f g h >temp
pr <temp
rm temp

but this is more work than necessary. Clearly what we want is to take the output of **cat** and connect it to the input of **pr**. So let us use a pipe:

cat f g h | pr

The vertical bar | means to take the output from **cat**, which would normally have gone to the terminal, and put it into **pr** to be neatly formatted.

There are many other examples of pipes. For example,

ls | pr –3

prints a list of your files in three columns. The program **wc** counts the number of lines, words and characters in its input, and as we saw earlier, **who** prints a list of currently-logged on people, one per line. Thus

who | wc

tells how many people are logged on. And of course

ls | wc

counts your files.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. You can have as many elements in a pipeline as you wish.

Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines. **pr** is one example:

pr –3 a b c

prints files **a**, **b** and **c** in order in three columns. But in

cat a b c | pr –3

pr prints the information coming down the pipeline, still in three columns.

The Shell

We have already mentioned once or twice the mysterious ‘‘shell,’’ which is in fact **sh**(1).† The shell is the program that interprets what you type as commands and arguments. It also looks after translating *, etc., into lists of filenames, and <, >, and | into changes of input and output streams.

The shell has other capabilities too. 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. Thus

date; who

does both commands before returning with a prompt character.

You can also have more than one program running *simultaneously* if you wish. For example, if you are doing something time-consuming, like the editor script of an earlier section, and you don’t want to wait around for the results before starting something else, you can say

ed file <script &

The ampersand at the end of a command line says ‘‘start this command running, then take further commands from the terminal immediately,’’ that is, don’t wait for it to complete. Thus the script will begin, but you can do something else at the same time. Of course, to keep the output from interfering with what you’re doing on the terminal, it would be better to say

ed file <script >script.out &

which saves the output lines in a file called **script.out**.

When you initiate a command with **&**, the system replies with a number called the process number, which identifies the command in case you later want to stop it. If you do, you can say

kill process-number

If you forget the process number, the command **ps** will tell you about everything you have running. (If you are desperate, **kill 0** will kill all your processes.) And if you’re curious about other people, **ps a** will tell you about *all* programs that are currently running.

You can say

(command-1; command-2; command-3) &

to start three commands in the background, or you can start a background pipeline with

† On Berkeley Unix systems, the usual shell for interactive use is the c shell, **csh(1).**

command-1 | command-2 &

Just as you can tell the editor or some similar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell, after all, is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (**tabs**, **date**, **who**) into a file, let's call it **startup**, and then run it with

sh startup

This says to run the shell with the file **startup** as input. The effect is as if you had typed the contents of **startup** on the terminal.

If this is to be a regular thing, you can eliminate the need to type **sh**: simply type, once only, the command

chmod +x startup

and thereafter you need only say

startup

to run the sequence of commands. The **chmod**(1) command marks the file executable; the shell recognizes this and runs it as a sequence of commands.

If you want **startup** to run automatically every time you log in, create a file in your login directory called **.profile**, and place in it the line **startup**. When the shell first gains control when you log in, it looks for the **.profile** file and does whatever commands it finds in it.† We'll get back to the shell in the section on programming.

III. DOCUMENT PREPARATION

UNIX systems are used extensively for document preparation. There are two major formatting programs, that is, programs that produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. **nroff** is designed to produce output on terminals and line-printers. **troff** (pronounced ''tee-roff'') instead drives a photo-typesetter, which produces very high quality output on photographic paper. This paper was formatted with **troff**.

Formatting Packages

The basic idea of **nroff** and **troff** is that the text to be formatted contains within it ''formatting commands'' that indicate in detail how the formatted text is to look. For example, there might be commands that specify how long lines are, whether to use single or double spacing, and what running titles to use on each page.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several ''packages'' of canned formatting requests are available to let you specify paragraphs, running titles, footnotes, multi-column output, and so on, with little effort and without having to learn **nroff** and **troff**. These packages take a modest effort to learn, but the rewards for using them are so great that it is time well spent.

In this section, we will provide a hasty look at the ''manuscript'' package known as **–ms**. Formatting requests typically consist of a period and two upper-case letters, such as **.TL**, which is used to introduce a title, or **.PP** to begin a new paragraph.

A document is typed so it looks something like this:

```
.TL
title of document
.AU
author name
.SH
section heading
.PP
paragraph ...
.PP
another paragraph ...
.SH
another section heading
.PP
etc.
```

The lines that begin with a period are the formatting requests. For example, **.PP** calls for starting a new paragraph. The precise meaning of **.PP** depends on what output device is being used (typesetter or terminal, for instance), and on what publication the document will appear in. For example, **–ms** normally assumes that a paragraph is preceded by a space (one line in **nroff**, ½ line in **troff**), and the first word is indented. These rules can be changed if you like, but they are changed by changing the interpretation of **.PP**, not by re-typing the document.

To actually produce a document in standard format using **–ms**, use the command

troff –ms files ...

for the typesetter, and

nroff –ms files ...

for a terminal. The **–ms** argument tells **troff** and **nroff** to use the manuscript package of formatting requests.

There are several similar packages; check with a local expert to determine which ones are in common use on your machine.

† The c shell instead reads a file called **.login**

Supporting Tools

In addition to the basic formatters, there is a host of supporting programs that help with document preparation. The list in the next few paragraphs is far from complete, so browse through the manual and check with people around you for other possibilities.

eqn and **neqn** let you integrate mathematics into the text of a document, in an easy-to-learn language that closely resembles the way you would speak it aloud. For example, the **eqn** input

sum from i=0 to n x sub i ~=~ pi over 2

produces the output

$$\sum_{i=0}^{n} x_i = \frac{\pi}{2}$$

The program **tbl** provides an analogous service for preparing tabular material; it does all the computations necessary to align complicated columns with elements of varying widths.

refer prepares bibliographic citations from a data base, in whatever style is defined by the formatting package. It looks after all the details of numbering references in sequence, filling in page and volume numbers, getting the author's initials and the journal name right, and so on.

spell and **typo** detect possible spelling mistakes in a document.† **spell** works by comparing the words in your document to a dictionary, printing those that are not in the dictionary. It knows enough about English spelling to detect plurals and the like, so it does a very good job. **typo** looks for words which are ''unusual'', and prints those. Spelling mistakes tend to be more unusual, and thus show up early when the most unusual words are printed first.

grep looks through a set of files for lines that contain a particular text pattern (rather like the editor's context search does, but on a bunch of files). For example,

grep ′ing$′ chap*

will find all lines that end with the letters **ing** in the files **chap***. (It is almost always a good practice to put single quotes around the pattern you're searching for, in case it contains characters like ***** or **$** that have a special meaning to the shell.) **grep** is often useful for finding out in which of a set of files the misspelled words detected by **spell** are actually located.

diff prints a list of the differences between two files, so you can compare two versions of something automatically (which certainly beats proofreading by hand).

wc counts the words, lines and characters in a set of files. **tr** translates characters into other characters; for example it will convert upper to lower case and vice versa. This translates upper into lower:

tr A–Z a–z <input >output

sort sorts files in a variety of ways; **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing). **sed** provides many of the editing facilities of **ed**, but can apply them to arbitrarily long inputs. **awk** provides the ability to do both pattern matching and numeric computations, and to conveniently process fields within lines. These programs are for more advanced users, and they are not limited to document preparation. Put them on your list of things to learn about.

Most of these programs are either independently documented (like **eqn** and **tbl**), or are sufficiently simple that the description in the *UNIX Programmer's Manual* is adequate explanation.

Hints for Preparing Documents

Most documents go through several versions (always more than you expected) before they are finally finished. Accordingly, you should do whatever possible to make the job of changing them easy.

First, when you do the purely mechanical operations of typing, type so that subsequent editing will be easy. Start each sentence on a new line. Make lines short, and break lines at natural places, such as after commas and semicolons, rather than randomly. Since most people change documents by rewriting phrases and adding, deleting and rearranging sentences, these precautions simplify any editing you have to do later.

Keep the individual files of a document down to modest size, perhaps ten to fifteen thousand characters. Larger files edit more slowly, and of course if you make a dumb mistake it's better to have clobbered a small file than a big one. Split into files at natural boundaries in the document, for the same reasons that you start each sentence on a new line.

The second aspect of making change easy is to not commit yourself to formatting details too early. One of the advantages of formatting packages like **–ms** is that they permit you to delay decisions to the last possible moment. Indeed, until a document is printed, it is not even decided whether it will be typeset or put on a line printer.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of requests like **.PP**, and then define them appropriately, either by using one of the canned packages (the better way) or by defining your own **nroff** and **troff** commands. As long as you have entered the text in some systematic way, it can always be cleaned up and reformatted by a judicious combination of editing commands and request definitions.

† "typo" is not provided with Berkeley Unix.

IV. PROGRAMMING

There will be no attempt made to teach any of the programming languages available but a few words of advice are in order. One of the reasons why the UNIX system is a productive programming environment is that there is already a rich set of tools available, and facilities like pipes, I/O redirection, and the capabilities of the shell often make it possible to do a job by pasting together programs that already exist instead of writing from scratch.

The Shell

The pipe mechanism lets you fabricate quite complicated operations out of spare parts that already exist. For example, the first draft of the **spell** program was (roughly)

```
cat ...      collect the files
| tr ...     put each word on a new line
| tr ...     delete punctuation, etc.
| sort       into dictionary order
| uniq       discard duplicates
| comm       print words in text
               but not in dictionary
```

More pieces have been added subsequently, but this goes a long way for such a small effort.

The editor can be made to do things that would normally require special programs on other systems. For example, to list the first and last lines of each of a set of files, such as a book, you could laboriously type

```
ed
e chap1.1
1p
$p
e chap1.2
1p
$p
etc.
```

But you can do the job much more easily. One way is to type

```
ls chap* >temp
```

to get the list of filenames into a file. Then edit this file to make the necessary series of editing commands (using the global commands of **ed**), and write it into **script**. Now the command

```
ed <script
```

will produce the same output as the laborious hand typing. Alternately (and more easily), you can use the fact that the shell will perform loops, repeating a set of commands over and over again for a set of arguments:

```
for i in chap*
do
        ed $i <script
done
```

This sets the shell variable **i** to each file name in turn, then does the command. You can type this command at the terminal, or put it in a file for later execution.

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, with variables, control flow (**if-else**, **while**, **for**, **case**), subroutines, and interrupt handling. Since there are many building-block programs, you can sometimes avoid writing a new program merely by piecing together some of the building blocks with shell command files.

We will not go into any details here; examples and rules can be found in *An Introduction to the UNIX Shell*, by S. R. Bourne.

Programming in C

If you are undertaking anything substantial, C is the only reasonable choice of programming language: everything in the UNIX system is tuned to it. The system itself is written in C, as are most of the programs that run on it. It is also a easy language to use once you get started. C is introduced and fully described in *The C Programming Language* by B. W. Kernighan and D. M. Ritchie (Prentice-Hall, 1978). Several sections of the manual describe the system interfaces, that is, how you do I/O and similar functions. Read *UNIX Programming* for more complicated things.

Most input and output in C is best handled with the standard I/O library, which provides a set of I/O functions that exist in compatible form on most machines that have C compilers. In general, it's wisest to confine the system interactions in a program to the facilities provided by this library.

C programs that don't depend too much on special features of UNIX (such as pipes) can be moved to other computers that have C compilers. The list of such machines grows daily; in addition to the original PDP-11, it currently includes at least Honeywell 6000, IBM 370 and PC families, Interdata 8/32, Data General Nova and Eclipse, HP 2100, Harris /7, Motorola 68000 family (including machines like Sun Microsystems and Apple Macintosh), VAX 11 family, SEL 86, and Zilog Z80. Calls to the standard I/O library will work on all of these machines.

There are a number of supporting programs that go with C. **lint** checks C programs for potential portability problems, and detects errors such as mismatched argument types and uninitialized variables.

For larger programs (anything whose source is on more than one file) **make** allows you to specify the dependencies among the source files and the processing steps needed to make a new version; it then checks the times that the pieces were last changed and does the minimal amount of recompiling to create a consistent updated version.

The debugger **adb** is useful for digging through the dead bodies of C programs, but is rather hard to learn to use effectively. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.†

The C compiler provides a limited instrumentation service, so you can find out where programs spend their time and what parts are worth optimizing. Compile the routines with the **–p** option; after the test run, use **prof** to print an execution profile. The command **time** will give you the gross run-time statistics of a program, but they are not super accurate or reproducible.

Other Languages

If you *have* to use Fortran, there are two possibilities. You might consider Ratfor, which gives you the decent control structures and free-form input that characterize C, yet lets you write code that is still portable to other environments. Bear in mind that UNIX Fortran tends to produce large and relatively slow-running programs. Furthermore, supporting software like **adb**, **prof**, etc., are all virtually useless with Fortran programs. There may also be a Fortran 77 compiler on your system. If so, this is a viable alternative to Ratfor, and has the non-trivial advantage that it is compatible with C and related programs. (The Ratfor processor and C tools can be used with Fortran 77 too.)

If your application requires you to translate a language into a set of actions or another language, you are in effect building a compiler, though probably a small one. In that case, you should be using the **yacc** compiler-compiler, which helps you develop a compiler quickly. The **lex** lexical analyzer generator does the same job for the simpler languages that can be expressed as regular expressions. It can be used by itself, or as a front end to recognize inputs for a **yacc**-based program. Both **yacc** and **lex** require some sophistication to use, but the initial effort of learning them can be repaid many times over in programs that are easy to change later on.

Most UNIX systems also make available other languages, such as Algol 68, APL, Basic, Lisp, Pascal, and Snobol. Whether these are useful depends largely on the local environment: if someone cares about the language and has worked on it, it may be in good shape. If not, the odds are strong that it will be more trouble than it's worth.

V. UNIX READING LIST

General:

K. L. Thompson and D. M. Ritchie, *The UNIX Programmer's Manual*, Bell Laboratories, 1978 (PS2:3)‡ Lists commands, system routines and interfaces, file formats, and some of the maintenance procedures. You can't live without this, although you will probably only need to read section 1.

D. M. Ritchie and K. L. Thompson, ''The UNIX Time-sharing System,'' CACM, July 1974. (PS2:1)‡ An overview of the system, for people interested in operating systems. Worth reading by anyone who programs. Contains a remarkable number of one-sentence observations on how to do things right.

The Bell System Technical Journal (BSTJ) Special Issue on UNIX, July/August, 1978, contains many papers describing recent developments, and some retrospective material.

The 2nd International Conference on Software Engineering (October, 1976) contains several papers describing the use of the Programmer's Workbench (PWB) version of UNIX.

Document Preparation:

B. W. Kernighan, ''A Tutorial Introduction to the UNIX Text Editor'' (USD:12) and ''Advanced Editing on UNIX,'' (USD:13) Bell Laboratories, 1978.† Beginners need the introduction; the advanced material will help you get the most out of the editor.

M. E. Lesk, ''Typing Documents on UNIX,'' Bell Laboratories, 1978. (USD:20)† Describes the **–ms** macro package, which isolates the novice from the vagaries of **nroff** and **troff**, and takes care of most formatting situations. If this specific package isn't available on your system, something similar probably is. The most likely alternative is the PWB/UNIX macro package **–mm**; see your local guru if you use PWB/UNIX.*

B. W. Kernighan and L. L. Cherry, ''A System for Typesetting Mathematics,'' Bell Laboratories Computing Science Tech. Rep. 17. (USD:26)†

M. E. Lesk, ''Tbl — A Program to Format Tables,'' Bell Laboratories CSTR 49, 1976. (USD:28)†

J. F. Ossanna, Jr., ''NROFF/TROFF User's Manual,'' Bell Laboratories CSTR 54, 1976. (USD:24)† **troff** is the basic formatter used by **–ms**, **eqn** and **tbl**. The reference manual is indispensable if you are going to write or maintain these or similar programs. But start with:

B. W. Kernighan, ''A TROFF Tutorial,'' Bell Laboratories, 1976. (USD:25)† An attempt to unravel the intricacies of **troff**.

† The "dbx" debugger, supplied starting with 4.2BSD, has extensive facilities for high-level debugging of C programs and is much easier to use than "adb".

† These documents (previously in Volume 2 of the Bell Labs Unix distribution) are provided among the "User Supplementary" Documents for 4.3BSD, available from the Usenix Association.
‡ These are among the "Programmer Supplementary" Documents for 4.3BSD. PS1 is Volume 1, PS2 is Volume 2.
*The macro package -me is additionally available on Berkeley Unix Systems. -mm is typically not available.

Programming:

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978. Contains a tutorial introduction, complete discussions of all language features, and the reference manual.

B. W. Kernighan and R. Pike, *The Unix Programming Environment*, Prentice-Hall, 1984. Contains many examples of C programs which use the system interfaces, and explanations of ''why''.

B. W. Kernighan and D. M. Ritchie, ''UNIX Programming,'' Bell Laboratories, 1978. (PS2:3)‡ Describes how to interface with the system from C programs: I/O calls, signals, processes.

S. R. Bourne, ''An Introduction to the UNIX Shell,'' Bell Laboratories, 1978. (USD:3)† An introduction and reference manual for the Version 7 shell. Mandatory reading if you intend to make effective use of the programming power of this shell.

S. C. Johnson, ''Yacc — Yet Another Compiler-Compiler,'' Bell Laboratories CSTR 32, 1978. (PS1:15)‡

M. E. Lesk, ''Lex — A Lexical Analyzer Generator,'' Bell Laboratories CSTR 39, 1975. (PS1:16)‡

S. C. Johnson, ''Lint, a C Program Checker,'' Bell Laboratories CSTR 65, 1977. (PS1:9)‡

S. I. Feldman, ''MAKE — A Program for Maintaining Computer Programs,'' Bell Laboratories CSTR 57, 1977. (PS1:12)‡

J. F. Maranzano and S. R. Bourne, ''A Tutorial Introduction to ADB,'' Bell Laboratories CSTR 62, 1977. (PS1:10)‡ An introduction to a powerful but complex debugging tool.

S. I. Feldman and P. J. Weinberger, ''A Portable Fortran 77 Compiler,'' Bell Laboratories, 1978. (PS1:2)‡ A full Fortran 77 for UNIX systems.