UNIX for Beginners

Brian W. Kernighan

Bell Laboratories, Murray Hill, N. J.

In many ways, UNIX is the state of the art in computer operating systems. From the user's point of view, it 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 point out high spots for new users, so they can get used to the main ideas of UNIX and start making good use of it quickly.

This paper is not an attempt to re-write the *UNIX Programmer's Manual;* often the discussion of something is simply "read section x in the manual." (This implies that you will need a copy of the *UNIX Programmer's Manual.)* Rather it suggests in what order to read the manual, and it collects together things that are stated only indirectly in the manual.

There are five sections:

1. Getting Started: How to log in to a UNIX, how to type, what to do about mistakes in typing, how to log out. Some of this is dependent on which UNIX 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 UNIX effectively: generally useful commands; the file system.
3. Document Preparation: Preparing manuscripts is one of the most common uses for UNIX. This section contains advice, but not extensive instructions on any of the formatting programs.
4. Writing Programs: UNIX is an excellent vehicle for developing programs. This section talks about some of the tools, but again is not a tutorial in any of the programming languages that UNIX provides.
5. A UNIX Reading List. An annotated bibliography of documents worth reading by new users.

I. GETTING STARTED

Logging In

Most of the details about logging in are in the manual section called "How to Get Started" (pages *iv-v* in the 5th Edition). Here are a couple of extra warnings.

You must have a UNIX login name, which you can get from whoever administers your system. You also need to know the phone number. UNIX is capable of dealing with a variety of terminals: Terminet 300's; Execuport, TI and similar portables; video terminals; GSI's; and even the venerable Teletype in its various forms. But note: UNIX will not handle IBM 2741 terminals and their derivatives (e.g., some Anderson-Jacobsons, Novar). Furthermore, UNIX is strongly oriented towards devices with *lower case.* If your terminal produces only upper case (e.g., model 33 Teletype), life will be so difficult that you should look for another terminal.

Be sure to set the switches appropriately on your device: speed (if it's variable) to 30 characters per second, lower case, full duplex, even parity, and any others that local wisdom advises. Establish a connection using whatever magic is needed for your terminal. UNIX should type "login:" at you. If it types garbage, you may be at the wrong speed; push the 'break' or 'interrupt' key once. 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 if the terminal has one. If a password is required, you will be asked for it, and (if possible) printing will be turned off while you type it, again followed by a RETURN. (On M37 Teletypes always use NEWLINE or LINEFEED in place of RETURN).

The culmination of your login efforts is a percent sign "%". The percent sign means that UNIX is ready to accept commands from the terminal. (You may also get a message of the day just before the percent sign or a notification that you have mail.)

Typing Commands

Once you've seen the percent sign, you can type commands, which are requests that UNIX do something. Try typing

```
date
```

followed by RETURN. You should get back something like

```
Sun Sep 22 10:52:29 EDT 1974
```

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. We won't show the carriage returns, but they have to be there.

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

```
who
```

gives something like

```
pjp   ttyf   Sep 22 09:40
bwk   ttyg   Sep 22 09:48
mel   ttyh   Sep 22 09:58
```

The time is when the user logged in.

If you make a mistake typing the command name, UNIX will tell you. For example, if you type

```
whom
```

you will be told

```
whom: not found
```

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. You can often fix this by logging out and logging back in. Or you can read the description of the command **stty** in section I of the manual. This will also tell you how to get intelligent treatment of tab characters (which are much used in UNIX) if your terminal doesn't have tabs. If it 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 the carriage 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:

```
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 over (on the same line!).

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

Readahead

UNIX has full readahead, 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 by UNIX and interpreted in the correct order. So you can type two 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). There are exceptions, like the text editor, where DEL stops whatever the program is doing but leaves you in that program. You can also just hang up the phone. The ‘‘interrupt’’ or ‘‘break’’ key found on most terminals has no effect.

Logging Out

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

```
login name–of–new–user
```

and let someone else use the terminal you were on. It is not sufficient just to turn off the terminal. UNIX has no time-out mechanism, so you'll be there forever unless you hang up.

Mail

When you log in, you may sometimes get the message

```
You have mail.
```

UNIX provides a postal system so you can send and receive letters from other users of the system. To read your mail, issue the command

```
mail
```

Your mail will be printed, and then you will be asked

```
Save?
```

If you do want to save the mail, type *y*, for ‘‘yes’’; any other response means ‘‘no’’.

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, usually called ‘‘EOT’’, is used throughout UNIX to mark the end of input from a terminal, so you might as well get used to it.

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** (I).

The notation **mail** (I) means the command **mail** in section (I) of the *UNIX Programmer's Manual.*

Writing to other users

At some point in your UNIX career, out of the blue will come a message like

```
Message from joe...
```

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 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 manual.)

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 (o+o), 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 ‘‘EOT’’ 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. It's also useful for getting the most up-to-date information on a command. To print a manual section, type ‘‘man section-name’’. Thus to read up on the **who** command, type

```
man who
```

If the section in question isn't in part I of the manual, you have to give the section number as well, as in

```
man 6 chess
```

Of course you're out of luck if you can't remember the section name.

II. DAY-TO-DAY USE

Creating Files _ The Editor

If we have to type a paper or a letter or a program, how do we 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** (I) 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 with some text in it, do the following:

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

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

```
w junk
```

ed will respond with the number of characters it wrote into the file called ‘‘junk’’.

Suppose we now add a few more lines with ‘‘a’’, terminate them with ‘‘.’’, and write the whole thing out as ‘‘temp’’, using

```
w temp
```

We should now have two files, a smaller one called ‘‘junk’’ and a bigger one (bigger by the extra lines) called ‘‘temp’’. Type a ‘‘q’’ to quit the editor.

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 we type

```
ls
```

the response will be

```
junk
temp
```

which are indeed our two files. They are sorted into alphabetical order automatically, but other variations are possible. For example, if we add the optional argument ‘‘-t’’,

```
ls –t
```

lists them 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 Sep 22 12:56 junk
–rw–rw–rw–  1 bwk   78 Sep 22 12:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters (you got the same thing from **ed**). ‘‘bwk’’ is the owner of the file _ 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’’ would give the same thing, 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** (I).

It is generally true of UNIX programs that ‘‘flag’’ arguments like ‘‘-t’’ precede filename arguments.

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 (about 65,000 characters or 4000 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 copies all the files in a list onto the terminal. So you can say

```
cat junk
```

or, to print two files,

```
cat junk temp
```

The two 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 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 list ‘‘junk’’ neatly, then skip to the top of a new page and list ‘‘temp’’ neatly.

pr will 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.

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 **roff**, **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 the hardware configuration of 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 a file 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 all of the files named. You will get a warning message if one of the named files wasn’t there.

Filename, What’s in a

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 already saw, for example, that in the **ls** command, ‘‘ls -t’’ meant 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. There are a number of other characters which have special meaning either to UNIX as a whole or to numerous commands. To avoid pitfalls, you would probably do well to use only letters, numbers and the period. (Don’t use the period as the first character of a filename, for reasons too complicated to go into.)

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 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** (I)). Using that fact, you can see how to list the files of 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. Thus

```
rm *junk*
```

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

```
pr *
```

prints all the files (alphabetical order), and

```
rm *
```

removes *all files.* (You had better be 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 of the book. Then you can say

```
pr chap[12349]*
```

The ‘‘[...]’’ means to match any of the characters inside the brackets. You can also do this with

```
pr chap[1–49]*
```

‘‘[a-z]’’ matches any character in the range *a* through *z*. There is also a ‘‘?’’ character, which matches any single character, so

```
pr ?
```

will print all files which have single-character names.

Of these niceties, ‘‘*’’ is probably 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 quotes (single or double), as in

```
ls "?"
```

What’s in a Filename, Continued

When you first made that file called ‘‘junk’’, how did UNIX know that there wasn’t another ‘‘junk’’ somewhere else, especially since the person in the next office is also reading this tutorial? The reason is that generally each user of UNIX has his own ‘‘directory’’, which contains only the files that belong to him. When you create a new file, unless you take special action, the new file is made in your own directory, and is unrelated to any other file of the same name that might exist in someone else’s directory.

The set of all files that UNIX knows about are organized into a (usually big) tree, with your files located several branches up 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 right set of branches.

To begin, type

```
ls /
```

‘‘/’’ is the name of the root of the tree (a convention used by UNIX). You will get a response something like this:

```
bin
dev
etc
lib
tmp
usr
```

This is a collection of the basic directories of files that UNIX knows about. On most systems, ‘‘usr’’ is a directory that contains all the normal users of the system, like you. Now try

```
ls  /usr
```

This should list a long series of names, among which is your own login name. Finally, try

```
ls  /usr/your–name
```

You should get what you get from a plain

```
ls
```

Now try

```
cat  /usr/your–name/junk
```

(if ‘‘junk’’ is still around). 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 through the tree of directories to get to a particular file. It is a universal rule in UNIX that anywhere you can use an ordinary filename, you can use a pathname.

Here is a picture which may make this clearer:

```
                  (root)
                   / | \
                  /  |  \
                 /   |   \
        bin    etc    usr    dev   tmp
       / | \   / | \  / | \  / | \  / | \
                     /  |  \
                    /   |   \
              adam   eve   mary
               /     / \      \
                    /   \    junk
               junk  temp
```

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 can have read-write-execute permissions for the owner, a group, and everyone else, to control access. See **ls** (I) and **chmod** (I) 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 a ‘‘%’’, the system simply looks for a file of that name. It 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 two 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:

```
chdir  /usr/your–friend
```

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.

If you forget what directory you're in, type

```
pwd
```

(‘‘print working directory’’) to find out.

It is often convenient to arrange one's 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

```
chdir book
```

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

```
/usr/your–name/book
```

To delete a directory, see **rmdir** (I).

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

```
chdir ..
```

‘‘..’’ 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 that the terminal can be replaced by a file for either or both of input and output. As one example, you could say

```
ls
```

to get a list of files. But you can also say

```
ls >filelist
```

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

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

Similarly, 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
```

Pipes

One of the novel contributions of UNIX 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 pipe-line.

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**, which formats it neatly.

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.

The Shell

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

The shell has other capabilities too. For example, you can start 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 ‘‘%’’.

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.’’ 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 have said

```
ed file <script >lines &
```

which would save the output lines in a file called ‘‘lines’’.

When you initiate a command with ‘‘&’’, UNIX 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
```

You might also read **ps** (I).

You can say

```
(command–1; command–2; command–3) &
```

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

```
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 ‘‘xxx’’, and then run it with either

```
sh xxx
```

or

```
sh <xxx
```

This says to run the shell with the file ‘‘xxx’’ as input. The effect is as if you had typed the contents of ‘‘xxx’’ on the terminal. (If this is to be a regular thing, you can eliminate the need to type ‘‘sh’’; see **chmod** (I) and **sh** (I).)

The shell has quite a few other capabilities as well, some of which we'll get to in the section on programming.

III. DOCUMENT PREPARATION

UNIX is extensively used for document preparation. There are three major *formatting* programs, that is, programs which produce a text with justified right margins, automatic page numbering and titling, automatic hyphenation, and the like. The simplest of these formatters is **roff**, which in fact is simple enough that if you type almost any text into a file and ‘‘roff’’ it, you will get plausibly formatted output. You can do better with a little knowledge, but basically it's easy to learn and use. We'll get back to **roff** shortly.

nroff is similar to **roff** but does much less for you automatically. It will do a great deal more, once you know how to use it.

Both **roff** and **nroff** are designed to produce output on terminals, line-printers, and the like. The third formatter, **troff** (pronounced ‘‘tee-roff’’), instead drives a Graphic Systems phototypesetter, which produces very high quality output on photographic paper. This paper was printed on the phototypesetter by **troff**.

Because **nroff** and **troff** are relatively hard to learn to use effectively, several ‘‘packages’’ of canned formatting requests are available which let you do things like paragraphs, running titles, multi-column output, and so on, with little effort. Regrettably, details vary from system to system.

ROFF

The basic idea of **roff** (and of **nroff** and **troff**, for that matter) 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. In general, you don't have to spell out all of the possible formatting details. Most of them have ‘‘default values’’, which you will get if you say nothing at all. For example, unless you take special precautions, you'll get single-spaced output, 65-character lines, justified right margins, and 58 text lines per page when you **roff** a file. This is the reason that **roff** is so simple _ most of the decisions have already been made for you.

Some things do have to be done, however. If you want a document broken into paragraphs, you have to tell **roff** where to add the extra blank lines. This is done with the ‘‘.sp’’ command:

```
this is the end of one paragraph.
.sp
This begins the next paragraph ...
```

In **roff** (and in **nroff** and **troff**), formatting commands consist of a period followed by two letters, and they must appear at the beginning of a line, all by themselves. The ‘‘.sp’’ command tells **roff** to finish printing any of the previous line that might be still unprinted, then print a blank line before continuing. You can have more space if you wish; ‘‘.sp 2’’ asks for 2 spaces, and so on.

If you simply want to ensure that subsequent text appears on a fresh output line, you can use the command ‘‘.br’’ (for ‘‘break’’) instead of ‘‘.sp’’.

Most of the other commonly-used **roff** commands are equally simple. For example you can center one or more lines with the ‘‘.ce’’ command.

```
.ce
Title of Paper
.sp 2
```

causes the title to be centered, then followed by two blank lines. As with ‘‘.sp’’, ‘‘.ce’’ can be followed by a number; in that case, that many input lines are centered.

‘‘.ul’’ underlines lines, and can also be followed by a number:

```
.ce 2
.ul 2
An Earth–shaking Paper
.sp
John Q. Scientist
```

will center and underline the two text lines. Notice that the ‘‘.sp’’ between them is not part of the line count.

You can get multiple-line spacing instead of the default single-spacing with the ‘‘.ls’’ command:

```
.ls 2
```

causes double spacing.

If you're typing things like tables, you will not want the automatic filling-up and justification of output lines that is done by default. You can turn this off with the command ‘‘.nf’’ (no-fill), and then back on again with ‘‘.fi’’ (fill). Thus

```
this section is filled by default.
.nf
here lines will appear just
as you typed them _
no extra spaces, no moving of words.
.fi
Now go back to filling up output lines.
```

You can change the line-length with ‘‘.ll’’, and the left margin (the indent) by ‘‘.in’’. These are often used together to make offset blocks of text:

```
.ll −10
.in +10
this text will be moved 10 spaces to
the right and the lines will also be
shortened 10 characters from the
right. The ‘‘+’’ and ‘‘−’’ mean to
change the previous value by that
much. Now revert:
.ll +10
.in −10
```

Notice that ‘‘.ll +10’’ adds ten characters to the line length, while ‘‘.ll 10’’ makes the line ten characters *long.*

The ‘‘.ti’’ command indents (in either direction) just like ‘‘.in’’, except for only one line. Thus to make a new paragraph with a 10-character indent, you would say

```
.sp
.ti +10
New paragraph ...
```

You can put running titles on both top and bottom of each page, like this:

```
.he "left top"center top"right top"
.fo "left bottom"center bottom"right bottom"
```

The header or footer is divided into three parts, which are marked off by any character you like. (We used a double quote.) If there's nothing between the markers, that part of the title will be blank. If you use a percent sign anywhere in ‘‘.he’’ or ‘‘.fo’’, the current page number will be inserted. So to get centered page numbers with dashes around them, at the top, use

```
.he ""− % −""
```

You can skip to the top of a new page at any time with the ‘‘.bp’’ command; if ‘‘.bp’’ is followed by a number, that will be the new page number.

The foregoing is probably enough about **roff** for you to go off and format most everyday documents. Read **roff** (I) for more details.

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 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.

The second aspect of making change easy is not to commit yourself to formatting details too early. For example, if you decide that each paragraph is to have a space and an indent of 10 characters, you might type, before each,

```
.sp
.ti +10
```

But what happens when later you decide that it would have been better to have no space and an indent of only 5 characters? It's tedious indeed to go back and patch this up.

Fortunately, all of the formatters let you delay decisions until the actual moment of running. The secret is to define a new operation (called a *macro),* for each formatting operation you want to do, like making a new paragraph. You can say, in all three formatters,

```
.de PP
.sp
.ti +10
..
```

This *defines* ‘‘.PP’’ as a new **roff** (or **nroff** or **troff**) operation, whose meaning is exactly

```
.sp
.ti +10
```

(The ‘‘..’’ marks the end of the definition.) Whenever ‘‘.PP’’ is encountered in the text, it is as if you had typed the two lines of the definition in place of it.

The beauty of this scheme is that now, if you change your mind about what a paragraph should look like, you can change the formatted output merely by changing the definition of ‘‘.PP’’ and re-running the formatter.

As a rule of thumb, for all but the most trivial jobs, you should type a document in terms of a set of macros like ‘‘.PP’’, and then define them appropriately. As long as you have entered the text in some systematic way, it can always be cleaned up and re-formatted by a judicious combination of editing and macro definitions. The packages of formatting commands that we mentioned earlier are simply collections of macros designed for particular formatting tasks.

One of the main differences between **roff** and the other formatters is that macros in **roff** can only be lines of text and formatting commands. In **nroff** and **troff**, macros may have arguments, so they can have different effects depending on how they are called (in exactly the same way that the ‘‘.sp’’ command has an argument, the number of spaces you want).

Miscellany

In addition to the basic formatters, UNIX provides a host of supporting programs. **eqn** and **neqn** let you integrate mathematics into the text of a document, in a language that closely resembles the way you would speak it aloud. **spell** and **typo** detect possible spelling mistakes in a document. **grep** looks for lines containing a particular text pattern (rather like the editor's context search does, but on a whole series of files). For example,

```
grep "ing$" chap*
```

will find all lines ending in the letters ‘‘ing’’ in the series of files ‘‘chap*’’. (It is almost always a good practice to put quotes around the pattern you're searching for, in case it contains characters that have a special meaning for the shell.)

wc counts the words and (optionally) lines 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]"
```

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). **sort** sorts files in a variety of ways; **cref** makes cross-references; **ptx** makes a permuted index (keyword-in-context listing).

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

IV. PROGRAMMING

UNIX is a marvelously pleasant and productive system for writing programs; productivity seems to be an order of magnitude higher than on other interactive systems.

There will be no attempt made to teach any of the programming languages available on UNIX, but a few words of advice are in order. First, UNIX is written in C, as is most of the applications code. If you are undertaking anything substantial, C is the only reasonable choice. More on that in a moment. But remember that there are quite a few programs already written, some of which have substantial power.

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, say a book, you could laboriously type

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

But instead you can do the job once and for all. 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.

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

```
cat ...  (collect the files)
| tr ...  (put each word on a new line,
            delete punctuation, etc.)
| sort  (into dictionary order)
| uniq (strip out duplicates)
| comm        (list words found in text but
            not in dictionary)
```

Programming the Shell

An option often overlooked by newcomers is that the shell is itself a programming language, and since UNIX already has a host of building-block programs, you can sometimes avoid writing a special purpose program merely by piecing together some of the building blocks with shell command files.

As an unlikely example, suppose you want to count the number of users on the machine every hour. You could type

```
date
who | wc –l
```

every hour, and write down the numbers, but that is rather primitive. The next step is probably to say

```
(date; who | wc –l) >>users
```

which uses ‘‘>>’’ to *append* to the end of the file ‘‘users’’. (We haven’t mentioned ‘‘>>’’ before _ it’s another service of the shell.) Now all you have to do is to put a loop around this, and ensure that it’s done every hour. Thus, place the following commands into a file, say ‘‘count’’:

```
: loop
(date; who | wc –l) >>users
sleep  3600
goto loop
```

The command **:** is followed by a space and a label, which you can then **goto**. Notice that it’s quite legal to branch backwards. Now if you issue the command

```
sh  count  &
```

the users will be counted every hour, and you can go on with other things. (You will have to use **kill** to stop counting.)

If you would like ‘‘every hour’’ to be a parameter, you can arrange for that too:

```
: loop
(date; who | wc – l) >>users
sleep  $1
goto loop
```

‘‘$1’’ means the first argument when this procedure is invoked. If you say

```
sh count 60
```

it will count every minute. A shell program can have up to nine arguments, ‘‘$1’’ through ‘‘$9’’.

The other aspect of programming is conditional testing. The **if** command can test conditions and execute commands accordingly. As a simple example, suppose you want to add to your login sequence something to print your mail if you have some. Thus, knowing that mail is stored in a file called ‘mailbox’, you could say

```
if –r mailbox  mail
```

This says ‘‘if the file ‘mailbox’ is readable, execute the **mail** command.’’

As another example, you could arrange that the ‘‘count’’ procedure count every hour by default, but allow an optional argument to specify a different time. Simply replace the ‘‘sleep $1’’ line by

```
if $1x = x sleep 3600
if $1x != x sleep $1
```

The construction

```
if $1x = x
```

tests whether ‘‘$1’’, the first argument, was present or absent.

More complicated conditions can be tested: you can find out the status of an executed command, and you can combine conditions with 'and', 'or', 'not' and parentheses _ see **if** (I). You should also read **shift** (I) which describes how to manipulate arguments to shell command files.

Programming in C

As we said, C is the language of choice: everything in UNIX is tuned to it. It is also a remarkably easy language to use once you get started. Sections II and III of the manual describe the system interfaces, that is, how you do I/O and similar functions.

You can write quite significant C programs with the level of I/O and system interface described in *Programming in C: A Tutorial*, if you use existing programs and pipes to help. For example, rather than learning how to open and close files you can (at least temporarily) write a program that reads from its standard input, and use **cat** to concatentate several files into it. This may not be adequate for the long run, but for the early stages it's just right.

There are a number of supporting programs that go with C. The C debugger, **cdb**, is marginally useful for digging through the dead bodies of C programs. **db**, the assembly language debugger, is actually more useful most of the time, but you have to know more about the machine and system to use it well. The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

You can instrument C programs and thus find out where they spend their time and what parts are worth optimising. 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 it's not super accurate or reproducible.

C programs that don't depend too much on special features of UNIX can be moved to the Honeywell 6070 and IBM 370 systems with modest effort. Read *The GCOS C Library* by M. E. Lesk and B. A. Barres for details.

Miscellany

If you *have* to use Fortran, 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 **db**, **prof**, etc., are all virtually useless with Fortran programs.

If you want to use assembly language (all heavens forfend!), try the implementation language LIL, which gives you many of the advantages of a high-level language, like decent control flow structures, but still lets you get close to the machine if you really want to.

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.

V. UNIX READING LIST

General:

UNIX Programmer's Manual (Ken Thompson, Dennis Ritchie, and a cast of thousands). 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 read section I.

The UNIX Time-sharing System (Ken Thompson, Dennis Ritchie). CACM, July 1974. 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.

Document Preparation:

A Tutorial Introduction to the UNIX Text Editor. (Brian Kernighan). Bell Laboratories internal memorandum. Weak on the more esoteric uses of the editor, but still probably the easiest way to learn **ed**.

Typing Documents on UNIX. (Mike Lesk). Bell Laboratories internal memorandum. A macro package to isolate the novice from the vagaries of the formatting programs. If this specific package isn't available on your system, something similar probably is. This one works with both **nroff** and **troff**.

Programming:

Programming in C: A Tutorial (Brian Kernighan). Bell Laboratories internal memorandum. The easiest way to start learning C, but it's no help at all with the interface to the system beyond the simplest IO. Should be read in conjunction with

C Reference Manual (Dennis Ritchie). Bell Laboratories internal memorandum. An excellent reference, but a bit heavy going for the beginner, especially one who has never used a language like C.

Others:

D. M. Ritchie, UNIX Assembler Reference Manual.

B. W. Kernighan and L. L. Cherry, A System for Typesetting Mathematics, Computing Science Tech. Rep. 17.

M. E. Lesk and B. A. Barres, The GCOS C Library. Bell Laboratories internal memorandum.

K. Thompson and D. M. Ritchie, Setting Up UNIX.

M. D. McIlroy, UNIX Summary.

D. M. Ritchie, The UNIX I/O System.

A. D. Hall, The M6 Macro Processor, Computing Science Tech. Rep. 2.

J. F. Ossanna, NROFF User's Manual _ Second Edition, Bell Laboratories internal memorandum.

D. M. Ritchie and K. Thompson, Regenerating System Software.

B. W. Kernighan, Ratfor_A Rational Fortran, Bell Laboratories internal memorandum.

M. D. McIlroy, Synthetic English Speech by Rule, Computing Science Tech. Rep. 14.

M. D. McIlroy, Bell Laboratories internal memorandum.

J. F. Ossanna, TROFF Users' Manual, Bell Laboratories internal memorandum.

B. W. Kernighan, TROFF Made Trivial, Bell Laboratories internal memorandum.

R. H. Morris and L. L. Cherry, Computer Detection of Typographical Errors, Computing Science Tech. Rep. 18.

S. C. Johnson, YACC (Yet Another Compiler-Compiler), Bell Laboratories internal memorandum.

P. J. Plauger, Programming in LIL: A Tutorial, Bell Laboratories internal memorandum.

Index