Introduction to UNIX

Academic Computing and Instructional Technology Services
The University of Texas at Austin
Austin, Texas 78712-1110

CCUG-1, Revised, November 1997

This manual is for new users of the ACITS UNIX Timesharing System (UTS), CCWF UNIX system (CCWF), and the Academic Data Server (ADS). Each computer system runs under a different version of the UNIX operating system: UTS runs Digital UNIX, CCWF runs Solaris, and ADS runs AIX. Although UNIX functions essentially the same way on all these, the systems will be referred to in this manual as UTS, CCWF, and ADS whenever there is a need to distinguish among them.

The reader of this manual is assumed to have some prior experience with computers, but not necessarily with UNIX. This manual does not attempt to teach any programming language, but it does explain how to:

- Connect to a UNIX system, log in, and log out.
- Change your password.
- Set up your terminal or workstation to communicate with a UNIX system.
- Display online help resources and get printed copies of online documents.
- Create and maintain files and directories of files.
- Use an editing program to alter files.
- Print files and move them to and from other computer systems.
- Compile and run programs you have written.
- Send mail to other users and read messages they send to you.
- Do elementary shell programming.

If you want to learn only enough UNIX to publish a Web page from a UNIX server, see

`www.utexas.edu/learn/pub/`

and select the UNIX topic.

CONTENTS

---

21 November 97
Documentation Group, ACITS at UT Austin
Comments to: remark@cc.utexas.edu

1. Introduction

Conventions Used Here

12. UNIX command words and their arguments are shown exactly as you should type them, including the distinction between uppercase letters and lowercase letters, which are not interchangeable on UNIX systems. For example, UNIX treats **MAIL**, **Mail**, and **mail** as three distinct entities.
13. If an item is shown in *italics*, substitute an appropriate real value for it, such as an actual filename, user name, directory, or whatever.
14. If an item in a command syntax line is shown in square brackets [ ], it's optional. Don't type the brackets in the command itself, but do type any other punctuation shown, such as a hyphen.
15. If an ellipsis ... is shown, it means you can repeat the previous item shown in the syntax line.
16. No command line is actually processed by the UNIX system until you press the **Return** key. In most cases, this manual does not show the Return.
17. Type control characters by holding down the key marked Control or CTRL while you press another key. Control characters are indicated in this manual with a caret character ^. For example, ^**c** means hold down the **Control** key while you press **c**.
18. The backspace key may be marked DEL, DELETE, RUB, or RUBOUT on your keyboard. **DEL** is used here.

To illustrate the conventions, here is the format of the **cp** command, which copies one or more files into a directory:

cp [-i] [-r] *file ... directory*

The command name **cp** must be given in lower case; the **-i** and **-r** arguments may be omitted or given exactly as shown; then one or more filenames of actual files; and finally an actual directory name.

Special Characters

Many punctuation characters and control characters have special meaning to UNIX. Even before your first login, you should know the ones below. Others are given in Chapter 4, ``The Shell and Command Processing".

Correcting Typing Mistakes

Before you press the **Return** key, you can correct a typing error by using:

```
DEL     to erase the previous character.
^w      to erase the previous word.
^u      to discard the whole command line.
```

Signaling End of Input ^d

If you are typing input to a process, press ^**d** to signal the end of input.
If you have used a VMS or DOS system, you may be familiar with typing ^**z** for this purpose, but on UNIX systems, ^**z** suspends a process: it does not end input.

Aborting a Process or Discarding Output ^c and ^o

You can abort a process and discard its output by pressing ^**c**. You can also discard output or skip over part of the display by using ^**o**. When you do this, the program producing the output does not stop, but its output is discarded until you give another ^**o** or until no more output remains.

2. Getting Access to the UNIX Systems

Connecting to a UNIX System

Before you can use any ACITS UNIX system, you must have a user number that is validated for the system you will use. Forms to request validation are available at the Help Desk at the Varsity Center. The ACITS UNIX systems include:

- UTS system (UNIX Timesharing System) --
  DEC AlphaServers running Digital UNIX (OSF/1); general timesharing.
  Network hostname: **uts.cc.utexas.edu**
- CCWF UNIX system --
  SPARCcenter 2000 running Solaris 2.5; general computing.
  Network name: **ccwf.cc.utexas.edu**
- ADS system (Academic Data Server) --
  IBM Model R40 running AIX 4.1; general computing and database service.
  Network hostname: **spice.cc.utexas.edu**

Computer systems at the Texas Advanced Computing Center also run under the UNIX operating system, but those specialized computers are not discussed here.

Connecting by Using the Telesys Dial-Up System

To connect to a UNIX system by using a telephone and modem with your computer or terminal, validate your user number for the Telesys dial-up system, as well as for the UNIX system you want. Telesys validation forms are also at the Help Desk.

If you are using a computer or terminal on campus, you will probably get access to ACITS UNIX systems through a network connection from your campus computer--see below.

If you use the connection software that comes with the UT Connect package, that software will make the Telesys connection, and you can then use the **telnet** program (also a part of UT Connect) to connect to a UNIX host computer. (The first time you use Telesys, use the original password that came with your user number.)

To change your Telesys password, run **telnet** and connect to **telesys.utexas.edu**. Then follow the instructions on screen.

Note: Changing your Telesys password or username has no effect on accounts or passwords on any other computers to which you connect.

For more information about Telesys, see *Dial up to the Internet with Telesys* (usage note Gen-11).

Connecting from Another Computer

If you are already logged in on another computer that is connected to the Internet, you should be able to connect to an ACITS UNIX system by giving a **telnet, rlogin,** or **ssh** command. For these, include the full hostname if your local machine is not one of ACITS' own computers. For example, type

```
telnet uts.cc.utexas.edu
```

to connect to UTS and get its **login:** prompt.

If you want to connect by using **telnet, rlogin,** or **ssh,** read the online help for these commands on your local system.

Logging In the First Time

Your first login session on any UNIX system will serve only to set up accounting information for your user number. In response to the **login:** prompt on your first UNIX login, type the 7-character user number (in **lower** case) issued to you by ACITS, and press the Return key. UNIX then prompts for your password:

Password:
Type, in lower case, the password issued to you when you received your user number, followed by Return. For security reasons, the password will not be displayed when you type it.
Next, UNIX displays any message of the day that may be posted. Such messages usually contain important system announcements.
Because you have never logged in on this system before, a special new-user program now runs. It may ask you for a UNIX login name that you will use for future logins (and for your electronic mail address). If it does, choose up to 8 lowercase letters that help others identify you for e-mail purposes--such as your first initial and surname. If someone else has already used the login name you choose, the machine will not accept it, and you must select something else. If you are not asked for a login name, your user number will be your login name and your e-mail name.
The new-user program will then require you to change your password by asking for a new one. When you type in that new password, you will be asked to type it again to make sure that you didn't mistype what you intended. The password you type will not display on the screen.
Choose a password at least 6 characters long that is meaningful ONLY to you; you don't want anyone else to guess it easily. In fact, if your password is easy to guess, the UNIX system will not even accept it. For example, it will not accept words from a dictionary. Don't choose your initials, nickname, or birthdate. Mixing uppercase letters and digits in your password to make it hard to guess. This password is the only thing that keeps someone else from using your account and having ull access to your work.
Be sure the password is something you can remember yourself. If you forget it, you will have to go in person, with proper identification, to the Help Desk at the Varsity Center to ask the staff to assign you a new one. If the password is for a user number under a departmental computer account, such as those accounts used for classes, the account sponsor must write a letter on departmental stationery stating that you are authorized to use that user number and that the password can be changed.
The new-user program may ask you to choose which command interpreter, called a ``shell'', you want to have as your login shell. On some UNIX systems, you are given the C shell, which is sometimes designated by its directory, /bin/csh. The C shell is recommended.
The program may also ask for optional data to put in a database that other users can look at. This data can include your real name (rather than your login name), address, phone number, and projects you are working on. If you don't want to include any individual item, just press Return in response to the prompt for it.
If you later want to add data or change data in this database, use the command

```
chfn
```

Exceptions: For the UTS system, you must run **chfn** on the machine named **curly.cc.utexas.edu.** For the CCWF system, send your request to change the data to

```
remark@cc.utexas.edu
```

When the new-user program has completed, you will be logged off the system automatically. After an hour or so, your account information should be updated, and you can log in again with your new login name and password.

Future Logins

After your first UNIX session, use your new login name (if you chose one) and password in response to the **login:** and **Password:** prompts. After you type these, you will see a line showing the time and date of your most recent login on that computer system. Pay attention to this line: it could indicate a login by someone who discovered your password and logged in under your user number.
Next, UNIX displays the message of the day, if there is one. Should the message scroll off the screen before you can read it--or if you want to see it again later--redisplay it by typing

```
more /etc/motd
```

Then you will see a line that looks like this:

```
TERM = (vt100)
```

If the terminal you are using is really a VT100 or is emulating a VT100 (such as a Macintosh or Windows computer running a terminal emulation program), just press Return. If your terminal is some other type, type in the correct value. If you don't know what kind of terminal you are using, just press Return to accept the vt100 value: it is the most common setting and will probably work well. You can call the Help Desk at 475-9400 if you suspect your terminal settings are causing problems.

Also, before using some programs, such as the **pine** mail program, the **trn** news reader, or the **vi** text editor, your terminal type must be defined. Under the C shell, you can do this by giving the command

setenv TERM *type*

where *type* identifies your terminal type, such as **vt100** or **xterm.**

Finally, you will see a prompt indicating that UNIX is ready for your commands. On some ACITS UNIX machines, the prompt includes the hostname (the machine's network name, such as **moe.cc.utexas.edu**). On other UNIX systems, this prompt may be a **$** or **%** character.

Login Initialization Files

Whenever you log in, the UNIX shell searches your directory for certain initialization files and executes them. Assuming you chose the C shell (/bin/csh) as your login shell, you can put commands into your initialization file, named .login (``dot-login''), to set up your terminal or do other routine tasks. To insert command lines into .login, you must use an editing program, as described in chapter 6. Here are sample lines from the default .login file given to new users:

```
# Sample .login for new csh users #
# The following line is used to set your terminal type
# when you login.
setenv TERM `tset - -Q -m 'unknown:?vt100' -m 'su:?vt100' -m 'dumb:?vt100' -m
'network:?vt100'`
setenv EDITOR vi
set mail = (0 /var/spool/mail/$USER)
# Current directory put at end of path for security reasons setenv PATH
/usr/local/bin:${PATH}:
# Delete '# ' at beginning of next line if you do NOT want ^D to log you out.
# set ignoreeof
```

The lines beginning with the pound sign (#) are explanatory comments. All the other lines are commands that are executed each time you log in. The line that begins ``setenv TERM'' is the one that causes the prompt:

```
TERM = (vt100)
```

each time you log in. If you always use a different type of terminal, you could edit this line to put that type everywhere it now has ``vt100''. Be sure to use the accent grave (`) and the apostrophe (') exactly as they are shown in the original line.

The line that begins ``setenv EDITOR'' identifies your default editor. Any value set by the **setenv** command is called an ``environment variable''. Many programs use the values of environment variables by default. For example, whenever you print a file, the output goes to the printer identified by your PRINTER environment variable, unless you explicitly send it to a different printer. So if you normally send your printed output to a printer in the Academic Center Workstation Lab, you could put this line into your .login file:

```
setenv PRINTER acwl_lw
```

The line that begins ``set mail'' tells where to look for your incoming mail. In this line, $USER means your own login name, whatever that may be.

If for some reason you do not have a .login file, or if you ruin yours with an editing mistake, you can copy the default file from /usr/local/lib/Login to your own directory. (See chapter 5 to learn how to do this.)

Changing Your Password

Your password is the only thing that keeps others from impersonating you on the system and getting full access to your files and computing resources. To protect your files and your user number, it is a good idea to change your password occasionally. To do this, type the command

```
passwd
```

You will be prompted first for your current password, then twice for a new one (the second time for verification). Neither your old password nor your new one will appear on the screen when you type it. Remember that the password should be something you can remember but others can't easily guess. It can be 6 to 8 characters long (6 to 16 character long on UTS). On some ACITS UNIX systems, the **passwd** program rejects passwords that match anything you put into the personal database (for example, your nickname), words from a dictionary, or words and names that people have used in the past to try to break into computer accounts.

Logging Out

At the end of the UNIX session, **don't just hang up the phone or turn off your terminal or workstation.** And don't ignore a ``frozen" terminal, assuming that your computer job will terminate on its own. Always log out before you leave. To do so, type the command

logout or **exit**

Some UNIX systems give no job summary or logout message. You simply see another prompt, depending on how you connected. When you see this prompt, type

```
exit
```

An alternative to typing **logout** is to type ^d, which signals the end of input you are typing into a file. If you are at command level instead of typing text into a file, then **^d** ends your UNIX session. To make the command interpreter ignore this ``end-of-file" signal so that you won't log out by accident--say if you inadvertently type ^d twice instead of once when you finish typing input to a file--give the command

```
set ignoreeof
```

at the beginning of each UNIX session. Or you can put the command in your .login file. NOTE: The **set ignoreeof** command works only if the C shell (or some variant of it, such as **tcsh**) is your command interpreter.

Should you get another login prompt when you type **logout,** type **^d** or **exit** to log out.

When you try to log out, you might see the message, ``There are suspended jobs." This means you have suspended some process that you should finish or kill before you log out. See chapter 4 to learn how to do this.

If your terminal seems to lock up and you cannot log out, you might have a runaway or disconnected job. Chapter 9 describes how to fix this problem.

3. Getting Help While on the System

Finding Out What's Available

All commands on the UNIX system are described online in a collection of files called as ``man pages'', because they were originally pages from the *UNIX Programmer's Manual.*
NOTE: On the ADS system, get online help by using the InfoExplorer program. Type **info -h** to get started.
The full set of man pages is organized into many sections having titles such as:

```
commands            games
library routines    system maintenance
special files       device drivers
file formats        system calls
miscellaneous
```

On most UNIX systems, you can find out what the categories are by typing:

`apropos intro`

If you know the name of a command, you can view its man page at your terminal. If you don't know its name, you can use the **apropos** command, which searches through the header lines of the man pages for whatever keyword you supply and shows you a list of the man pages it finds. To use it, type

`apropos` *`topic`*

where *topic* is a word describing what you want to know. For example, if you can't remember the commands to run Fortran or use its libraries, you might type

`apropos fortran`

which would produce a list of man pages that contain ``fortran'' in their header lines:

```
imsl              imsl (7)            - Fortran Subroutine Libraries for
Numerical Computation
asa               asa (1)             - convert FORTRAN carriage-control
output to printable form
asa               asa (1)             - convert FORTRAN carriage-control
output to printable form
f77               f77 (1)             - FORTRAN compiler
f77_floatingpoint f77_floatingpoint (3f) - FORTRAN IEEE floating-point
definitions
fpr               fpr (1)             - convert FORTRAN carriage-control
output to printable form
fputc             putc (3f)           - write a character to a FORTRAN
logical unit
fsplit            fsplit (1)          - split a multi-routine FORTRAN file
into individual files
intro             intro (1)           - introduction to FORTRAN Manual Pages
intro             intro (3f)          - introduction to FORTRAN library
functions and subroutines
libm_double       libm_double (3f)    - FORTRAN access to double precision
libm functions and subroutines
libm_quadruple    libm_quadruple (3f) - FORTRAN access to quadruple-precision
libm functions (SPARC only)
libm_single       libm_single (3f)    - FORTRAN access to single-precision
libm functions and subroutines
putc              putc (3f)           - write a character to a FORTRAN
logical unit
ratfor            ratfor (1)          - rational FORTRAN dialect
tclose            topen (3f)          - FORTRAN tape I/O
topen             topen (3f)          - FORTRAN tape I/O
tread             topen (3f)          - FORTRAN tape I/O
trewin            topen (3f)          - FORTRAN tape I/O
tskipf            topen (3f)          - FORTRAN tape I/O
tstate            topen (3f)          - FORTRAN tape I/O
twrite            topen (3f)          - FORTRAN tape I/O
```

Notice that **apropos** does not require an exact case match: it successfully found uppercase FORTRAN and mixed-case Fortran. It will also find partial words, such as ``fortr''. The numbers in parentheses identify the section number for each man page. From this list you could choose the one you need.

Getting Specific Information

Once you know the names of the man pages you need, display them on the screen by giving the **man** command:

man [*section***]** *name* (on Solaris systems, **man [-s** *section***]** *name*)

If you omit the *section* number, the man program searches through each section in turn until it finds the named man page. This is fine if *name* is unique. But a name may exist in more than one section, in which case omitting the section number would get you only the first man page.
Most manual sections have an ``intro'' man page, describing that section in general. So to find out more about library functions (section 3 of the man pages), type

man 3 intro (on Solaris systems, **man -s 3 intro**)

When **man** displays information on your terminal, it pauses after each screenful, allowing you to read the current screenful before you go to the next. To see the next screenful, press the space bar. To see just one more line, press **Return**. To quit reading, type **q.**
To print the man page instead of viewing it at your terminal, type

man *name* **| ul -t dumb | lpr -P***site*

where *site* is the location of the printer where the man page is to be printed. The vertical bar is called a ``pipe'' and is used to pass results of one command on to another command (see chapter 4). Here, the output from **man** is passed to **ul,** which controls the underlining method (for a ``dumb'' output device in this case), and that output is passed on to the **lpr** command, which sends the output to a printer. To see a list of printing sites, type

man sites

InfoExplorer on ADS

If you use the Academic Data Server, which runs the AIX version of UNIX, you can use InfoExplorer to retrieve online information. InfoExplorer has interfaces for plain text (ASCII) and for the X Window System. For more information, type

info -h

World Wide Web

In general, UNIX man pages are not written with the novice in mind, and you may not find the answer to your question by using only **apropos** and **man**. Fortunately, other tools exist for finding information online. The Web is a useful source of online information, not only about UNIX, but about a wealth of topics, from sources all over the world. Using your Web browser, connect to the ACITS home page at

www.utexas.edu/cc/

From there, you have access to various forms of documentation, FAQs, and system resources. For example, under ``Publications'', you can search a set of UNIX frequently asked questions (FAQs) to find answers to your questions, and under ``Systems'', you will find a large collection of information about UNIX--or go directly to

www.utexas.edu/cc/unix/

4. The Shell and Command Processing

The UNIX program that interprets your commands is called the ``shell''. The shell examines each command line it receives from your terminal or workstation, expands the command by substituting actual values for special characters, and then either executes the command itself or calls another program to do so. After the command has completed execution, the shell prompts you for a new command.

Most UNIX systems offer a choice of shells--the Bourne shell (sh) and the C shell (csh) are the most common. The default shell for ACITS UNIX systems, and the one described in this manual, is the C shell. Since this manual describes the C shell, you should make sure that is the one you are using. To do so, type

```
echo $SHELL
```

If the C shell is reading your commands, the response will be

```
/bin/csh
```

(The Bourne shell is /bin/sh.) The following sections discuss the shell concepts most needed by novice users.

Command Syntax

UNIX commands begin with a command name, often an abbreviation of the command's action (such as **cp** for ``copy'' or **mv** for ``move''). Most commands include ``flags'' and ``arguments''. A flag identifies some optional capability and begins with a hyphen. An argument is usually the name of a file, such as one to be read. For example, the command line

```
cat -n stuff
```

calls the **cat** program (to ``concatenate'' files). In this case, **cat** reads the file named ``stuff'' and displays it. The **-n** flag tells **cat** to number the lines in the display.

The hyphen that precedes a flag is a special character used to distinguish flags from filenames. In the example above, the hyphen prevents **cat** from trying to display a file named ``n''. Commands can contain other special characters as well. The shell interprets such characters, and sometimes replaces them with other values, before it passes the command with its flags and arguments to the program that actually executes the command.

Remember that uppercase and lowercase letters are not interchangeable. Thus if you tried to give the **cat** command by typing CAT, it would fail; there is no program named (uppercase) CAT. Or notice in the **echo** command shown above that SHELL (and only SHELL) must be all uppercase.

Special Characters

The following sections describe many of the characters that have special meaning to the C shell. Keep in mind that MOST punctuation characters have some special meaning. Therefore, you should not include any punctuation character other than period or underscore in the names you give to files and directories, or you may get surprising results. Also, don't try to use spaces in file and directory names.

Input/Output Redirection (the >, >> , and <

Characters)

If a program normally reads its input from the terminal (referred to as ``standard input'' or ``stdin'') or writes its output to the terminal (``standard output'' or ``stdout''), you may want it to read from or write to an alternate file instead. For example, if you give the command

```
who
```

the login names of others currently logged in are written to standard output--displayed at your terminal. If you want to have this list placed into a file called ``namelist'', you could use the character to redirect standard output to namelist, like this:

```
who  > namelist
```

If you already have a file named namelist, the shell erases its contents, then calls **who** to write the new information to namelist. If you do not have a file named namelist, the shell creates a new one before it calls the **who** program.
You can then do what you want with the file namelist--print it, sort its contents, display it with the **cat** command, or whatever.
To append output to an existing file instead of overwriting it, use the symbol >> . Thus, if you first type

```
who > namelist
```

and later

```
who >> namelist
```

The file namelist will contain two sets of results: the second set is appended to the first instead of destroying it.
Normally, you will not need to redirect standard input, as most commands that can read from standard input also accept an input filename as an argument. However, for commands that normally expect input from your terminal, the character

```
mail user
```

and then to type the text of the message from your terminal. If the text of the message is already in a file named letter, you could redirect standard input to send the mail this way:

```
mail user < letter
```

> **Note:** The shell performs input and output redirection BEFORE it calls the command you want to execute. This means that files can be accidentally destroyed. For example, the command
>
> ```
> sort < myfile  > myfile
> ```
>
> destroys ``myfile'' because the shell (1) opens myfile for reading and then (2) opens it again for writing. When you open a file for writing, UNIX destroys the old contents of the file. All this happens BEFORE the shell runs the **sort** program to actually read the file.

Pipes (the | Character)

A pipeline is a useful way to use the output from one command as input to another without creating an intermediate file. Suppose that you want to use the **who** command to see who is logged in, and you want to see the results sorted alphabetically. The **sort** program reads a file and displays the sorted results on your terminal--the standard output. So you could accomplish what you want with the following sequence of commands:

```
who > namelist
sort namelist
```

Afterward, you could give another command to get rid of the file namelist.
A pipeline enables you to do all this on a single command line, using the pipe symbol | (vertical bar). When commands are separated by the | symbol, output from the command on the symbol's left side becomes input to the one on the right of it. Thus you could type

```
who | sort
```

so that the results of the **who** program are passed to the **sort** program as input and displayed, by default, on your terminal.
More than one pipe symbol can be used to make a series of commands, the standard output from each becoming the standard input to the next.

> **Note:** If a command is not capable of reading from standard input, it cannot be placed to the right of a pipe symbol.

Characters Used to Expand File and Directory Names (the * ? [ ] - and ~ Characters)

The shell also interprets other special characters it finds on a command line before it passes this line to the program that will execute it. These characters are normally used in place of filenames or directory names:

*****

An asterisk matches any number of characters in a filename, including none. Thus, the command

```
cat a*
```

would display the file named ``a'', if it exists, as well as any file whose name begins with ``a''. An asterisk will not match a leading period in a file name, but it will match a period in any other position.

?

The question mark matches any single character. Thus

```
cat ? F?
```

would display all files with single-character names, plus all files that have a two-character name beginning with F. Like the asterisk, a question mark does not match a leading period, but does match a period in any other position.

[ ]

Brackets enclose a set of characters, any one of which may match a single character at that position. For example,

```
cat draft[125]
```

displays files draft1, draft2, and draft5 if they exist. Remember that the shell interprets special characters BEFORE it calls the **cat** program. So there is a difference between

```
cat draft[125]
```

and

```
cat draft1 draft2 draft5
```

In the first form, the shell considers it a match if ANY of draft1, draft2, or draft5 exist. It builds a **cat** command that includes only the names of files it finds. If there is no draft2, for example,

```
cat draft[125]
```

displays draft1 and draft5. However, explicitly giving the command

```
cat draft1 draft2 draft5
```

displays draft1 and draft 5, but also gives an error message from the **cat** program: ``draft2: no such file or directory''. If no files begin with ``draft'', then the command

```
cat draft[125]
```

produces a message from the C shell: ``no match.'' In this case, the shell doesn't even call the **cat** command.

-

A hyphen used within **[ ]** denotes a range of characters. For example,

```
cat draft[1-9]
```

has the same meaning as

```
cat draft[123456789]
```

Again, the shell expands this to build a **cat** command that includes only the filenames that actually exist.

~

A tilde at the beginning of a word expands to the name of your home directory (the directory in which you are placed when you log in on UNIX). It is a useful special character if you changed to some different directory after you logged in, and want to copy a file to or from your home directory. If you append another user's login name to the ~ character, it refers to that user's home directory--useful if you are sharing files with someone else. For example,

```
~cssmith
```

means ``the home directory of the user whose login name is cssmith''. The usefulness of this notation will become more apparent in the next chapter.

> **Note:** On most UNIX systems, the ~ character has this special meaning to the C shell only, not to the Bourne shell. In shell scripts, which are generally processed by the

Bourne shell, use the variable $HOME to specify your own home directory.

Action Characters

As already noted in chapter 2, a **^d** signals the end of input. If a program is reading data from standard input (your terminal), it reads everything you type until you use **^d**. This is also true if the program is the shell. To make the C shell ignore **^d**, so that you don't log out accidentally, disable it with the command

```
set ignoreeof
```

Another special control character for the C shell is **^z**. Whenever you type **^z**, it suspends the current program. UNIX responds with the message

```
stopped
```

and gives you a new shell prompt. You can later resume the suspended program by giving the **fg** (foreground) command, or resume it in the background with the **bg** command. To kill a suspended program, give the command **jobs -l** to get the program's job number. Then use

```
kill %job-no
```

to terminate it.

If you fail to kill or resume a suspended process, when you try to log out you will get the message: ``There are suspended jobs". You can disable **^z** entirely under UNIX by making the ``suspend" character undefined. To do so, type the following command or place it in your .login file:

```
stty susp ^-
```

You could also use the **stty** command to redefine the suspend character to something other than **^z**.

Quotation Characters (\ and ')

Although you should not use special characters in filenames and directory names, you may need to include these characters in command lines without having the shell treat them as ``special". To make the shell's special characters be treated as ordinary characters, do one of the following:

19. prefix a single character with a backslash \
20. enclose a sequence of characters in apostrophes ' '

For example, suppose you wanted to use a question mark as your C shell prompt. If you give the command

```
set prompt=?
```

the shell will expand the question mark to be the first single-character filename it finds in your directory, and that filename will be your prompt. To avoid that problem, use

```
set prompt=\?
```

History Substitution Characters (the ! ^ and $ Characters)

If you use the command

```
set history=n
```

then the C shell keeps a record of your most recent *n* events, where an event is one command line. (That is, an event might include several commands in a pipeline, or several commands that you have typed on a single line.) You can use special characters to reissue the commands without retyping them. Here are some simple ways to do this:

!!

On a line by itself, simply reissues the most recent event.

!*cmd*

Reissues the most recent event that started with the command *cmd*.

!?*string*

Reissues the most recent event that contained *string*.

!-*n*
Reissues the nth previous event. For example, **!-1** Reissues the immediately preceding event, and **!-2** reissues the one before that.
!*n*
Reissues command line *n*. To see how previous commands are numbered, just type:

```
history
```

It will display as many lines as you told **set history** to keep.
^*old*^*new*
Substitutes the string *new* for the first occurrence of the string *old* in the most recent event, and reissues that command line. **Note:** This is the real caret character, not a representation of the Control key.
For example, suppose you want to copy a file from one directory to another, using the **cp** command, and issued the command

```
cl my_old_directory/the_file_to_copy my_new_directory/the_new_filename
```

Then you got the message "cl: Command not found". You can easily correct your mistake by simply typing

```
^l^p
```

:
Selects specific words from an event line, so that you can reissue parts of a command. The : also separates these words from actions to take on them, such as a substitute command to correct a misspelled word. For example,

```
!15:s/cot/cat/
```

would reissue event number 15, substituting the string ``cat'' for the string ``cot''. See how the **:** is used with **$** below.
$
The words on each event line are numbered, beginning with 0. The last word can be designated as $. So for the event

```
who | sort | lpr -Pfacsmf_lw
```

the word ``who'' is word 0, the first vertical bar is word 1 and so on. The word ``-Pfacsmf_lw'' is word $. This command creates a list of logged in users, sorts the list into alphabetical order, and prints it at the SMF output site in FAC 212. Suppose that you also want to print this at the ACWL output site in FAC 29. The new command looks almost the same, except that it must end with acwl_lw instead of facsmf_lw. One way to reissue the original command is

```
!!:0-4 -Pacwl_lw
```

This takes the first 5 words (0-4) of the previous command, followed by the new argument to that command (-Pacwl_lw).
There are other special characters for history substitutions, but these are the basics.
Examples:

```
!f77
```

This reissues the most recent **f77** command--say, after you have corrected a source-file error that caused the previous attempt to abort.

```
!!:0-2 !-2:$
```

This creates a new command from parts of two that have already been issued: words 0, 1 and 2 from the most recent event, followed by the last word from the event before that.

The Command Separator Character (;)

To put more than one command on a line, separate complete commands with a semicolon:

```
cd ~colleague;ls
```

The Background Character (&)

To place a slow-running job in the ``background'' so that you don't have to wait for it to finish before issuing another command, use the ampersand character. For example:

```
sort verylargefile &
```

The shell will notify you when the background job is finished.

Other Special Characters

Besides the special characters discussed in this chapter, there are some others that have special meaning to the shell and which, if not quoted in a command line, will not be treated as ordinary characters. They include:

```
`    {    }    #    "
```

Within a program, you should not need to quote special characters to make them ordinary; it is only when the shell interprets your command that such characters are expanded instead of being treated as text.

Shell Initialization Files

As noted in chapter 2, every time you log in under the C shell, it executes any commands that you have in a file named .login. But logging in is not the only way to start up the C shell. The command **csh** starts a new copy of the C shell for you if you are already logged in. Some programs, such as editors, allow you to ``escape" temporarily to the shell and then resume your editing. In this case, you also start up a new copy of the shell instead of going back to the shell that called the editor. Because you are not logging in, this C shell ignores the file .login and executes commands it finds in the file .cshrc. In fact, when you log in, the C shell also executes commands in .cshrc--before it executes those in .login.

If you find that you occasionally need to start new copies of the C shell, be sure to use the file .cshrc for commands you want executed every time you start a new C shell. In general, environment variables specified by a **setenv** command (such as your **setenv EDITOR** command) should be in .login. Any **set** commands (such as **set history=40**) should be in .cshrc so every new copy of the C shell will be able to use them.

You can also put **alias** commands in .cshrc. For example, if you want to type just **h** instead of **history** to display your recently issued commands, put this line in your .cshrc file:

```
alias h history
```

Keep in mind that any commands you put into your .cshrc file will not become effective until the next time you start a new copy of the C shell. To make the commands effective immediately, type

```
source .cshrc
```

This will execute every command within .cshrc.

If you do not have a .cshrc file, you can copy the file /usr/local/lib/Cshrc instead of making a file from scratch. See chapter 5 to learn how to do this.

Note: The Bourne shell uses just one initialization file--named .profile.

Search Paths

Remember that the shell is itself a program. After the shell examines and expands each of your command lines, it determines if each command is one of its own built-in commands, which it can execute directly. The commands **history, set**, and **setenv** are examples of the C shell's built-in commands. Most built-in commands do not have individual man pages. Instead, they are described within the **csh** man page (on the ADS UNIX system, use InfoExplorer instead-- see chapter 3).

However, if the command is not a built-in command, the shell must call an external program to execute it. But such programs are not all stored in one place: they are organized into hierarchies of directories. To determine where to look for the needed program, the shell examines a variable called PATH. Normally, the path tells the shell to look first in one or more system directories in some particular order, then in your own current directory.

To see what your current path is, give the command:

```
echo $PATH
```

You will see a sequence of pathnames, separated by colons. The first pathname on the list is where the shell looks first if the command you gave is not a built-in command; the second pathname is where it looks next if the first path fails, and so on. A null path--a colon alone at the end or the beginning of the list, or a pair of colons-- means your current directory. A period, or dot, as a pathname also means your current directory.

You may never have to do anything to create or change the path variable. But if the shell cannot find a command you want to issue, the reason might be that the command is outside your search path. Also realize that if you give one of your own executable files the same name as a built-in command, your own command might never be executed. The shell will use its own code instead, if its pathname precedes your own pathname.

The next chapter discusses paths and pathnames in more detail.

5. Directories and Files

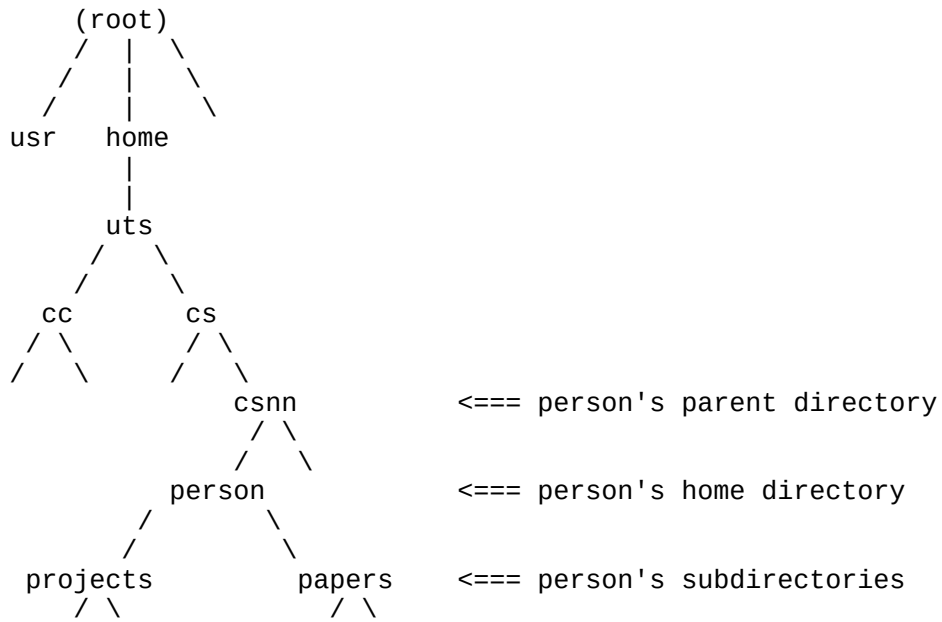
Directory Hierarchy and Pathnames

When you log in to UNIX, you are placed in a directory that contains your files. This directory is part of a hierarchy of directories for the entire UNIX system. Your login directory, known as your ``home directory", is on one path down that hierarchy from the top (the root). Each user's home directory has its own path from the root. You can create subdirectories below your own home directory. And you may need to copy files to or from another user's directory. If so, you must know how to specify pathnames of these directories in order to get to the files in them. (Microcomputer users might compare this UNIX directory structure to a hierarchy of nested folders and files on Macintosh or Windows computers.)

Pathnames can be either ``relative" or ``absolute". An absolute pathname begins with a slash (/) and starts at the root. Each successive directory name down the path is also preceded by a slash. When you examined the shell's search path with the **echo $PATH** command in the previous chapter, you saw some absolute pathnames. The absolute pathname to the home directory for user ``person" might look like this:

```
/home/uts/cs/csnn/person
```

The directory ``person" contains the names of files in that directory, and it may also identify other directories below it in the hierarchy. ``Person" is itself under the directory csnn, which is under the directory cs. That is, the csnn directory is the ``parent" of person; subdirectories below person are the ``children" of person. This hierarchy is (in part):



A relative path starts from the directory you are working in and moves downward to a lower directory. Instead of starting with a slash, a relative pathname starts with the name of the first directory below this working directory; each lower directory down the path is prefixed by a slash.

In both absolute and relative pathnames, the rightmost component is either the ultimate directory name, or a file in that directory. For example, suppose ``person" has two subdirectories: ``projects" and ``papers", and ``report1" is a file in ``papers". If ``person" is the working directory, then here is a command to display ``report1":

```
cat papers/report1
```

Another way to display the same file would use an absolute pathname:

```
cat /home/uts/cs/csnn/person/papers/report1
```

Remember that the C shell interprets ~person to mean ``the home directory of the user whose login name is person". Thus, an equivalent command under the C shell would be simply

```
cat ~person/papers/report1
```

Seeing What is in Your Directory

To see the names of any files or lower-level directories you have in your current working directory, use the **ls** (list) command. Its simplest form is

```
ls
```

which lists the names of any files and directories that do not begin with a period (.). To see ALL the names, use the **-a** (all) flag:

```
ls -a
```

Even if you have not yet created any files in your directory, you should see the names ``.login" and ``.cshrc" (the C shell's initialization files) and two entries named ``." and ``..". The entry named only ``." is the directory you are working in, and ``.." identifies its parent, even though these directories have full pathnames, too. All directories use ``." to refer to their own names and ``.." to identify their parents.

To see how big your files are, and to see their protection modes, use the **-l** (long) flag:

```
ls -l
```

The protection modes determine what actions you and others may perform on the files. What these modes mean and how to change them is described below.

The **-F** flag is also useful. It marks the names of directories with a trailing slash (/) character and the names of executable files with a trailing asterisk (*). In fact, the command:

```
alias ls 'ls -F'
```

is often included in **.login** and **.cshrc** files so that directories and executables are always indicated by the trailing characters.

To see directory information about selected files and lower-level directories, you can use the special characters described in chapter 4. For example, suppose you are ``person" in the hierarchy shown above, working in your home directory, ~person. If you give the command

```
ls p*
```

this expands to

```
ls papers projects
```

and lists the names of files in the subdirectory ``papers", then those in the subdirectory ``projects", except for files whose names start with a period.

To list the names of all files in the lower directory ``papers", you might use

```
ls -a papers
```

and get something like

```
.  ..  abstract  outline  report1  report2  report3
```

Note that this directory, like your home directory, uses . to mean its own name and .. to identify its parent. So to list the names of files in the directory above the current one, you could type

```
ls ..
```

If you only need to know which files in subdirectory ``papers" have names beginning with ``report", you could use

```
ls papers/report*  or  ls papers/report[1-9]  or  ls papers/report?
```

The last two forms would, of course, not list a file named ``report12".

Note from these examples that the **ls** command lists files in the working directory only, unless you include the pathname to another directory whose filenames you want to list.

Naming Files and Directories

Whenever you make new directories below your home directory and create files in any of your directories, you must give these directories and files appropriate names. In general, names should include only letters of the alphabet, digits, period (.) and underscore (_). As shown above in the description of the **ls** command, files whose names begin with period do not show in a directory listing unless you use the **-a** flag. It is customary for initialization files to begin with a period

(like **.login** or **.cshrc**). The underscore character is useful for readability in filenames (e.g. test_scores). Most other punctuation characters have special meaning to the shell. Avoid them in filenames or directory names.
A common use of period is to separate a filename into two segments--a base and a suffix. Related files might be those with a common base name and different suffixes, such as

```
fortprog.f   fortprog.o
```

or with different base names and matching suffixes, such as

```
prog.c   newsource.c   test.c
```

Using the period this way, you can select sets of files for particular operations. For example, you could use ***.c** to select all files whose suffix is ``c''.
Certain programs such as compilers use filename suffixes to determine what action to take when they process files. Examples of this are in chapter 8.
Finally, no file or directory name can exceed 255 characters and no complete pathname can exceed 1024 characters.

Making New Directories and Moving Between Directories

If you have many projects and you want to separate one project's files from another's, you can create a separate subdirectory to contain each file set. Such subdirectories are below your home directory in the hierarchy, and you can make additional subdirectories below these if needed. If you are working in your home directory, you can make a new subdirectory immediately below it by using the **mkdir** command:

mkdir *name*

where *name* is the name of your new subdirectory. Suppose you call it ``to_do''. To make another subdirectory, named ``july'', below ``to_do'', you could use

```
mkdir to_do/july
```

specifying the relative pathname from your home directory to the new subdirectory you are calling ``july''. Or you could move from your home directory to directory to_do first, then make july from there. By moving to subdirectory to_do, you make it your ``working directory''. To move to to_do from your home directory, use the **cd** (change directory) command:

```
cd to_do
```

Now you can create the directory july below it by typing just

```
mkdir july
```

This concept of ``working directory'' is very important. Whatever directory you are currently in is your working directory. Any time you use a filename with no pathname preceding it, that file is assumed to be in your working directory. Thus, if you start from your home directory and type

```
cd to_do/july      cat  project1
```

then **cat** looks for file project1 ONLY in the subdirectory july, the new working directory. It does not look in your home directory or in subdirectory to_do. Likewise, if ``.'' is in the shell's search path, then the shell looks in your working directory for commands it cannot execute directly. This means that where the shell looks depends on where you are when you issue commands.
When you give the **cd** command with no arguments, it makes your home directory your working directory.
If you use **cd** very often to change working directories, you may need to be reminded where you are. To display the absolute pathname of your working directory, give the command

```
pwd
```

which means ``print working directory''.

Moving and Renaming Files and Directories

To move files from one directory to another, use the **mv** (move) command:

mv *fromname toname*

Include the needed path information about the ``from'' or ``to'' directory if it is not your working directory.

For example, suppose you want to move file source.a from your home directory to the next lower directory, Mysources. If Mysources is your working directory, give one of these commands:

mv ~/source.a source.a or **mv ../source.a source.a**

If your home directory is the working directory, do it this way:

mv source.a Mysources/source.a

The **mv** command can also be used to rename files and directories. For example,

mv xyz abc

simply renames file ``xyz'' to ``abc'' in your working directory, since no directory change is involved. Similarly,

mv Newstuff Oldstuff

renames the directory ``Newstuff'' to ``Oldstuff'', without affecting the names or contents of the files in ``Newstuff''.

To prevent destroying a file that already exists, include the **-i** flag. For example, if you type

mv -i source.a oldsource.a

and ``oldsource.a'' already exists, **mv** will ask if you really want to overwrite that old copy with the newer one.

Protecting Files and Directories

Your files and directories are protected from other users' perusal and destruction by protection-mode settings. These settings control who may read, write, and execute your files. To see what the settings are, issue the **ls** command using the **-l** (long) flag. To the left of each file or directory name is displayed a sequence of 10 characters, such as

```
drwx------ or      -rw-r--r--
```

The leftmost character is ``-'' for a filename entry, ``d'' for a directory name entry. The next 9 characters are three triplets of protection settings; the first pertains to you (the owner), the middle triplet pertains to members of your group (if a group has been established), and the rightmost pertains to everyone else.

Thus the first line shown here says the entry is a directory for which you have read (r), write (w), and execute (x) permission and for which everyone else has no privileges at all. The second entry is a file. In this case, you may read and write it, but others can only read it.

To change the protection modes of your files or directories, use the **chmod** (change mode) command. It has several forms, but the easiest to use is

chmod *[who]oppermission names*

where *names* identifies one or more file or directory names. The *whooppermission* sequence must be given with no blanks between portions. The *who* portion can be any combination of

u user (owner of the file or directory)
g group members
o others
a all of the above (default if you omit the who argument)

The *op* is one of

```
+  add the designated permission
-  remove the designated permission
=  replace all existing permissions for the relevant who with
```

the new ones of this chmod command

and *permission* is any or all of

```
r  read:  If chmod is for a file, it allows that file to be
   read, provided the directory containing it has x permission.
   If chmod is for a directory, it allows that directory to be
```

```
   listed with filenames only.

w  write:  If chmod is for a file, it allows that file to be
   written (altered or replaced).  If chmod is for a directory,
   it allows new files to be added to that directory and existing
   files to be removed.

x  execute:  If chmod is for a file, it makes that file executable
   (see chapter 10).  If chmod is for a directory, it allows a
   detailed listing (if r is also set), and allows files to be read
   or written in that directory according to their individual file
   permission settings.
```

The usual protection for a directory that you want others to be able to look through is r-x for everybody.
Examples:

chmod u=rx *.prog

permits the owner of all files whose suffix is ``prog'', in the working directory, to read and execute these files. If write privileges had existed for the owner, these privileges are removed. No one other than the owner is affected, and no protection settings of any other files are affected.

chmod +r abstract

gives everybody permission to read file ``abstract''. No other existing permissions are altered for the file. The directory containing ``abstract'' must have x permission as well, if attempts to read ``abstract'' are to succeed.

chmod +rx .

gives everyone permission to read and search the working directory.

> **Note:** If you need to establish a ``group'', the computer account sponsor should arrange this by sending e-mail to
>
> **remark@cc.utexas.edu**
>
> Include in the message the name wanted for the group and a list of the users to be in the group. Call the Help Desk at 475-9400 for more information.

Displaying Files in Your Directory

We have already discussed the **cat** command, which concatenates one or more files to standard output, your terminal. Its format is simply

cat *file1 [file2] ...*

To display files that take up several screens, you may want to use the command

more *file1 [file2] ...*

The **more** program pauses after each screenful. At the bottom of the screen, it tells you how much of the file has been displayed so far. To see the next screenful, press the space bar. To see just one more line, press **Return**. For information about other options, type **h**. To quit without seeing the rest, press **q**.
If you just want to see the beginning of a file, use the **head** program:

head *[-n] file*

which displays only the first n lines of the file. The default for *n* is 10. The **tail** command shows the last *n* lines:

tail *[-n] file*

Printing Files

To get a printed copy of a file, use the **lpr** command:

lpr -P*site file*

where *site* identifies the printer where you want to pick up your output. To see the names and locations of sites, type

man sites

At each output location, output is filed according to the last three digits of the ACITS user number.

Making Copies of Files

To make a duplicate of a file's contents, use the **cp** (copy) command:

cp *fromfile tofile* or **cp** *file directory*

The second form copies a file to a new directory, keeping its original filename. The *fromfile* and *tofile* should include appropriate pathnames whenever the files are not in your working directory. Be sure that they are two different files.

Examples:

```
cp myfile Backup/myfile.old
```

copies file ``myfile'' to the next lower directory, named ``Backup'', as a file named ``myfile.old''.

```
cp ~csaa123/report report
```

makes a copy of file ``report'' from the home directory of user csaa123 and places it, with the same name, in your working directory. Since you are not changing the file's name, just copying it to the working directory, you could also have typed

```
cp ~csaa123/report .
```

Removing Files and Directories

To get rid of files you no longer want, use the **rm** (remove) command:

rm *file1 [file2] ...*

If you do not include a pathname, the files are assumed to be in your working directory. If you use any special characters such as * or ? with the **rm** command, it is a good precaution to add the **-i** flag, like this:

```
rm -i file*
```

This interactive option displays, one at a time, each candidate for removal. To remove the file, type **y** (for ``yes'') and press **Return**. Any other response keeps the file.

Once **rm** has removed a file, that file cannot be retrieved: there is no ``undelete'' command. So you might want to put this command in your .cshrc file:

```
alias rm 'rm -i'
```

This will cause **rm** to always use the **-i** flag, even if you type **rm** alone.

If you have a filename that contains one of the shell's special characters, **rm** might not be able to delete it. (A common case is accidentally creating a filename whose first character is a hyphen.) In this case, use the **-** flag and put the filename in quotation marks, like this:

```
rm - "badname"
```

To remove a directory that you no longer need, use **rmdir:**

rmdir *path*

where *path* is the relative or absolute pathname of the directory. The **rmdir** command will not delete a directory while it is your working directory, nor will it delete a directory that contains any files (besides **.** and **..**).

To remove a directory AND any files and subdirectories it contains, as well as files in those subdirectories, use the ``recursive'' switch with the **rm** command:

rm -r *path*

> **Be Careful**--Without also using the **-i** switch, it's possible to quickly delete entire directory structures. To be safe, use
>
> **rm -ir** *path*

Altering Search Paths

Remember that the shell searches certain directories, in a specific order, to find commands that it cannot execute itself. If you put all of your executable files into a particular directory, you must alter the shell's search path so it will examine this directory. Suppose, for example, that you put all your executable files into the subdirectory ``actions'' just below your home directory. To add this subdirectory to the C shell's search path, give the command:

```
setenv PATH ${PATH}:$HOME/actions
```

Here ${PATH} means the current value of your search path and $HOME means the name of your home directory. In this case, you have made $HOME/actions the final place the C shell will search. To make the C shell look there first, reverse the positions of $path and $HOME/actions. Put the **setenv** command in your **.cshrc** file to make it take effect every time you log in or start a new copy of the C shell.

You may also need to set the PATH variable if you want to use a program outside your own directories and also outside the default search path. For example, a collection of programs to convert graphics files from one format to another is in the directory /usr/local/pbm/bin. To call up any one of these programs without having to give its full pathname, you could add /usr/local/pbm/bin to your search path.

Note: Much of the optional software commonly used on UNIX systems is installed in

```
/usr/local/bin      or      /usr/local/gnu/bin
```

Disk Quotas

When your user number is validated for one of the ACITS UNIX systems, you are assigned quotas limiting the amount of disk space you may use. For most accounts, each user is normally assigned a ``soft" quota of 1000 kilobytes and a ``hard" quota of 1250 kilobytes. (A kilobyte is actually 1024 bytes, or characters.) If you exceed your ``soft" quota, you receive a warning message, but you can continue running programs and can log out without losing any files. However, there are two reasons that you should take prompt action to remove unneeded files to get below the soft quota.

The most serious reason is that if you continue using resources, you may reach the ``hard" quota. If you do, you will not be able to run programs. In fact, a running program will abort if it attempts to use disk space that exceeds the hard quota. Furthermore, when you try to modify existing files, they may get truncated to zero-length files--without warning or notice.

The second reason is that even if you do stay below the hard quota but remain over the soft quota for seven days (on the CCWF and UTS systems), you will not be able to alter existing files or create new ones until you get back below the soft quota.

To see how much disk space you are using, give the command

```
quota -v
```

If you need a higher disk quota than the one originally established, you can change it by giving the command

```
chquota
```

The quota you request must be in multiples of 1000 kilobytes, with a minimum of 1000 kilobytes (approximately 1 megabyte). Keep in mind that you are charged for the disk space you reserve (i.e., your quota), rather than the disk you actually use; so don't request too much more than you expect to need.

Kapitel 6 och 7

Utgår i DOK00

8. Running Programs on UNIX

This chapter describes the general procedures needed to compile, load, link, and execute programs written in Fortran 77, C, and Pascal. To be able to use these compilers on the CCWF system, you must include in your search path the two directories

```
/usr/ccs/bin      /opt/SUNWspro/bin
```

To run programs written in other languages, see the appropriate man pages.

Fortran 77 Programs

The Fortran 77 compiler on UNIX is f77, which is called with the command

f77 *[options] file ...*

where *file* is the name of your Fortran source file. If the last two characters of the source filename are ``.f" or ``.F", the file is treated as a Fortran 77 program. Source files with the ``.F" suffix are passed through the C language preprocessor before f77 compiles them.

In the simplest form of the **f77** command, when no options are specified, the source file is compiled and the object program is written to a file with the same name but with the ``.F", or ``.f" suffix replaced by ``.o". This object file is then loaded into an executable file named a.out. If you have specified more than one filename, all of the ``.o" files are loaded, in order, into a.out. To execute the program, simply type

a.out

as a command.

Here are some of the most useful **f77** command options:

-c

Suppresses loading into a.out and merely compiles--that is, creates ``.o" files. (Normally, the ``.o" files would be deleted after a.out is created.) These ``.o" files can be used as arguments to subsequent **f77** commands.

-g

Generates a special symbol table that can be used by debugging programs (described in chapter 8).

-o *outfile*

Names your executable file *outfile* instead of a.out. Note that whenever a compiler makes a new file a.out, it destroys any previous one.

-U

Prevents uppercase characters from being converted to lowercase ones.

For a complete list of options, see the **f77** man page. (On ADS, see the **xlf** man page or simply type **xlf** with no command arguments.)

C Programs

The C compiler on UNIX is cc, which is called with the command

cc *[options] file ...*

Like f77, cc gives special interpretations to filename suffixes. It treats filenames ending in ``.c" as C source programs. When it compiles these programs, it writes object programs to files that have the same name but with the ``.c" suffix replaced by ``.o". Unless options on the **cc** command prevent it, the ``.o" file is loaded into an executable file named a.out and the ``.o" file is then removed. To execute the resulting program, simply type

a.out

as a command.

Here are some of the most useful options to the **cc** command:

-c

Suppresses loading and compiles only: it merely creates ``.o" files (which may be used as input to a subsequent **cc** command).

-g

Generates a special symbol table that can be used by debugging programs (see chapter 8).

-o *outfile*

Names the resulting executable file *outfile* instead of a.out, thus preventing you from overwriting any existing file named a.out.

For a complete list of options, see the man page for the **cc** command. (On ADS, see the **xlc** man page or simply type **cc** with no command arguments.)

Pascal Programs

The Pascal compiler on UNIX is pc, which is called with the command

```
pc [options] file ...
```

On ADS systems, use

```
xlp [options]  file  ...
```

The pc compiler treats any file whose final two characters are ``.p" as a Pascal source program. When Pascal compiles a source program, which may exist as one or several ``.p" files, it writes object code to files with the same name but with ``.p" replaced by ``.o". These object files are then loaded, in order, into an executable file whose name is a.out. To execute the program, simply type

```
a.out
```

as a command.

Here are some useful **pc** command options:

-c

Suppresses loading of executable file a.out and compiles only, producing ``.o" files, which can be used as input to a later **pc** command.

-g

Generates output to be used by debugging programs (see chapter 8).

-o *outfile*

Names the executable file *outfile* instead of a.out.

-s

(On CCWF only) Issues warning messages if any nonstandard Pascal constructs are encountered.

-std

(On UTS only) Issues warning messages if any nonstandard Pascal constructs are encountered.

For a complete list of options, see the **pc** man page. (On ADS, see the **xlp** man page or simply type **xlp** with no command arguments.)

Getting Listings of Source Code and Error Messages

To print a listing of your source file and add to the listing page headers, page breaks, and line numbers, use the **-n** flag on the **pr** command, like this:

```
pr -n mysource.p | lpr -Pacwl_lw
```

To intersperse compiler error messages with source code, you can use the **error** program (not on AIX systems). For example, suppose you have a C program in a file named testrun.c and you compile only:

```
cc -c testrun.c
```

If there are errors, the C compiler will write its error messages to your terminal. It would be more useful to intersperse these error messages into the source code at the place where they occur, and the **error** command does this. For the sample C program (and assuming you are running the C shell), you could use

```
cc -c testrun.c |& error -q
```

The **|&** here assures that both standard output and error output from **cc** will be piped to the **error** program. The **-q** option tells the **error** program to query you before putting error messages into the file testrun.c. Its output might look like this:

```
File ``testrun.c'' has 3 errors.          3 of these errors can be
inserted into the file.      Do you want to preview the errors first?
```

At this point, if you answer **y,** you will see just the messages and be prompted again:

```
Do you want to touch file ``testrun.c''?
```

If you answer **y** again, the **error** program rewrites testrun.c, with each error message preceding the line that caused the error. You can then edit the file and try compiling it again.

For more information about how **error** handles messages, see

```
man error
```

Interactive Debugging Tools

You can use the interactive debugging program dbx to monitor and control execution of an f77, C, or Pascal program, or to examine the state of a program whose execution aborted.

If you want to use **dbx**, include the **-g** option on the **f77**, **pc**, or **cc** command that compiles your source program. This option produces symbol tables from which the debugger can locate all source files that the compiler uses to produce its object (``.o") files.

To execute the program under control of the debugger, do not give ``a.out" as a command. Instead, use **dbx** like this:

```
dbx a.out
```

to control its execution. The dbx program will give you a prompt, after which you can give commands to set and remove break points, to step through execution, and to trace execution by printing each statement before it is executed.

If an executing program aborts, it dumps a core image, usually as a file named ``core". The dbx program can examine this core file to determine and display the state of the program when it aborted.

For detailed information, see

```
man dbx
```

Tools for Maintaining Large Programs

If you write large programs that require libraries of routines, or programs that consist of many files, there are UNIX tools to help you maintain them and keep track of modifications.

The **ar** command maintains groups of files combined into a single archive file. The **ranlib** command converts archives to random libraries for easier loading by the loader. On UNIX systems running Solaris, use the **lorder** command instead.

The **make** program is useful for maintaining and updating groups of programs in parallel.

The **sccs** program provides mechanisms for tracking revisions so that you can revert to an earlier version of a program or maintain multiple versions.

For more information, see

```
man sccs
```

9. Other Useful Commands

Setting Spending Limits and Monitoring Your Charges

Each user has a default limit for expenditures (such as pages sent to a printer, or daily charges for Telesys). Each night, information about that day's computer use is transmitted to the central accounting system from every ACITS computer system. The accounting system processes this information and sends back the current status and accumulated expenditures for each user. Whenever you meet (or exceed) either the limit, the accounting system will change your status so that the next day you cannot continue to use the services. On ACITS UNIX systems, if you try to log in the day after you exceed your limit, you will see a message that describes the problem. You will not be able to run jobs, read mail or news, or print anything until a higher spending limit has been set.

To keep track of your accumulated expenditures, use the **spend** command with the **l** flag (letter ``el'' for ``list''):

```
spend -l
```

This lists your accrued charges (as of the previous night's accounting update).

Any computer account sponsor or individually funded (IF) user can use the **spend** command to set an annual ceiling on computer charges. For an IF user, these are the maximum out-of-pocket expenses that can accrue over the fiscal year (1 September through 31 August), before the user number is deactivated. If you are an IF user, use the **spend** command like this:

```
spend -s nnnn.nn
```

For example, the command

```
spend -s 50.00
```

would give you a total expenditure limit of $50.

Computer account sponsors can use a **-u** flag on the **spend** command to set limits for individual user numbers under the computer account. For example, a sponsor in the astrology department could set limits for the user STAR322 like this:

```
spend -s 40.00 -u star322
```

and set that limit for each and every user under the STAR stem like this:

```
spend -s 40.00 -u star
```

If you are neither an IF user nor a sponsor, you will not be able to change limits with the **spend** command. In this case, if you reach your expenditure limit and cannot run jobs, contact your computer account sponsor. The sponsor is the only person who can fix the problem and make your user number work again.

For complete information on the **spend** command, type

```
man spend
```

Viewing and Updating Personal Information

If you supplied information for the ``personal information block'' at the time of your first UNIX login, other users can see this information when they type

```
finger -m username
```

This information can be useful in reading and sending mail, to determine who sent you a message if the sender is identified only by login name, or to ensure that your own messages are directed to the right person.

If you did not supply information in the first login session, or if you want to add, delete, or revise information, use the **chfn** program:

```
chfn
```

It displays, one line at a time, information it already has for you. For example, its first line might be

```
Name [Pat Pending]
```

If ``Pat Pending'' is correct, just press **Return**. To change it, type the new value and press **Return**. To remove a value completely, type **none** and press **Return**. After you have responded

to all the prompts, **chfn** gives the message ``Updating user information" and exits. It may take one or two hours for the update to take effect.

> **Exceptions:** For the UTS UNIX system, you must run **chfn** on the machine named **curly.cc.utexas.edu.** For the CCWF system, send an e-mail request to change your personal information block to
>
> **remark@cc.utexas.edu**

Searching for Pattern Matches in Files

The **grep** (get regular expression) program searches in one or more files for a particular string of characters or pattern. There are three variants: **grep**, **egrep** (extended **grep**), and **fgrep** (fixed-string **grep**). Only **grep** and **fgrep** are discussed here. See the man pages for a discussion of **egrep.**

To use **grep** or **fgrep,** give the command

grep *[options] expression [file] ...* or **fgrep** *[options] [string] [file] ...*

If you give no filenames, **grep** and **fgrep** search standard input (your terminal), and so they can be used at the end of a pipeline of commands (see chapter 4). The **grep** program expands special characters in the given expression. The **fgrep** program does not expand any characters, but searches for exact matches of the character string you specify. Normally, both **grep** and **fgrep** will search the files you specify and produce a list, on standard output, of every line containing the text string matching the string or expression you specified.

The **grep** command can expand special characters to find matches in much the way the shell expands special characters in file and directory names. Below is a simplified explanation of that expansion. The term ``special character" here means one of the seven characters

\ [ ] . ^ * $

and a delimiter (such as apostrophe) used to mark the beginning and end of an expression. The delimiters are required if the expression contains any blanks or special characters, and they can be used even if the expression does not contain such characters.

Any nonspecial character matches itself. Thus

```
grep 'man page' intro.draft
```

searches through the file intro.draft for all occurrences of the expression ``man page".

A period matches any character. So

```
grep 'an.' intro.draft
```

would find ``and", ``any", ``manual", ``can", etc. in the file ``intro.draft".

If a set of characters is placed inside square brackets, each one is considered for matching in that position. Thus

```
grep '[mh]an' intro.draft
```

would match any words containing the sequence ``man" or ``han", but not ``ran", ``can" or `` and".

You can use the hyphen within square brackets to denote an inclusive range. Thus

```
grep '[a-c]r' intro.draft
```

would match strings containing ``ar", ``br" and ``cr".

The character \ turns off the special nature of the character following it, provided that \ is not within square brackets. Inside square brackets, \ is an ordinary character. So

```
grep 'manual\.' intro.draft
```

turns off the ``special" nature of the period and matches ``manual.", but not ``manual ", ``manuals", or ``manually".

The **fgrep** command does none of this expansion; however, you can search for more than one string of characters with a single **fgrep** command. To do this, place the possible strings, one per line, in a file. Then use the option **-f** *file* instead of the string option when you call **fgrep**, like this:

```
fgrep -f list intro.draft
```

If file ``list'' contains

```
manual.
manuals
manually
```

fgrep would find all lines in intro.draft that contain ``manual.'' or ``manuals'' or ``manually''.
Here are some useful options to both **grep** and **fgrep:**

```
-i    Ignores case, so that uppercase and lowercase characters
      match each other.
-n    Displays the line number with each line containing a
      match.
-l    Displays only the names of files that contain matching
      lines, but not the lines themselves.  This is useful if
      you are searching through a set of files to see which of
      them contain a particular pattern.
-v    Displays lines that don't match a given pattern.
```

Comparing Contents of Files

The **diff** program is useful in determining how the contents of two files or directories differ. The simplest way to use it is

diff *oldfile newfile*

This produces, on standard output, a list of lines that must be changed (c), appended (a), or deleted (d) to make the first file match the second. Lines from the first file are prefixed by ``<'' and lines from the second are prefixed by ``''.
If you only want to determine IF two files differ, and not HOW they differ, you can use **cmp**:

cmp *file1 file2*

The **cmp** program merely notifies you if the files don't match, stopping its comparison after it finds the first difference.

Transferring Files To and From Other Computer Systems

The FTP (File Transfer Protocol) program moves files from one computer to another. To use FTP, you call it, open a connection to the remote computer (remote host) and, within FTP, log in on that remote host. The remote host will be running its own version of FTP as well, with the local FTP (called the user interface) talking to the remote FTP (the server). Within this FTP environment, you can list the files in your remote directory, make local copies of remote files, make remote copies of local files, and delete remote files.

> **Note:** It is generally not a good idea to transfer binary executable files from one computer architecture to another. Compiled programs will almost certainly not run on the receiving system without recompilation, so you should transfer source files of programs instead.

To illustrate a simple FTP session, the following example shows user csab123 logged in on the CCWF UNIX system, using FTP to get file cprog.data from UTS and naming the CCWF copy of it cdata.sun. User csab123 has the login name jdoe on UTS. What the user types is shown in boldface. Everything else is from the local CCWF FTP or, in the case of responses that begin with a number, remote UTS FTP.

```
% ftp
  ftp> open uts.cc.utexas.edu                                    (1)
  Connected to uts.cc.utexas.edu.
  220 curly.cc.utexas.edu FTP server (OSF/1 Version 5.60) ready.    (2)
  Name (uts.cc.utexas.edu:csab123): jdoe
  331 Password required for jdoe.
  Password: xxxxxx                                               (3)
  230 User jdoe logged in.
  ftp> get cprog.data cdata.sun                                  (4)
  200 PORT command successful.
  150 Opening data connection for cprog.data (128.83.134.11,3686) (528
```

```
bytes).
  226 Transfer complete.
  local: cdata.sun remote: cprog.data
  544 bytes received in 0.025 seconds (21 Kbytes/s)
  ftp> quit
  221 Goodbye.
```

21. In the **open** command, the user gave the full name of the UTS host, but in this instance just ``uts'' would suffice.
22. Notice that the response is from ``curly'', a specific member of the UTS cluster.
23. The password is not actually displayed.
24. The **get** command has the syntax
    get *remotefile localfile*

The remote file is assumed to be in your login directory on the remote machine. The local copy will be placed in your current working directory. However, you can give a pathname as part of either *remotefile* or *localfile* to transfer between different remote directories and local directories to which you have appropriate access.
Had user csab123 wanted to **send**, instead of **get**, a file, the command would be
put *localfile remotefile*
Besides the man page, FTP has internal help that provides short explanations of all available commands. In response to the ftp prompt, type
help
Two particularly useful commands are **mget** and **mput,** which enable you to specify wildcards to get or put multiple files. The FTP program prompts you for each matching file before it transfers it. For example, suppose you have five files that end in .txt on a remote machine, in a large directory, and you want to get only two of those five files. The files are:

```
document_list.txt
file1.txt
file2.txt
myfile.txt
notitle.txt
```

That part of your FTP session might look like this:

```
ftp> mget *.txt
  mget Document_list.txt? n
  mget file1.txt? n
  mget file2.txt? y
  200 PORT command successful.
  150 Opening BINARY mode data connection for file2.txt (128.83.42.2,2784)
(553 bytes).
  226 Transfer complete.
  553 bytes received in 0.09 seconds (0.011 Kbytes/s)
  mget myfile.txt? n
  mget notitle.txt? y
  200 PORT command successful.
  150 Opening BINARY mode data connection for notitle.txt (128.83.42.2,2786)
(227 bytes).
  226 Transfer complete.
  227 bytes received in 0.065 seconds (0.015 Kbytes/s)
  ftp>
```

The "FTP Pocket Reference List", describes basic FTP commands. It is online at
www.cc.utexas.edu/cc/docs/ccrl33.html

Reading USENET News

Several thousand electronic bulletin boards, called USENET newsgroups, contain discussions of topics ranging from specific sciences to computer languages to recreational pursuits. See

www.cc.utexas.edu/cc/usenet/

for more information about USENET news.

A file in your login directory, named .newsrc, keeps track of all the newsgroup names and which messages in each group you have already seen. To read any of these newsgroups, use a news-reading program. One such program is called **trn.** The **trn** program examines your .newsrc file and displays unread articles in the same way that **more** displays a file--one screenful at a time. (Be sure your terminal type is defined correctly--see chapter 2.) When you first run **trn,** you will be ``subscribed to'' a newsgroup named news.announce.newusers, which contains useful information and guidelines for what is called ``netiquette''--common courtesy to use in posting messages.

The **trn** program has many more options than can be discussed here-- selecting articles by subject matter, subscribing and unsubscribing to newsgroups, saving articles, and searching for patterns within an article. You can also reply to the sender of an article or post a follow-up message. For a complete description, see the **trn** man page.

To post your own new message to a newsgroup, use the **Pnews** program. (Notice the capital ``P''.) It prompts you for necessary information.

Another popular news reader is called **tin.** It also uses the .newsrc file, but when you first use it, it shows you all available newsgroups--several thousand of them. You can go through the list, or search for keywords in newsgroup names, subscribing to the ones you want. For more information, see **man tin.**

Making a Record of Your Terminal Session

You can make a running log of your terminal session, recording everything you type and every response you receive. This can be especially useful in reporting some problem to the consultants, as you will have an exact record of your actions and their results. To make such a log, use the **script** command:

script *filename*

Now everything you type and everything displayed on the screen will also be copied to the file *filename.* To stop the recording, use **^d** or **exit**.

Disconnected or Runaway Jobs

If your terminal or workstation seems to lock up and you cannot log out, you might have a runaway or disconnected job. To correct this problem, first get back to the Telesys or CCNet prompt and disconnect the session (or use another terminal). (To get back to the prompt, hold down the CTRL key while you type the ^ character, then type **x**.) Then follow these steps:

1. Log in again on the same machine (if need be, from another terminal) and type the commands:

   tty **ps -u** *username*

   The first command identifies the terminal you are now logged into. The second shows you all processes you (username) own that are running on the system.
2. Look for lines that say:

   csh

   in the COMMAND column. These are the top-level login shell processes for your jobs.
3. If any of these lines where the terminal line shown in the TTY column is NOT your current terminal, look in the PID column for the process id number of that C shell. Send that C shell a hangup (HUP) signal:

   kill -HUP *pid*

   where *pid* is the process id number you just found.

4. Give the **ps -u** *username* command again to make sure the other process is gone. If the unwanted process is still there, reissue the **kill** command like this:
   kill -9 *pid*

Here is a sample session, showing the prompts (a percent sign), commands, and responses:

```
% tty                          # Determine current login terminal
/dev/tty5c                     # Its TTY number is 5c
% ps -u myname                 # Look at all processes I am running
  PID TT STAT  TIME COMMAND
27595 4a E     0:24 csh        # Shell running on another terminal
28200 5c E     0:03 csh
28275 5c R     0:01 ps xg
% kill -HUP 27595              # Send that shell a hangup signal
% ps -u myname                 # Check to make sure it worked
  PID TT STAT  TIME COMMAND
28200 5c IW    0:03 csh
28350 5c R     0:01 ps xg
```

Text Formatting

ACITS UNIX systems have **troff**, **nroff**, **TeX**, and **LaTeX** text formatters. These use embedded commands within a text file to produce formatted output. The CCWF system also has **DECwrite**, a formatter that runs under the X Window System.
Troff, TeX, and LaTeX can produce output for PostScript laser printers. TeX/LaTeX is particularly useful for text containing mathematical expressions.

> **Note:** If you use the Maple program (for symbolic computation), you can produce LaTeX output directly from that program. See **man maple** for more information.

Descriptions of these formatting languages are outside the scope of this beginner's manual. DECwrite has built-in help, including a tutorial. If you have used FRAME on some other UNIX system, or a text processor such as Microsoft Word on a microcomputer, the concepts of DECwrite will already be familiar to you.

10. Some Basics of Shell Programming

As described in chapter 4, a shell is a command language interface to the UNIX operating system. But a shell can also be used as a programming language. You might write a shell script to make a complicated sequence of commands easy to execute or even use such a script as a substitute for a program in a more conventional programming language. The Bourne shell is the one most used for shell programming and it will be described in this section. When you call a shell script from the C shell, and #!/bin/sh is the first line of the file, it is the Bourne shell that executes the script. Note carefully, then, that any shell built-in commands you use in a shell script must be those for the Bourne shell. For a description of the Bourne shell, see

```
man sh
```

This chapter does not try to teach you to write shell scripts. Its purpose is to give you a basic understanding of the Bourne shell's capabilities as a programming language.

Running a Shell Script

A shell script is simply a file containing shell commands that you can execute like this:

```
sh filename [arg1 arg2 ... argn]
```

A shell script may also be executed by name if the file containing the shell commands has read and execute permission (see chapter 5). If file ``do_it'' contains the shell commands and has such permissions, then the previous example is equivalent to:

```
do_it [arg1 arg2 ... argn]
```

In this case, executing a shell script works the same as executing a program. Remember that its first line should be #!/bin/sh to be sure the Bourne shell is the command interpreter that reads the script.

Simple Shell Scripts

The simplest shell script contains one or more complete commands. For example, if you wanted to know the number of files in your current directory, you could use

```
ls -l | wc -l
```

If you were to create a file called ``countf'' that contained this line (and with the correct read and execute permissions), you could then count the number of files simply by typing:

```
countf
```

Any number of commands can be included in a file to create shell scripts of any complexity. For more than simple scripts, though, it is usually necessary to use shell variables and to make use of special shell programming commands.

Shell Variables

Shell variables are used for storing and manipulating strings of characters. A shell variable name begins with a letter and can contain letters, digits, and underscores, such as

```
x
x1
abc_xyz
```

Shell variables can be assigned values like this:

```
x=file1
x1=/usr/man/man1/sh.1
abc_xyz=4759300
```

Notice that there are no spaces before or after the equals-sign. The value will be substituted for the shell variable name if the name is preceded by a $. For example,

```
echo $x1
```

would echo

```
/usr/man/man1/sh.1
```
Several special shell variables are predefined. Some useful ones are
```
$#, $*, $?, and $$.
```
Arguments can be passed to a shell script. These arguments can be accessed inside the script by using the shell variables $1, $2,...,$n for positional parameter 1,2,...,n. The filename of the shell script itself is $0. The number of such arguments is $#. For example, if file do_it is a shell script and it is called by giving the command
```
do_it xyz
```
then $0 has the value do_it, $1 has the value xyz, and $# has the value 1.
$* is a variable containing all the arguments (except for $0) and is often used for passing all the arguments to another program or script.
$? is the exit status of the program most recently executed in the shell script. Its value is 0 for successful completion. This variable is useful for error handling (see section 10.6).
$$ is the process id of the executing shell and is useful for creating unique filenames. For example,
```
cat $1 $2  tempfile.$$
```
concatenates the files passed as parameters 1 and 2, appending them to a file called tempfile.31264 (assuming the process id is 31264).

Flow Control

Most programming languages provide constructs for looping and for testing conditions to know when to stop looping. The shell provides several such flow control constructs, including **if**, **for**, and **while**.
if
The **if** command performs a conditional branch. It takes the form

```
if command-list1
then      command-list2
else      command-list3
fi
```

A *command-list* is one or more commands. You can put more than one command on a line, but if you do so, separate them by semicolons. If the last command of *command-list1* has exit status 0, then *command-list2* is executed. But if the exit status is nonzero, then *command-list3* is executed.
for
The **for** command provides a looping construct of the form

```
for shell-variable in word-list
do command-list
done
```

The shell variable is set to the first word in *word-list* and then *command-list* is executed. The shell variable is then set to the next word in *word-list* and the process continues until *word-list* is exhausted. A common use of **for**-loops is to perform several commands on all (or a subset) of the files in your directory. For example, to print all the files in your directory, you could use

```
for i in *
do echo printing file $i
lpr $i
done
```

In this case, * would expand to a list of all filenames in your directory, **i** would be set to each filename in turn, and **$i** would then substitute the filename for **i** (in the **echo** and **lpr** commands).
while
The **while** command provides a slightly different looping construct:

```
while command-list1
do  command-list2
done
```

While the exit status of the last command in *command-list1* is 0, *command-list2* is executed.

Test

The **test** command can be used to compare two integers, to test if a file exists or is readable, to determine if two strings are equal, or to test several other conditions. For example, to test whether the value of shell variable x is equal to 5, use

```
test $x -eq 5
```

If **$x** is equal to 5, **test** returns true.
Other useful tests include

```
test -s file    (true if file exists and has a size larger than 0)
test -w file    (true if file exists and is writable)
test -z string  (true if the length of string is 0)
test string1 != string2 (true if string1 and string2 are not identical)
```

The **test** command is often used with the flow-control constructs described above. Here is an example of **test** used with the **if** command:

```
if test "$1" = ""            (or if ["$1" = ""] )
then
echo usage: myname  xxxx
exit 1
fi
```

This tests to see if the command line contains an argument (**$1**). If it does not (**$1** is null), then **echo** prints a message.
A complete list of test operators can be found in the man page for **test**.

Error Handling

Each time a program is executed from within a shell script, a value is returned to indicate whether the program ran successfully or not. In most cases, a value of zero is returned on successful execution, and a nonzero number is returned if the program encountered an error. This exit status is available in the shell variable **$?.**
For example,

```
grep $1 phonelist
if test $? -ne 0
then
        echo I have no phone number for $1
fi
```

will run a program (**grep**) and examine the exit status to determine if the program ran properly.

Traps

Some signals cause shell scripts to terminate. The most common one is the interrupt signal ^c typed while a script is running. Sometimes a shell script will need to do some cleanup, such as deleting temporary files, before exiting. The **trap** command can be used either to ignore signals or to catch them to perform special processing. For example, to delete all files called ``tmp.*'' before quitting when an interrupt signal is generated, use the command

```
trap 'rm tmp.*; exit' 2
```

The interrupt signal is signal 2, and two commands will be executed when an interrupt is received (**rm tmp.*** and **exit**). You can make a shell script continue to run after logout by having it ignore the hangup signal (signal 1). The command

```
trap ' ' 1
```

allows shell procedures to continue after a hangup (logout) signal.

Command Substitution

A useful capability in shell programming is to assign the output from a program to a shell variable or use it as a pattern in another command. This is done by enclosing the program call between accent grave (`) characters. For example, the command

```
where=`pwd`
```

will assign the string describing the current working directory (the results of the **pwd** command) to the shell variable **where**. Here is a more complicated example:

```
for i in `ls -t *.f`
do f77 $i
        a.out >output
        cat $i output | lpr -P$1
        rm a.out output
done
```

In this case, the shell script executes a series of commands for each file that ends with ``.f'' (all Fortran programs). The **`ls -t *.f`** is executed and expands into all filenames ending with ``.f'', sorted by time, most recent to oldest. Each is compiled and executed. Then the source file and output file are sent to the printer identified by the first argument ($1) passed to the shell script. Then these files are deleted.

I/O Redirection

Besides the I/O redirection already described, there are a few additional forms of redirection. Under the Bourne shell, standard input is also associated with file descriptor 0, standard output with file descriptor 1, and standard error output with file descriptor 2. So both file and 1file redirect standard output to file, and 2file redirects standard error output to file.
To merge standard output (file descriptor 1) and standard error output (file descriptor 2), then redirect them to another file, use this notation:

command file **2&1**

Another method of redirecting input in shell scripts allows a command to read its input from the shell script itself without using temporary files. For instance, to run the editor **ed** to change all x's in a file to z's, you could create a temporary file of **ed** commands, then read it to perform those commands, and finally delete it, like this:

```
 echo "1,$s/x/z/g" edtmp.$$
echo "w"     edtmp.$$
echo  "q"    edtmp.$$
ed  filename  <edtmp.$$
rm  edtmp.$$
echo  "x's  changed  to  z's"
```

The same thing can be accomplished without a temporary file by using the << symbol and a unique string, like this:

```
 ed filename <<%
1,$s/x/z/g
w
q
%
echo  "x's  changed  to  z's"
```

The << symbol redirects the standard input of the command to be right here in the shell script, beginning from the next line and continuing up to the line that matches the string following the << (in this case **%**). The terminating string must be on a line by itself. The string is arbitrary: for example, **<<EOF** will read up to a line that consists of the string EOF.

Debugging Shell Scripts

Two methods of tracing shell script execution are useful for debugging. The script may be called with a verbose flag (**-v**) or execution trace flag (**-x**). The **-v** flag causes each line of the shell script to be echoed as it is read but after all substitutions are performed. The **-x** flag causes each line to be echoed just before it is executed. For example:

```
sh -v do_it      sh -x do_it
```

To turn on both flags, use **-vx**.