More Unix: Files and The T-Shell

Brandy Bechtel
Joseph Pellegrini
MIT Project Athena
Revision B
June 1, 1987

1. Rough Draft

THIS IS A ROUGH DRAFT!!!
Many things are still changing.
Chapter XX or Chapter ?? stand for yet unspecified chapters
*text delimited by ** indicates thoughts or questions pertinent to*
the surrounding material

2. Introduction

Unix is the operating system that runs on Project Athena's workstations. The More Unix series is a supplement to Essential Unix. Knowledge of the information in *Essential Unix* and *More Unix-Files* is assumed in this document.

The More Unix series is directed at increasing your power and productivity on Athena by introducing some of the more advanced Unix commands and features.

When you finish reading this manual you will know:

- How to use the HISTORY mechanism
- How to set up aliases for frequently used commands
- How to program a shell script
- How to modify your .login and .cshrc
- How to create and control multiple processes
- What the T-Shell is used for
- How to create .Xkeymap and .tcshrc files

Every time you login to a Project Athena computer system, the system you are usn g starts a **shell** for you. A shell is essentially a command interpreter; it is a program that executes the commands you type. On Project Athena, the standard shell is the T-Shell. There are several other types of shell programs, including the Bourne Shell and the C-Shell. The T_Shell is in fact a superset of the C-Shell; a command that is native to the C-Shell will perform identically in the T-Shell. The only difference between the T-Shell and the C-Shell is that the T-Shell has some additional line editing features.

First we will explore advanced features of the C-Shell- the shell to which the T-Shell makes requests. Then we will examine some of the advanced and unique abilities of the T-Shell.

3. The C-Shell

The C-Shell acts as a translator between you and the Unix operating system; it reads the commands that you type into your terminal, interprets them, checks for syntax errors, and then **orders** the system to execute your instructions.

For example, when you type

```
%emacs myprog.c
```

it is the C-Shell that interprets your command, starts emacs, and tells emacs that you want to work on a file named *myprog.c*.Furthermore, the C-Shell is capable of doing much more than just interpreting one line commands. It is, in fact, a fairly sophisticated program you may use to create your own commands and customize your user interface.

The most important thing to remember about the C-Shell is that it is a program. It takes its input from your terminal keyboard, interacts with the operating system, and communicates with you by displaying information on your monitor.

The features of the C_Shell that allow you to do this include the following --

- a **history** mechanism that lets you recall command lines you have already typed,
- an **alias** mechanism that allows you to rename command lines, so they may be executed by a single word of your choice
- the ability to have multiple processes (programs) running simultaneously
- the ability to create and/or change **environment variables** and **shell variables** that are used by the C-Shell and other programs to describe your working environment and user interface
- and the ability to create **shell scripts**, which are programs consisting of a series of Unix and shell commands.

4. The History Mechanism

The **history** mechanism is one of the most useful features of the C shell. The **history** mechanism makes it possible to

- repeat commands that you have already executed,
- reuse arguments, such as filenames, from previous commands, or
- fix typing errors in command line on which you have already hit RETURN.

4.1 The History List

The C shell maintains a list of commands that you have executed, called the ‘‘history list.’’ The shell variable **history**, usually set-up in the *.cshrc* file in your home directory, specifies the length of the history list. The *.cshrc* file supplied by Project Athena sets the **history** variable to 40, which means that your forty most recent commands are in the list. You can see this list by typing **history** to the shell. For example, if the *history* variable is set to 5, you might see:

```
% history
     7  emacs myfile
     8  talk fakeuser
     9  date
    10  diff thisfile thatfile
    11  history
%
```

4.2 Re-Executing Previous Commands

When you want to make use of a previous command in the history list, you signal the fact with the character **!**. This **!** is called the "history substitution" character because it tells the shell you want to substitute some historical text for parts of the command to come.

You can put the ! anywhere in a command line. When you type a command line with a history substitution, the C shell echoes the command line as you type it, then prints the command line again with the history substitutions expanded to their actual values. This happens before the C shell executes the command line, so you can see exactly what commands the C shell executes.

The simplest case is the command **!!**, which runs the previous command again; for example:

```
% pwd
/mit/auser
% !!
pwd
/mit/auser
%
```

You can repeat other previous commands by specifying:

- the event number of the desired command line relative to the current command line, or
- the event number of the desired command line, or
- the first few letters of the initial command name in the desired command line

Thus, using the history list shown above as the example, you would execute the command **date** again by typing:

- **!-3**, since it is the third most recent command, or
- **!9**, since its event number is 9, or
- **!da**, since **da** is a unique prefix for this command

Your screen would appear as follows:

% **!-3**
date
Mon Feb 4 12:00 EST 1985
% **!9**
date
Mon Feb 4 12:00 EST 1985
% **!da**
Mon Feb 4 12:00 EST 1985

In general, you need only indicate enough letters to uniquely specify the string you want in the most recent command with that prefix. If you typed **!d**, the shell would execute **diff thisfile thatfile** again instead of **date.**

4.3 Substituting Parts of Previous Commands

It is also possible to select a particular word from a previous command line. (A "word" is a string of characters separated by spaces. For example, in the command **chmod 777 myfile**, the words are: **chmod**, **777**, and **myfile**.) These words are numbered beginning with 0, which is usually the name of the command. Word number 1 is the first argument, and so on. You select a particular word using a **:** (colon) after the event number, followed by the number of the word. For example, to edit the file **thisfile** from command line 10 in the history example above, you could type:

% **emacs !10:1**
emacs thisfile

and then Emacs would run with file *thisfile*.

After you select a word using the word designators, you may use several "modifiers" to it. With these, you can substitute one string for another, alter filenames, and so on. To make a substitution, use **s/*oldstring*/*newstring*** as in:

% **!di:s/file/prog/**
diff thisprog thatfile
{ no output would indicate that *thisprog* and *thatfile* are identical }
% **emacs !10:1:s/file/prog/**

emacs thisprog

and Emacs would edit the file *thisprog*.

The first example changes the command line **diff thisfile thatfile** to **diff thisprog thatfile** and the second takes word 1 from event number 10, substitutes the string **prog** for **file**, and uses the result as an argument to **emacs.**

A shorthand notation for the substitution command uses the up-arrow character **^**. When a line begins with a **^**, the history mechanism treats it as a substitution, with the original and replacement strings separated by another **^** character. For example,

% **ls /usr/unspported**
/usr/unspported not found
% **^spp^supp**
ls /usr/unsupported
{ a directory listing of */usr/unsupported* would print here }

In this case, we use the **^** to correct the spelling of **unspported** in the previous command line.

Another useful option is **t**, which strips the leading parts of the pathname. Thus,

% **ls /usr/unsupported/kermit**
/usr/unsupported/kermit not found
% **ls !!:1:t**
ls kermit
{ list of contents of the file *kermit* under the current directory }

Here the second command line takes the name from the first argument of the previous command.

The **p** option prints the command on the screen without evaluating it. This allows you to verify that you have the desired command. For example:

% **ls /usr/unsupported/kermit**
/usr/unsupported/kermit not found
% **ls !!:1:t:p**
ls kermit
%

5. Process Handling

Processes vary extensively in their nature. Some processes are started automatically when you login, such as the T-Shell. Others processes you initiate. Processes like the T-Shell and emacs are interactive- they require data from the user. Alternately there are many processes that do not need to send or receive data from your terminal. These processes can operate 'invisibly' to you and are called **background processes**.

5.1 What is a Process?

A process is any definable task that your computer may be working on. Since Unix is a multi-tasking system, your computer system can handle many processes at one time. A process must be **started** by executing the proper executable file. When you give your system a command, you are telling it to start a process. Processes, like files, have **owners**. The owner is the user who started that process and who has control over it. ** in some cases root owns the process(consider rlogins)- this needs clarification ** Processes also have a characteristic known as a **state**. Each state has a single letter for identification. Some of the different states a process may occupy include --

- R : presently running
- T : stopped process
- S : sleeping (dormant for less than 20 seconds)
- I : idle (dormant for more than 20 seconds)
- N : niced (uses limited amounts of CPU time if the load is heavy)

Processes, in general, need to input data from somewhere and output data to somewhere. In the case of the C-Shell process the input comes from the terminal keyboard (known as **stdin** or standard input) and the output is displayed on your terminal screen or xterm window (known as **stdout** or standard output). ** this may be redirected as in shell scripts ** Processes may also use files for their input and output, rather than the screen and keyboard. One example of this is the **scribe** program, which takes its input from a text file and sends its output to a print-formatted file. Some processes have the ability to create or start other processes. The C-Shell is an example of this; when you type a command to the C-Shell, you are actually asking the C-Shell to start that process for you. Any process that starts another process is referred to as the **parent** of the process it has started.

5.2 Monitoring Processes with ps

You may get information about the processes running on your system by using the **ps** command. The **ps** command without any parameters will give you a list of the processes that are owned by you.** but it doesn't always give you all of them ps aux |grep yourname the x option includes processes with no terminal ** For example,

```
% ps

   PID TT STAT  TIME COMMAND
   269 p4 I     0:44 emacs mac.mss
   276 p4 I     2:27 emacs main2.mss
   299 p4 R     0:00 ps
   226 v0 IW    0:04 -csh (csh)
   239 v0 I     0:01 uwm
   240 v0 S     0:01 xload
   241 v0 I     0:00 xclock
```

According to this example, you would have 7 processes running during the time the **ps** command was being executed. Note that the **ps** command itself is a process and it is running (indicated by the R) at the time the **ps** command is being executed.

The PID column indicates the **process id**, an identifier that is unique to each individual process.

The TT column indicates the terminal or xterm window that started the process. In the example above, **v0** indicates terminal#0 on the system and #b(p4) indicates an xterm window.

The W in the STAT column for the C-Shell (csh) process indicates that C-Shell process has been temporarily swapped out of memory.

The TIME column tells you the total CPU time that has been used by the process.

The COMMAND column identifies the process with a name that is meaningful to the user- as opposed to a number.

5.2.1 Getting More Information from ps

There are many options you may add onto the **ps** command in order to get more information. By using the **a** (all) and **u** (user format) options, you can get a listing of all** really ** the current processes on the computer system, in the following form:

```
% ps -au
USER       PID %CPU %MEM   SZ  RSS TT STAT TIME COMMAND
enjones  14192 24.2  4.0  166  114 p0 R    0:00 ps -au
mlpanes  13981  9.7  7.8  714  224 p4 S    1:08 emacs Guy.mss
root     13192  1.8  0.3   48    3 05 S    0:44 rlogin E40-32-5.MIT.EDU -l cha
sprager  14084  1.7  1.5   54   39 p6 S    0:08 mh-more inbox/1 inbox/2 inbox/
sumu     14095  1.1  9.3  706  270 p2 I    0:43 gnuemacs locke.mss
root     14003  0.8  0.4   48    6 01 IW   0:15 (nxterm)
root     14043  0.6  0.6   48   11 04 I    0:01 (loadst)
mrbird   14185  0.5  1.5   68   39 p5 I    0:00 msgs
root     14115  0.5  0.8   48   16 02 S    0:02 rlogin adam.PIKA.MIT.EDU -l ls
enjones  14092  0.1  1.7  140   43 p0 S    0:05 -csh (csh)
```

This example is a truncated section of an actual listing. During peak usage the list of processes on a large VAX can easily fill several pages. It is often convenient to alias(see Chapter ??) **ps** to **ps -au |more** using the following command:

```
alias ps  "ps -au |more"
```

example

This lists the current processes and prevents them from scrolling off the top of the screen.

If you would like more information about the **ps** command and some of its other options, you may type **man ps** to get a listing of its man page.

5.3 Background Processes

As noted above, there are some processes that do not need to interact with the user while they are operating. A good example of this would be a text formatter that takes input from one file and outputs to another file.

It is useful to take processes like these and place them in the **background**. Putting a process in the background is merely a way of telling the operating system that the process does not need to interact with the terminal during operation. This prevents the operating system from tying up your terminal, while it waits for the process to complete its task.

5.3.1 Processes That Start Their Own Windows

The most common reason for starting a background process is that the process you are starting creates its own window for input and output operations. Since each window on a workstation can be considered as a separate terminal, a process that starts its own window can be operated in the background without any difficulty.

Examples of processes that start their own windows include, but are not limited to,

- emacs
- xterm
- xclock
- xload

When you start any of the above programs, it is recommended that you start them running in the background. If you don't, the window you started them from will become "frozen" until the program is finished running. This can become very annoying.

5.3.2 Starting a Background Process

You may start a background process by adding a **&** symbol to the end of the command line you use to start the process.

```
% xclock option &
[1] 6219
%
```

In the above example the command issued by the user starts the **xclock** process which opens a window for the clock. Since the user has specified that this process can run in the background, the operating system immediately returns with a prompt in the window where the command was typed. You can now enter a new command and start another process while the **xclock** process is running.

The number that is returned by the computer, is the process's identification number. The process id may be used later to stop, restart, or kill a process.

5.4 How to Kill a Process

It is possible to stop a process from running and remove it from the process list. This is done with the **kill** command and may be done to any process that you own. The **kill** command takes a process's id as an argument. To find the id of a process you may use the **ps** command --

```
% ps

  PID TT STAT  TIME COMMAND
  269 p4 I     0:44 emacs mac.mss
  276 p4 I     2:27 emacs main2.mss
  299 p4 R     0:00 ps
  226 v0 IW    0:04 -csh (csh)
  239 v0 I     0:01 xwm
  240 v0 S     0:01 xload
  241 v0 I     0:00 xclock
```

To kill the **xclock** program in the above process list, you would type --

```
% kill 241
```

If this failed to work, you can use the **-9** option with the **kill** command. The **-9** guarantees a sure kill of the process. To avoid accidentally killing a critical process (like your **login** process) you should not use the **-9** option unless a normall **kill** has already failed. Normally, processes that can be safely killed will respond to **kill** without the **-9** option.

6. C-Shell Variables

The C-Shell recognized two types of variables ** are there more **

6.1 Environment Variables

Like any other program, the C-Shell uses variables to keep track of information. Some of these variables contain information concerning your working environment. These variables are properly named **environment variables**. Many of these variables may be modified by you for the purpose of customizing your working environment.

Environment variables may be used by the C-Shell itself, or they may be passed to an application program to determine default operation. There is virtually no limit to the number of environment variables; any program is allowed to establish its own environment variables for the purpose of setting up default values.

Table 6-1: Environment Variables

Variable	Used by *	Description
HOME	C-Shell	Pathname to your home directory
SHELL	C-Shell	Pathname to the login shell program
TERM	Any program	Identifies the name of the terminal you are on
TERMCAP	Any program	Description of the terminal identified by TERM or a pointer to a termcap file
USER	C-Shell	Your username
PATH	C-Shell	A list of paths to search for executable files
MAIL	mh	Pathname to the location of your mail spool
EDITOR	mh, others	Pathname to default text editor
VISUAL	mh, others	Pathname to default text editor
MORE	more	Default options for **more**
DISPLAY	X	Address for window output
APRINTER	lpr, lpq, lprm	Address of default printer

* In theory, all environment variables may, by definition, be used by any program. In practice, however, it is knwon that the programs listed in this column are the ones most likely to use the corresponding environment variables.

6.1.1 Looking at Environment Variable Values

To examine the values of your environment variables you should use the **printenv** command followed by the name of the environment variable you would like to see displayed. For example:

```
%printenv DISPLAY
sophocles:0
```

displays the current value of your **DISPLAY** variable.

Using **printenv** without any arguments will cause the C-Shell to display the values for all of the active environment variables. For example:

```
%printenv

HOME=/mit/e/n/enjones
SHELL=/bin/csh
TERM=xterm
USER=enrick
PATH=.:/mit/e/n/enjones/Bin:/usr/athena:/usr/new:/usr/new/mh:/usr/ucb:/bin:
      /usr/bin:/usr/local:/usr/games:/usr/hosts:/usr/unsupported:/usr/man:
      /usr/unsupported/sipb
MAIL=/usr/spool/mail/enjones
VISUAL=/usr/athena/emacs
EDITOR=/usr/athena/emacs
MORE=-cs
TERMCAP=vs|xterm|vs100:cr=^M:do=^J:nl=^J:bl=^G:le=^H:ho=\E[H:co#82:li#58:cl=\
E[H\E[2J:bs:am:cm=\E[%i%d;%dH:nd=\E[C:up=\E[A:ce=\E[K:cd=\E[J:so=\E[7m:se=2\E
[m:us=\E[4m:ue=\E[m:md=\E[1m:mr=\E[7m:me=2\E[m:ku=\EOA:kd=\EOB:kr=\EOC:kl=\EOD
:kb=^H:k1=\EOP:k2=\EOQ:k3=\EOR:k4=\EOS:ta=^I:pt:sf=\n:sr=\EM:al=\E[L:dl=\E[M:i
c=\E[dc=\E[P:MT:ks=\E[?1h\E=:ke=\E[?1l\E>:is=\E[r\E\E[m\E[H\E[?7h\E[?1;3;4;6
l:rs=\E[r\E<\E[m\E[H\E[?7h\E[?1;3;4;6l:xn:
DISPLAY=sophocles:0
```

6.1.2 Setting Environment Variable Values

You may create an environment variable, or change the value of an existing variable by using the **setenv** command. The syntax for using this command is

```
% setenv [VARIABLE-NAME] [value]
```

For example, if you would like to set your **DISPLAY** variable to the value *sophocles:0* you would type

```
% setenv DISPLAY sophocles:0
```

If the value you are setting has embedded spaces, it is necessary to enclose the **[value]** argument in quotes. For example, if you wanted to set the environment variable **NAME** to the value *Capt. Quark* you would type

```
% setenv NAME "Capt. Quark"
```

Most of your environment variables are automatically set by your *.login* and *.cshrc* files, located in your home directory. Your *.login* and *.cshrc* files are discussed in chapter XX of this document.

6.1.3 Using Your PATH Variable

The environment variable **PATH** serves a very useful function. **PATH** contains a list of *pathnames* that the C-Shell will automatically search when you type a command.

For example, if our **PATH** variable is set as

```
PATH=.:/mit/e/n/enjones/Bin:/usr/athena:/usr/new:/usr/new/mh:/
      /usr/bin:/usr/local:/usr/games:/usr/hosts:/usr/unsupport
      /usr/unsupported/sipb
```

and we type the command

```
%xterm &
```

the C-Shell will search for the executable file, *xterm*, in the directories listed in the **PATH** variable. The C-Shell checks the *pathnames* in **PATH** from left to right.

In our example, this means the C-Shell will first search your current directory (denoted by the single dot), and then move on and check */mit/e/n/enjones*, continuing through all the pathnames listed until the command is found, or the end of the **PATH** variable is reached.

Note: The ordering of the *pathnames* within **PATH** is significant, especially if there are several versions of the same command exisiting in different directories. Remember -- the leftmost pathnames are searched first.

If the C-Shell is unable to find the command you typed in any of the directories pointed to by **PATH**, it will return a ***Command not found*** error.

Your **PATH** variable is normally set by your *.login* file, and should be changed there. Refer to chapter XX of this manual for more information on your *.login* file.

6.2 Shell Variables

There is another set of variables used by the C-Shell called **shell variables**. Shell variables are very similar to environment variables except

- shell variables are used by the C-Shell only
- shell variables are not passed to programs
- the labels of shell variables are all lower case

- values held in shell variables do not propogate into new shells
- different commands are used to look at and change shell variables

Many shell variables are duplicates in name and value of some of the environment variables. These shell variables have identical environment variable counterparts so that their values may be carried over when new shells are started. The best example of this is the PATH/path variable.

It should be noted that changing the value of one variable in a shell/environment pair does not automatically change the value of the other variable.

Table 6-2: Shell Variables

Variable	Description
cdpath	List of pathnames to search when you issue a **cd** command
cwd	Pathname of your current working directory
history	Number of commands to save in your history list
ingnoreeof	Ignore end-of-file flag: stops CTRL-D from logging you out
prompt	Input prompt string (i.e. 'sophocles%')
path	Identical to the environment variable PATH
savehist	Number of commands to save when you logout
user	Your username

6.2.1 How to Look at Shell Variables

You make look at the values of **shell variables** in the same way you look at environment variables. The shell variable analogy of **setenv** command is **set**. Therefore, to see a list of all your shell variables and their values, you would type

```
%set
argv            ()
cdpath          (/mit/enjones /mit/enjones/Munix.dir)
cwd             /mit/enjones/Munix.dir/C_Shell.dir
history         40
home            /mit/enjones
ignoreeof
mail            (120 /usr/spool/mail/enjones)
old             /site/mit/enjones
path            (. /mit/enjones/Bin /bin/athena /usr/athena /usr/new
                /usr/new/mh /usr/ucb /bin /usr/bin /usr/local /usr/gam
                /usr/hosts)
prompt          sophocles%
savehist        50
shell           /bin/csh
term            xterm
user            enjones
```

In most cases, shell variables will hold values in the same way environment variables do. However, shell variables can also act as flags, whereby their mere existence is recognized by the C-Shell. An example of this can be seen above in the shell variable **ignoreeof**, which tells the C-Shell to ignore end-of-file commands from the keyboard. This prevents you from accidentally loggin out with a CTRL-D.

6.2.2 How to Change the Value of Shell Variables

The syntax for setting or changing a shell variable is

set `shell_variable`

to create a shell variable as a valueless flag, or

set `shell_variable=value`

to set the value of *shell_variable* to *value*.

For example,

```
%set prompt=">"
```

will set the input prompt (normally *hostname%*) to the > symbol.

6.3 Evaluating Shell and Environment Variables

You may use the **echo** command to display the evaluation of any C-Shell variable. To evaluate a variable place a **$** in front of it. For example:

```
%echo $USER
enjones
```

The **$** in front of the variable name translates to **"the evaluation of"**.

The ability to evaluate variables in the C-Shell can be very useful. For example, say you are doing work in a directory with a path */foo/bar/doc/etc/test* and you want to be able to jump to this directory very quickly. You can define an **environment variable** that will help you do this:

```
% setenv JUMP '/foo/bar/doc/etc/test'
% echo $JUMP
/foo/bar/doc/etc/test
% cd $JUMP
```

As you can see, once the variable, JUMP, is defined with the path of a directory, you can jump to that directory by using **cd** and using the **$** symbol to evaluate the value of JUMP.

The ability to evaluate shell and environment variables is very useful in the programming and implementation of **shell scripts**, a subject that is covered in chapter XX of this document.

7. Using Aliases

One of the major features of the C-Shell is its ability to customize your operating environment. One path the C-Shell uses to accomplish this is through the use of **aliases**.

An **alias** is a alternate name you can assign to a command or set of commands. For example, you are printing a series of files to a laser printer with the address *ln03-bldg99-09*. To send your files to this printer you would normally have to type

`% lpr -Pln03-bldg99-09` *filename*

for each and every file you are sending to the printer. By using the **alias** command you can eliminate the need of typing this long command over and over. The first step is to define an alias for the command. In the example below we define the alias using the name **laser** --

```
% alias laser 'lpr -Pln03-bldg99-09'
```

Note that the value being assigned to **laser** is enclosed in quotes, so that the embedded spaces are also captured.

Once an alias is defined it may be used in place of the command it defines. In the example above, we can now send any file to the laser printer with the command:

`% laser` *filename*

As you can see, the proper use of aliases can make your life at the keyboard much more pleasant.

7.1 Using alias to Define Multiple Commands

It is possible, using **alias**, to execute several programs successively by typing a single command word. This is done by placing a semicolon in between the commands when you define an alias.

For example -- you want to create an alias that will clear your window and list the files in you current directory. To do this you can define an alias called **dir** with the following command:

```
% alias dir 'clear ; ls -l'
```

You may want to play around with variations on the above idea, in order to get more familiar with the characteristics of the alias command.

7.2 Miscellaneous Facts About alias

You should take note of the following characteristics of the **alias** command.

First, **alias** has the ability to detect loops within an alias definition and will not allow execution of an alias that will force an infinite loop. For example

```
%alias a b
%alias b a
%a
Alias loop.
%
```

Second, you may list all the current alias definitions by typing **alias** without any arguments. For example--

```
back    set back=$old; set old=$cwd; cd $back; unset back; dir
cd      set old=$cwd; chdir !*
in      /usr/athena/in !* ; source ~/.cshrc; source ~/.login
laser   lpr -Pln03-bldg99-09
mail    Mail
term    setenv TERM ‘tset -Q - !*‘
xresize set noglob; eval ‘resize‘
```

You may notice that there are some aliases already existing in your shell that you did not create. These aliases have been set automatically by your *.login* or *.cshrc* files. For more information about *.login* and *.cshrc* refer to chapter XX of this document.

8. Shell Scripts

8.1 What is a Shell Script?

A shell script, simply defined, is a series of Unix commands that may be executed with a single command. The concept of combining several commands under one name is called **batching** and is a common feature on most computers.

The major purpose of a C-Shell script is very similar to that of the **alias** command. You will find, however, that shell scripts are much more powerful and are applicable to a wider variety of tasks than simple aliases.

We will begin our excursion into the world of shell scripts with a simple example. You would like to create a shell shell script named *print-one* that will perform the following steps:

1. **scribe** the file named *document.mss* located in **~/thesis/originals**.
2. Use **lpr** to send the output to the printer named **ln03-bldg99-9**.
3. Remove the file **document.lpt** after it has been printed.

The benefit of combining these three tasks in a shell script is that once you execute the shell script you can ignore the whole process until your printout is ready to be picked up. This is the "hands off" characteristic of shell scripts.

In the next two sections we will describe the steps for programming and executing this shell script.

8.2 Programming a Shell Script

A shell script is stored in the file system as a **text file**. Therefore, to create it you need a text editor, like **emacs**.

The rules for programming a shell script are as follows --

- The first character on the first line must be a **#** symbol
- Any characters between a **#** symbol and a carriage return are considered comments and are ignored at execution. (# symbols within batched commands are not considered as comment indicators; the C-Shell is intelligent enough to recognize that these symbols are parts of the commands.)
- Each Unix command must exist on its own line, followed only by a carriage return or a comment and a carriage return.
- Leading spaces and tabs are ignored.

For the example given in the previous section, the following script file will do the job we want:

```
#  This script file will perform the following three tasks
#
#               1) scribe document.mss in ~/thesis/originals
#               2) Send the output to the ln03 in building 99
#               3) Delete the .lpt file created by scribe
#
#

        scribe ~/thesis/originals/document.mss

        lpr -Pln03-bldg99-9 ~/thesis/originals/document.lpt

        rm ~/thesis/originals/document.lpt
```

Once you have typed your script into **emacs**, all you need to do is save your script file by using CTRL-x CTRL-s and entering a name to save it under. After you have saved your script file, you are ready to prepare it for execution. For our example, we have decided to call our shell script **print-one**.

8.3 Executing a Shell Script

To prepare a script file for execution you need to set its **execute permission**. This only needs to be accomplished once and is done by typing

% chmod a+x *script-file-name*

or

% chmod u+x *script-file-name*

where *script-file-name* is the name you have saved the script file under(**print-one** in our example). In the first case (% chmod a+x), everyone is given permission to run your shell script. In the second example (% chmod u+x), only you, the owner of the file, are given permission to execute the script.

Once you have set the execute permission the shell script is ready to be run. To execute a shell script simply type its name, as if it were a regular Unix command. For the example given above we would type

% print-one

8.4 Advanced Shell Script Programming

The example shown in the previous section could easily have been accomplished using a simple **alias** command. With some of the advanced shell script tools, described in the following section, you will find it possible to perform powerful tasks, far beyond the scope of our first example. These tools include the ability to interactively communicate with the user as the shell is running, the ability to access information in shell variables, and several commands designed for use within a shell script.

8.4.1 Using Shell Variables in a Script

One of the major purposes of shell variables(see Chapter 4) is their use in shell scripts. By using the **$** evaluation symbol, it is possible to make shell scripts much more flexible and efficient. For example, we can generalize the example given in the previous section to do the following:

1. **scribe** the file whose name is stored in the environment variable **FILE** located in the directory whose path is stored in the variable **DIRECTORY**.
2. Use **lpr** to send the output to the printer whose address is stored in **DPRINTER**.
3. Remove the file $FILE.lpt after it has been printed.

We assume the environment variables needed are set by the user. The resulting script becomes

```
# This script file will perform the following three tasks
#
#              1) scribe $FILE.mss in $DIRECTORY
#              2) Send the output to $DPRINTER
#              3) Delete the .lpt file created by scribe
#
#

        scribe $DIRECTORY/$FILE.mss

        lpr -P$DPRINTER $DIRECTORY/$FILE.lpt

        rm $DIRECTORY/$FILE.lpt
```

To execute this shell script the user would have to enter the following commands -

```
% setenv FILE 'document'
% setenv DIRECTORY '~/thesis/originals'
% setenv DPRINTER 'ln03-bldg99-9'
% print-one
%
```

In the above example, the C-Shell forces the evaluation of the environment variables following the **$** symbol and substitutes the evaluation prior to execution of the command.

Environment variables may be set from within a shell by including the **setenv** commands as part of the batch.

8.4.2 Passing Arguments to Scripts

One useful feature of a shell script is the ability to pass arguments to the script. These arguments can be accessed by the shell script just like environment variables.

Let's create a shell script with the following features. Given a command of the form

`% cleardir` *directory action*

we would like a shell script that will find all the 'dead files' in (and below) the argument given by *directory* and perform the command in *action* upon them. In our case we will consider 'dead files' to include files ending with **~** symbols, files beginning with **#** symbols, and files named **core**.

For example given the command

```
% cleardir ~ rm
```

we want our shell script to delete all the 'dead files' in our home directory and all of its subdirectories.

If we are a little more cautious we might give the command

```
% cleardir ~/programs 'mv ~/dead'
```

which would find all the 'dead files' in and below the subdirectory names *programs* and move them to the subdirectory named *dead*.

A script file that would do this will need to access the two arguments being passed to it. The way arguments are passed gives each argument a unique variable name based upon its location in the argument list following the command. In the example given above the *directory* argument would be given the variable name **1** and would be evaluated using **$1**. The *action* argument would be given the variable name **2** and would be evaluated using **$2**.

The shell script for the above example could be programmed as follows --

```
# This shell script  removes all 'dead' files from
# the directory given as an argument and all the
# directories below it.
#
# The following files are removed:
#
#       1) All 'core' files
#       2) All *~ files
#       3) All #* files
#
# This shell script uses the 'find' command, please refer
# to More Unix Files for information on the 'find' comman

        find $1 -name "core" -exec $2 {} \;

        find $1 -name "*~" -exec $2 {} \;

        find $1 -name "#*" -exec $2 {} \;
```

The syntax of the **find** command is documented in **More Unix Files**. You may also type **man find**, while online, to learn more about this useful command.

8.4.3 Sending Messages to the User

The **echo** command allows a shell script to display messages on the user's terminal. The echo command has the following syntax.

% echo [-n] *expression*

The **-n** option is used to suppress the default carriage return normally tacked onto output. The *expression* argument is evaluated and displayed on the terminal. Echo removes quotation marks and evaluates commands enclosed in ' marks. Special characters such as ! and \ are recognized as such and therefore must be blackslashed. The following examples illustrate how **echo** may be used:

```
% echo "Hello, how are you today?"
Hello, how are you today?
%
% setenv MESSAGE "C-Shell programming is fun\!"
% echo $MESSAGE
C-Shell programming is fun!
%
% echo 'hostname':'pwd'\!
your_machine_name:your_current_working_directory!
```

Since the **cleardir** shell script (described in the previous section) may take a long time to execute, it should inform the user as to what task it is currently working on. The following edited **cleardir** shell script does this:

```
# This shell script  removes all 'dead' files from
# the directory given as an argument and all the
# directories below it.
#
# The following files are removed:
#
#       1) All 'core' files
#       2) All *~ files
#       3) All #* files
#
# This shell script uses the 'find' command, please refer
# to More Unix Files for information on the 'find' comman

        echo
        echo -n "Directory to search = "
        echo $1
        echo

        echo -n "Removing all core files..."
        find $1 -name "core" -exec $2 {} \;
        echo "done"

        echo -n "Removing all ~ files..."
        find $1 -name "*~" -exec $2 {} \;
        echo "done"

        echo -n "Removing all # files..."
        find $1 -name "#*" -exec $2 {} \;
        echo "done"
        echo
```

The output from this script file would look like this:

```
%
% cleardir ~/programs  rm

Directory to search = ~/programs

Removing all core files...done
Removing all ~ files...done
Removing all # files...done

%
```

8.4.4 Reading Data From the Keyboard

8.4.5 Conditional Statements

8.4.6 Useful Unix Commands

8.5 Sample Shell Scripts

9. Using .cshrc and .login

10. Features of the T-Shell

The T-Shell is, in fact, a new level added onto the standard C-Shell. Any commands you enter while using the T-Shell are actually processed by the C-Shell. This is done when the T-Shell performs a call to the C-Shell.

10.1 Using the T-Shell-Starting Out

10.2 Using the T-Shell-Commands

10.3 Personalizing the T-Shell

10.3.1 The .tcshrc File

10.3.2 The .Xkeymap File

10.4 More Information

Table of Contents

List of Tables