Chapter 8

Shell Programming

Introduction

Shell Programming − WHY?

While it is very nice to have a shell at which you can issue commands, have you had the feeling that something is missing? Do you feel the urge to issue multiple commands by only typing one word? Do you feel the need for variables, logic conditions and loops? Do you strive for automation?

If so, then welcome to shell programming.

(If you answered no to any of the above then you are obviously in the wrong frame of mind to be reading this − please try again later :)

Shell programming allows system administrators (and users) to create small (and occasionally not−so−small) programs for various purposes including automation of system administration tasks, text processing and installation of software.

Perhaps the most important reason why a Systems Administrator needs to be able to read and understand shell scripts is the UNIX startup process. UNIX uses a large number of shell scripts to perform a lot of necessary system configuration when the computer first starts. If you can't read shell scripts you can't modify or fix the startup process.

Shell Programming − WHAT?

A shell program (sometimes referred to as a shell script) is a text file containing shell and UNIX commands. Remember − a UNIX command is a physical program (like **`cat`**, **`cut`** and **`grep`**) where as a shell command is an "interpreted" command − there isn''t a physical file associated with the command; when the shell sees the command, the shell itself performs certain actions (for example, **`echo`**)

When a shell program is executed the shell reads the contents of the file line by line. Each line is executed as if you were typing it at the shell prompt. There isn't anything that you can place in a shell program that you can't type at the shell prompt.

Shell programs contain most things you would expect to find in a simple programming language. Programs can contain services including:

- variables
- logic constructs (IF THEN AND OR etc)
- looping constructs (WHILE FOR)
- functions
- comments (strangely the most least used service)

The way in which these services are implemented is dependant on the shell that is being used (remember − there is more than one shell). While the variations are often not major it does mean that a program written for the bourne shell (

`sh`/`bash`) will not run in the c shell (`csh`). All the examples in this chapter are written for the bourne shell.

Shell Programming − HOW?

Shell programs are a little different from what you'd usually class as a program. They are plain text and they don't need to be compiled. The shell "interprets" shell programs − the shell reads the shell program line by line and executes the commands it encounters. If it encounters an error (syntax or execution), it is just as if you typed the command at the shell prompt − an error is displayed.

This is in contrast to C/C++, Pascal and Ada programs (to name but a few) which have source in plain text, but require compiling and linking to produce a final executable program.

So, what are the real differences between the two types of programs? At the most basic level, interpreted programs are typically quick to write/modify and execute (generally in that order and in a seemingly endless loop :). Compiled programs typically require writing, compiling, linking and executing, thus are generally more time consuming to develop and test.

However, when it comes to executing the finished programs, the execution speeds are often widely separated. A compiled/linked program is a binary file containing a collection direct systems calls. The interpreted program, on the other hand, must first be processed by the shell which then converts the commands to system calls or calls other binaries − this makes shell programs slow in comparison. In other words, shell programs are not generally efficient on CPU time.

Is there a happy medium? Yes! It is called Perl. Perl is an interpreted language but is interpreted by an extremely fast, optimised interpreter. It is worth noting that a Perl program will be executed inside one process, whereas a shell program will be interpreted from a parent process but may launch many child processes in the form of UNIX commands (ie. each call to a UNIX command is executed in a new process). However, Perl is a far more difficult (but extremely powerful) tool to learn − and this chapter is called "Shell Programming"...

The Basics

A Basic Program

It is traditional at this stage to write the standard "Hello World" program. To do this in a shell program is so obscenely easy that we're going to examine something a bit more complex − a hello world program that knows who you are...

To create your shell program, you must first edit a file − name it something like "hello", "hello world" or something equally as imaginative − just don't call it "test" − we will explain why later.

In the editor, type the following (or you could go to the 85321 website/CD−ROM and cut and paste the text from the appropriate web page)

```
#!/bin/bash
# This is a program that says hello
echo "Hello $LOGNAME, I hope you have a nice day!"
```

(You may change the text of line three to reflect your current mood if you wish)

Now, at the prompt, type the name of your program − you should see something like:

```
bash: ./helloworld: Permission denied
```

Why?

The reason is that your shell program isn't executable because it doesn't have its execution permissions set. After setting these (Hint: something involving the `chmod` command), you may execute the program by again typing its name at the prompt.

An alternate way of executing shell programs is to issue a command at the shell prompt to the effect of:

```
<shell> <shell program>
```

eg

```
bash helloworld
```

This simply instructs the shell to take a list of commands from a given file (your shell script). This method does not require the shell script to have execute permissions. However, in general you will execute your shell scripts via the first method.

And yet you may still find your script won''t execute − why? On some UNIX systems (Red Hat Linux included) the current directory (.) is not included in the **`PATH`** environment variable. This mans that the shell can''t find the script that you want to execute, even when it''s sitting in the current directory! To get around this either:

§ Modify the PATH variable to include the "." directory:

```
PATH=$PATH:.
```

§ Or, execute the program with an explicit path:

```
./helloworld
```

An Explanation of the Program

Line one, **`#!/bin/bash`** is used to indicate which shell the shell program is to be run in. If this program was written for the C shell, then you might have **`#!/bin/csh`** instead.

It is probably worth mentioning at this point that UNIX "executes" programs by first looking at the first two bytes of the file (this is similar to the way MS−DOS looks at the first two bytes of executable programs; all .EXE programs start with "MZ"). From these two characters, the system knows if the file is an interpreted script (**`#!`**) or some other file type (more information can be obtained about this by typing **`man file`**). If the file is an interpreted script, then the system looks for a following path indicating an interpreter. For example:

```
#!/bin/bash
#!/usr/bin/perl
#!/bin/sh
```

Are all valid interpreters.

Line two, **`# This is a program that says hello`**, is (you guessed it) a comment. The "#" in a shell script is interpreted as "anything to the right of this is a comment, go onto the next line". Note that it is similar to line one except that line one has the "`!`" mark after the comment.

Comments are a very important part of any program − it is a really good idea to include some. The reasons why are standard to all languages − readability, maintenance and self congratulation. It is more so important for a system administrator as they very rarely remain at one site for their entire working career, therefore, they must work with other people's shell scripts (as other people must work with theirs).

Always have a comment header; it should include things like:

```
# AUTHOR:       Who wrote it
# DATE:         Date first written
# PROGRAM:      Name of the program
# USAGE:        How to run the script; include any parameters
# PURPOSE:      Describe in more than three words what the
#               program does
#
# FILES:        Files the shell script uses
#
# NOTES:        Optional but can include a list of "features"
#               to be fixed
#
# HISTORY:      Revisions/Changes
```

This format isn't set in stone, but use common sense and write fairly self documenting programs.

> *Version Control Systems*
>
> Those of you studying software engineering may be familiar with the term, version control. Version control allows you to keep copies of files including a list of who made what changes and what those changes were. Version control systems can be very useful for keeping track of source code and is just about compulsory for any large programming project.
>
> Linux comes with CVS (Concurrent Versions System) a widely used version control system. While version control may not seem all that important it can save a lot of heartache.
>
> Many large sites will actually keep copies of system configuration files in a version control system.

Line three, **`echo "Hello $LOGNAME, I hope you have a nice day!"`** is actually a command. The **`echo`** command prints text to the screen. Normal shell rules for interpreting special characters apply for the **`echo`** statement, so you should generally enclose most text in "". The only tricky bit about this line is the **`$LOGNAME`**. What is this?

`$LOGNAME` is a shell variable; you can see it and others by typing "set" at the shell prompt. In the context of our program, the shell substitutes the **`$LOGNAME`** value with the username of the person running the program, so the output looks something like:

```
Hello jamiesob, I hope you have a nice day!
```

All variables are referenced for output by placing a "**$**" sign in front of them − we will examine this in the next section.

Exercises

Modify the **helloworld** program so its output is something similar to:
`Hello <username>, welcome to <machine name>`

All You Ever Wanted to Know About Variables

You have previously encountered shell variables and the way in which they are set. To quickly revise, variables may be set at the shell prompt by typing:

```
[david@faile david]$ variable="a string"
```

Since you can type this at the prompt, the same syntax applies within shell programs.

You can also set variables to the results of commands, for example:

```
[david@faile david]$ variable=‘ls −al‘
```

(Remember − the ‘ is the execute quote)

To print the contents of a variable, simply type:

```
[david@faile david]$ echo $variable
```

Note that we've added the "**$**" to the variable name. **Variables are always accessed for output with the "$" sign, but without it for input/set operations.**

Returning to the previous example, what would you expect to be the output?

You would probably expect the output from **ls −al** to be something like:

```
drwxr−xr−x   2 jamiesob users        1024 Feb 27 19:05 ./
drwxr−xr−x  45 jamiesob users        2048 Feb 25 20:32 ../
−rw−r−−r−−   1 jamiesob users         851 Feb 25 19:37 conX
−rw−r−−r−−   1 jamiesob users       12517 Feb 25 19:36 confile
−rw−r−−r−−   1 jamiesob users           8 Feb 26 22:50 helloworld
−rw−r−−r−−   1 jamiesob users       46604 Feb 25 19:34 net−acct
```

and therefore, printing a variable that contains the output from that command would contain something similar, yet you may be surprised to find that it looks something like:

```
drwxr−xr−x 2 jamiesob users 1024 Feb 27 19:05 ./ drwxr−xr−x 45
jamiesob users 2048 Feb 25 20:32 ../ −rw−r−−r−− 1 jamiesob users 851
Feb 25 19:37 conX −rw−r−−r−− 1 jamiesob users 12517 Feb 25 19:36
confile −rw−r−−r−− 1 jamiesob users 8 Feb 26 22:50 helloworld −rw−r−−
r−− 1 jamiesob users 46604 Feb 25 19:34 net−acct
```

Why?

When placing the output of a command into a shell variable, the shell removes all the end−of−line markers, leaving a string separated only by spaces. The

use for this will become more obvious later, but for the moment, consider what the following script will do:

```
#!/bin/bash
$filelist=‘ls‘
cat $filelist
```

Exercise

> Type in the above program and run it. Explain what is happening. Would the above program work if "**`ls -al`**" was used rather than "**`ls`**" − Why/why not?

Predefined Variables

There are many predefined shell variables, most established during your login. Examples include **`$LOGNAME`**, **`$HOSTNAME`** and **`$TERM`** − these names are not always standard from system to system (for example, **`$LOGNAME`** can also be called **`$USER`**). There are however, several standard predefined shell variables you should be familiar with. These include:

```
$$    (The current process ID)
$?    (The exits status of last command)
```

How would these be useful?

`$$`

`$$` is extremely useful in creating unique temporary files. You will often find the following in shell programs:

```
some command > /tmp/temp.$$
.
.
some commands using /tmp/temp.$$>
.
.
rm /tmp/temp.$$
```

`/tmp/temp.$$` would always be a unique file − this allows several people to run the same shell script simultaneously. Since one of the only unique things about a process is its PID (Process−Identifier), this is an ideal component in a temporary file name. It should be noted at this point that temporary files are generally located in the **`/tmp`** directory.

`$?`

`$?` becomes important when you need to know if the last command that was executed was successful. All programs have a numeric exit status − on UNIX systems 0 indicates that the program was successful, any other number indicates a failure. We will examine how to use this value at a later point in time.

Is there a way you can show if your programs succeeded or failed? Yes! This is done via the use of the **`exit`** command. If placed as the last command in your shell program, it will enable you to indicate, to the calling program, the exit status of your script.

`exit` is used as follows:

```
exit 0            # Exit the script, $? = 0 (success)
exit 1            # Exit the script, $? = 1 (fail)
```

Another category of standard shell variables are shell parameters.

Parameters − Special Shell Variables

If you thought shell programming was the best thing since COBOL, then you haven't even begun to be awed − shell programs can actually take parameters. Table 8.1 lists each variable associated with parameters in shell programs:

Variable	Purpose
$0	the name of the shell program
$1 thru $9	the first thru to ninth parameters
$#	the number of parameters
$*	all the parameters passed represented as a single word with individual parameters separated
$@	all the parameters passed with each parameter as a separate word

Table 8.1
Shell Parameter Variables

The following program demonstrates a very basic use of parameters:

```
#!/bin/bash
# FILE:         parm1
VAL=‘expr ${1:−0} + ${2:−0} + ${3:−0}‘
echo "The answer is $VAL"
```

> ***Pop Quiz**: Why are we using **`${1:−0}`** instead of **`$1?`** Hint: What would happen if any of the variables were not set?*

A sample testing of the program looks like:

```
[david@faile david]$ parm1 2 3 5
The answer is  10

[david@faile david]$ parm1 2 3
The answer is  5

[david@faile david]$ parm
The answer is  0
```

Consider the program below:

```
#!/bin/bash
# FILE:         mywc

FCOUNT=‘ls $*  2> /dev/null | wc −w‘
echo "Performing word count on $*"
echo
wc −w $* 2> /dev/null
echo
echo "Attempted to count words on $# files, found $FCOUNT"
```

If the program that was run in a directory containing:

```
conX          net−acct      notes.txt     shellprog~    t1~
confile       netnasties    notes.txt~    study.htm     ttt
helloworld    netnasties~   scanit*       study.txt     tes/
my_file       netwatch      scanit~       study_~1.htm
mywc*         netwatch~     shellprog     parm1*
```

Some sample testing would produce:

```
[david@faile david]$ mywc mywc
Performing word count on mywc

34 mywc

Attempted to count words on 1 files, found       1
[david@faile david]$ mywc mywc anotherfile
Performing word count on mywc anotherfile

34 mywc
34 total


Attempted to count words on 2 files, found       1
```

Exercise

> Explain line by line what this program is doing. What would happen if the user didn't enter any parameters? How could you fix this?

Only Nine Parameters?

Well that's what it looks like doesn't it? We have **$1** to **$9** − what happens if we try to access **$10?** Try the code below:

```
#!/bin/bash
# FILE:           testparms
echo "$1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11 $12"
echo $*
echo $#
```

Run testparms as follows:

```
[david@faile david]$ testparms a b c d e f g h I j k l
```

The output will look something like:

```
a b c d e f g h i a0 a1 a2
a b c d e f g h I j k l
12
```

Why?

The shell only has 9 command−line parameters defined at any one time **$1** to **$9**. When the shell sees "**$10**" it interprets this as "**$1**" with a "**0**" after it. This is where $10 in the above results in a0. The a is the value of $1 with the 0 added.

On the otherhand **$*** allows you to see all the parameters you typed!

So how do you access $10, $11 etc. To our rescue comes the **shift** command. **shift** works by removing the first parameter from the parameter list and shuffling the parameters along. Thus **$2** becomes **$1**, **$3** becomes **$2** etc. Finally, (what was originally) the tenth parameter becomes **$9**. However, beware! Once you've run **shift**, you have lost the original value of **$1** forever − it is also removed from **$*** and $@. **shift** is executed by, well, placing the word "shift" in your shell script, for example:

```
#!/bin/bash
echo $1 $2 $3
```

```
shift
echo $1 $2 $3
```

Exercise

Modify the **testparms** program so the output looks something like:

a b c d e f g h i a0 a1 a2
a b c d e f g h I j k l
12
b c d e f g h i j b1 b2 b3
b c d e f g h i j k l
11
c d e f g h i j k c0 c1 c2
c d e f g h I j k l
10

The difference between *$** and *$@*

$* and $@ are very closely related. They both are expanded to become a list of all the command line parameters passed to a script. However, there are some subtle differences in how these two variables are treated. The subtleties are made even more difficult when they appear to act in a very similar way (in some situations). For example, let's see what happens with the following shell script

```
#for name in $*
for name in $@
do
  echo param is $name
done
```

The idea with this script is that you can test it with either $* or $@ by uncommenting the one you want to experiment and comment out the other line. The following examples show what happens when I run this script. The first time with $@, the second with $*

```
[david@faile david]$ tmp.sh hello "how are you" today 1 2 3
param is hello
param is how
param is are
param is you
param is today
param is 1
param is 2
param is 3
[david@faile david]$ tmp.sh hello "how are you" today 1 2 3
param is hello
param is how
param is are
param is you
param is today
param is 1
param is 2
param is 3
```

As you can see no difference!! So what's all this fuss with $@ and $*? The difference comes when $@ and $* are used within double quotes. In this situation they work as follows

- $@
  Is expanded to all the command−line parameters joined as a single word

with usually a space seperating them (the separating character can be changed).

- $*
  Expands to all the command−line parameters BUT each command−line parameter is treated as if it is surrounded by double quotes "". This is especially important when one of the parameters contains a space.

Let's modify the our example script so that $@ and $* are surrounded by ""

```
#for name in "$*"
for name in "$@"
do
  echo param is $name
done
```

Now look at what happens when we run it using the same parameters as before. Again the $@ version is executed first then the $* version.

```
[david@faile david]$ tmp.sh hello "how are you" today 1 2 3
param is hello
param is how are you
param is today
param is 1
param is 2
param is 3
[david@faile david]$ tmp.sh hello "how are you" today 1 2 3
param is hello how are you today 1 2 3
```

With the second example, where $* is used, the difference is obvious. The first example, where $@ is used, shows the advantage of $@. The second parameter is maintained as a single parameter.

The basics of input/output (IO)

We have already encountered the "**echo**" command, yet this is only the "O" part of IO − how can we get user input into our programs? We use the "**read**" command. For example:

```
#!/bin/bash
# FILE:           testread
read X
echo "You said $X"
```

The purpose of this enormously exciting program should be obvious.

Just in case you were bored with the `echo` command. Table 8.2 shows a few backslash characters that you can use to brighten your shell scripts:

Character	Purpose
\a	alert (bell)
\b	backspace
\c	don't display the trailing newline
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\nnn	the character with ASCII number nnn (octal)

Table 8.2
echo backslash options

(type "**man echo**" to see this exact table :)

To enable **echo** to interpret these backslash characters within a string, you must issue the **echo** command with a "**−e**" switch. You may also add a "**−n**" switch to stop **echo** printing a new−line at the end of the string − this is a good thing if you want to output a prompting string. For example:

```
#!/bin/bash
# FILE:           getname
echo −n "Please enter your name: "
read NAME
echo "Your name is $NAME"
```

(This program would be useful for those with a very short memory)

At the moment, we've only examined reading from STDIN (standard input a.k.a. the keyboard) and STDOUT (standard output a.k.a. the screen) − if we want to be really clever we can change this.

What do you think the following does?

```
read X < afile
```

or what about

```
echo $X > anotherfile
```

If you said that the first read the contents of `afile` into a variable **$X** and the second wrote the value of **$X** to `anotherfile` you'd almost be correct. The `read` operation will only read the first line (up to the end−of−line marker) from `afile` − it doesn't read the entire file.

You can also use the "**>>**" and "**<<**" redirection operators.

Exercises

What would you expect:

read X << END

would do? What do you think **$X** would hold if the input was:

Dear Sir
I have no idea why your computer blew up.
Kind regards, me.
END

And now for the hard bits

Scenario

So far we have been dealing with very simple examples − mainly due to the fact we've been dealing with very simple commands. Shell scripting was not invented so you could write programs that ask you your name then display it. For this reason, we are going to be developing a real program that has a useful purpose. We will do this section by section as we examine more shell programming concepts. While you are reading each section, you should consider how the information could assist in writing part of the program.

The actual problem is as follows:

You've been appointed as a system administrator to an academic department within a small (anonymous) regional university. The previous system administrator left in rather a hurry after it was found that department''s main server had being playing host to plethora of pornography, warez (pirate software) and documentation regarding interesting alternative uses for various farm chemicals.

There is some concern that the previous sys admin wasn't the only individual within the department who had been availing themselves to such wonderful and diverse resources on the Internet. You have been instructed to identify those persons who have been visiting "undesirable" Internet sites and advise them of the department's policy on accessing inappropriate material (apparently there isn't one, but you've been advised to improvise). Ideally, you will produce a report of people accessing restricted sites, exactly which sites and the number of times they visited them.

To assist you, a network monitoring program produces a datafile containing a list of users and sites they have accessed, an example of which is listed below:

```
FILE: netwatch

jamiesob    mucus.slime.com
tonsloye    xboys.funnet.com.fr
tonsloye    sweet.dreams.com
root  sniffer.gov.au
jamiesob    marvin.ls.tc.hk
jamiesob    never.land.nz
jamiesob    guppy.pond.cqu.edu.au
tonsloye    xboys.funnet.com.fr
tonsloye    www.sony.com
janesk      horseland.org.uk
```

```
root  www.nasa.gov
tonsloye    warez.under.gr
tonsloye    mucus.slime.com
root  ftp.ns.gov.au
tonsloye    xboys.funnet.com.fr
root  linx.fare.com
root  crackz.city.bmr.au
janesk      smurf.city.gov.au
jamiesob    mucus.slime.com
jamiesob    mucus.slime.com
```

After careful consideration (and many hours of painstaking research) a steering committee on the department's policy on accessing the internet has produced a list of sites that they have deemed "prohibited" − these sites are contained in a data file, an example of which is listed below:

```
FILE: netnasties


mucus.slime.com
xboys.funnet.com.fr
warez.under.gr
crackz.city.bmr.au
```

It is your task to develop a shell script that will fulfil these requirements (at the same time ignoring the privacy, ethics and censorship issues at hand :)

(Oh, it might also be an idea to get Yahoo! to remove the link to your main server under the /Computers/Software/Hackz/Warez/Sites listing... ;)

if ... then ... maybe?

Shell programming provides the ability to test the exit status from commands and act on them. One way this is facilitated is:

```
if command
then
  do other commands
fi
```

You may also provide an "alternate" action by using the "if" command in the following format:

```
if command
then
  do other commands
else
  do other commands
fi
```

And if you require even more complexity, you can issue the if command as:

```
if command
then
  do other commands
elif anothercommand
  do other commands
fi
```

To test these structures, you may wish to use the `true` and `false` UNIX commands. `true` always sets `$?` to 0 and `false` sets `$?` to 1 after executing.

Remember: **`if`** tests the exit code of a command − it isn't used to compare values; to do this, you must use the **`test`** command in combination with the **`if`** structure − **`test`** will be discussed in the next section.

What if you wanted to test the output of two commands? In this case, you can use the shell's && and `||` operators. These are effectively "smart" AND and OR operators.

The **`&&`** works as follows:

```
command1 && command2
```

`command2` will only be executed if **`command1`** succeeds.

The **`||`** works as follows:

```
command1 || command2
```

`command2` will only be executed if **`command1`** fails.

These are sometimes referred to as "short circuit" operators in other languages.

Given our problem, one of the first things we should do in our program is to check if our datafiles exist. How would we do this?

```
#!/bin/bash
# FILE:           scanit
if ls netwatch && ls netnasties
then
      echo "Found netwatch and netnasties!"
else
      echo "Can not find one of the data files − exiting"
      exit 1
fi
```

Exercise

> Enter the code above and run the program. Notice that the output from the **`ls`** commands (and the errors) appear on the screen − this isn't a very good thing. Modify the code so the only output to the screen is one of the **`echo`** messages.

Testing Testing...

Perhaps the most useful command available to shell programs is the **`test`** command. It is also the command that causes the most problems for first time shell programmers − the first program they ever write is usually (imaginatively) called `test` − they attempt to run it − and nothing happens −

why? (Hint: type `which test`, then type `echo $PATH` − why does the system command `test` run before the programmer's shell script?)

The test command allows you to:

- test the length of a string
- compare two strings
- compare two numbers
- check on a file's type
- check on a file's permissions
- combine conditions together

`test` actually comes in two flavours:

```
test an_expression
```

and

```
[ an_expression ]
```

They are both the same thing − it's just that `[` is soft−linked to `/usr/bin/test` ; `test` actually checks to see what name it is being called by; if it is [then it expects a] at the end of the expression.

What do we mean by "expression"? The expression is the string you want evaluated. A simple example would be:

```
if [ "$1" = "hello" ]
then
   echo "hello to you too!"
else
   echo "hello anyway"
fi
```

This simply tests if the first parameter was `hello`. Note that the first line could have been written as:

```
if test "$1" = "hello"
```

Tip: Note that we surrounded the variable $1 in quotes. This is to take care of the case when $1 doesn't exist − in other words, there were no parameters passed. If we had simply put $1 and there wasn't any $1, then an error would have been displayed:

```
test: =: unary operator expected
```

This is because you'd be effectively executing:

```
test NOTHING = "hello"
```

`=` expects a string to its left and right − thus the error. However, when placed in double quotes, you be executing:

```
test "" = "hello"
```

which is fine; you're testing an empty string against another string.

You can also use `test` to tell if a variable has a value in it by:

```
test $var
```

This will return true if the variable has something in it, false if the variable doesn't exist OR it contains null ("").

We could use this in our program. If the user enters at least one username to check on, them we scan for that username, else we write an error to the screen and exit:

```
if [ $1 ]
then
  the_user_list=echo $*
else
  echo "No users entered – exiting!
  exit 2
fi
```

Expressions, expressions!

So far we've only examined expressions containing string based comparisons. The following tables list all the different types of comparisons you can perform with the `test` command.

Expression	True if
`-z string`	length of string is 0
`-n string`	length of string is not 0
`string1 = string2`	if the two strings are identical
`string != string2`	if the two strings are NOT identical
`String`	if string is not NULL

Table 8.3
String based tests

Expression	True if
`int1 -eq int2`	first int is equal to second
`int1 -ne int2`	first int is not equal to second
`int1 -gt int2`	first int is greater than second
`int1 -ge int2`	first int is greater than or equal to second
`int1 -lt int2`	first int is less than second
`int1 -le int2`	first int is less than or equal to second

Table 8.4
Numeric tests

Expression	True if
`-r file`	`File` exists and is readable
`-w file`	`file` exists and is writable
`-x file`	`file` exists and is executable
`-f file`	`file` exists and is a regular file
`-d file`	`file` exists and is directory
`-h file`	`file` exists and is a symbolic link
`-c file`	`file` exists and is a character special file
`-b file`	`file` exists and is a block special file
`-p file`	`file` exists and is a named pipe
`-u file`	`file` exists and it is setuid
`-g file`	`file` exists and it is setgid
`-k file`	`file` exists and the sticky bit is set
`-s file`	`file` exists and its size is greater than 0

Table 8.5
File tests

Expression	Purpose
`!`	reverse the result of an expression
`-a`	AND operator
`-o`	OR operator
`( expr )`	group an expression, parentheses have special meaning to the shell so to use them in the test command you must quote them

Table 8.6
Logic operators with test

Remember: **`test`** uses different operators to compare strings and numbers − using **`-ne`** on a string comparison and **`!=`** on a numeric comparison is incorrect and will give undesirable results.

Exercise

> Modify the code for **`scanit`** so it uses the **`test`** command to see if the datafiles exists.

All about `case`

Ok, so we know how to conditionally perform operations based on the return status of a command. However, like a combination between the `if` statement and the `test $string = $string2`, there exists the **`case`** statement.

```
case value in
  pattern 1)      command
                anothercommand ;;
  pattern 2)      command
                  anothercommand ;;
esac
```

case works by comparing value against the listed patterns. If a match is made, then the commands associated with that pattern are executed (up to the "**;;**" mark) and **$?** is set to 0. If a match isn't made by the end of the case statement (**esac**) then **$?** is set to 1.

The really useful thing is that wildcards can be used, as can the "|" symbol which acts as an OR operator. The following example gets a Yes/No response from a user, but will accept anything starting with "Y" or "y" as YES, "N" or "n" as no and anything else as "MAYBE"

```
echo –n "Your Answer: "
read ANSWER
case $ANSWER in
  Y* | y*) ANSWER="YES" ;;
  N* | n*) ANSWER="NO" ;;
  *) ANSWER="MAYBE" ;;
esac
echo $ANSWER
```

Exercise

> Write a shell script that inputs a date and converts it into a long date form. For example:
> $~ > mydate 12/3/97
> 12th of March 1997
>
> $~ > mydate
> Enter the date: 1/11/74
> 1st of November 1974

Loops and Repeated Action Commands

Looping − "*the exciting process of doing something more than once*" − and shell programming allows it. There are three constructs that implement looping:

```
while – do – done
for – do – done
until – do – done
```

while

The format of the **while** construct is:

```
while command
do
  commands
done
```

(while `command` is true, `commands` are executed)

Example

```
while [ $1 ]
do
  echo $1
```

```
  shift
done
```

What does this segment of code do? Try running a script containing this code with **a b c d e** on the command line.

`while` also allows the redirection of input. Consider the following:

```
#!/bin/bash
# FILE:           linelist
#
count=0
while read BUFFER
do
  count=‘expr $count + 1‘     # Increment the count
  echo "$count $BUFFER"       # Echo it out
done < $1               # Take input from the file
```

This program reads a file line by line and echo’s it to the screen with a line number.

Given our **scanit** program, the following could be used read the netwatch datafile and compare the username with the entries in the datafile:

```
while read buffer
do
  user=‘echo $buffer | cut –d" " –f1‘
  site=‘echo $buffer | cut –d" " –f2‘
  if [ "$user" = "$1" ]
  then
    echo "$user visited $site"
  fi
done < netwatch
```

Exercise

> Modify the above code so that the site is compared with all sites in the prohibited sites file (**netnasties**). Do this by using another **while** loop. If the user has visited a prohibited site, then **echo** a message to the screen.

for

The format of the **for** construct is:

```
for variable in list_of_variables
do
  commands
done
```

(for each value in list_of_variables, "commands" are executed)

Example

We saw earlier in this chapter examples of the for command showing the difference between $* and $@.

Another example

```
for count in 10 9 8 7 6 5 4 3 2 1
do
  echo –n "$count.."
done

echo
```

Modifying *scanit*

Given our **scanit** program, we might wish to report on a number of users. The following modifications will allow us to accept and process multiple users from the command line:

```
for checkuser in $*
do
  while read buffer
  do
    while read checksite
    do
      user=‘echo $buffer | cut −d" " −f1‘
      site=‘echo $buffer | cut −d" " −f2‘
      if [ "$user" = "$checkuser" −a "$site" = "$checksite" ]
      then
        echo "$user visited the prohibited site $site"
      fi
    done < netnasties
  done < netwatch
done
```

Problems with running scanit

A student in the 1999 offering of 85321 reported the following problem with the scanit program on page 160 of chapter 8 of the 85321 textbook.

When running her program she types

bash scanit jamiesob

and quite contrary to expectations she gets 80 lines of output that includes

```
root visited the prohibited site crackz.city.bmr.au
root visited the prohibited site crackz.city.bmr.au
janesk visited the prohibited site smurf.city.gov.au
janesk visited the prohibited site smurf.city.gov.au
janesk visited the prohibited site smurf.city.gov.au
janesk visited the prohibited site smurf.city.gov.au
jamiesob visited the prohibited site mucus.slime.com
jamiesob visited the prohibited site mucus.slime.com
```

If everything is working the output you should get is three lines of code reporting that the user jamiesob has visited the site mucus.slime.com.

So what is the problem?

Well let’s have a look at her shell program

```
for checkuser in $*
do
  while read buffer
  do
    while read checksite
    do
      user=‘echo $buffer | cut −d" " −f1‘
      site=‘echo $buffer | cut −d" " −f2‘
      if [ "$user"="$checkuser" −a "$site"="$checksite"
]
      then
        echo "$user visited the prohibited site $site"
      fi
    done < netnasties
  done < netwatch
done
```

Can you see the problem?

How do we identify the problem? Well let's start by thinking about what the problem is. The problem is that it is showing too many lines. The script is not excluding lines which should not be displayed. Where are the lines displayed?

The only place is within the if command. This seems to imply that the problem is that the if command isn't working. It is matching too many times, in fact it is matching all of the lines.

The problem is that if command is wrong or not working as expected.

How is it wrong?

Common mistakes with the if command include

- not using the test command
  Some people try comparing "things" without using the test command
  if "$user"="$checkuser" −a "$site"="$checksite"
  The student is using the test command in our example. In fact, she is using the [form of the test command. So this isn't the problem.
- using the wrong comparison operator
  Some people try things like
  if ["$user" == "$checkuser"] or
  if ["$user" −eq "$checkuser"]
  Trouble with this is that == is comparison operator from the C/C++ programming languages and not a comparison operator supported by the test command. −eq is a comparison operator supported by test but it is used to compare numbers not strings. This isn't the problem here.

The problem here is some missing spaces around the = signs.

Remember that [is actually a shell command (it's the same command test). Like other commands it takes parameters. Let's have a look at the parameters that the test command takes in this example program.

The test command is
```
[ "$user"="$checkuser" −a "$site"="$checksite" ]
```
Parameters must be surrounded by spaces. So this command has four parameters (the first [is the command name)

1. `"$user"="$checkuser"`
2. `−a`
3. `"$site"="$checksite"`
4. `]`

By now you might start to see the problem. For the test command to actual compare two "things" it needs to see the = as a separate parameter. The problem is that because there are no spaces around the = characters in this test command the = is never seen. It's just part of a string.

The solution to this problem is to put some space characters around the two =. So we get
```
[ "$user" = "$checkuser" −a "$site" = "$checksite" ]
```

So what is happening

So what is actually happening? Why is the test always returning true. We know this because the script displays a line for all the users and all the sites.

To find the solution to this problem we need to take a look at the manual page for the test command. On current Linux computers you can type man test and you will see a manual page for this command. However, it isn't the one you should look at.

Type the following command which test. It should tell you where the executable program for test is located. Trouble is that on current Linux computers it won't. That's because there isn't one. Instead the test command is actually provided by the shell, in this case bash. To find out about the test command you need to look at the man page for bash.

The other approach would be to look at Table 8.3 from chapter 8 of the 85321 textbook. In particular the last entry which says that if the expression in a test command is a string then the test command will return true if the string is non−zero (i.e. it has some characters).

Here are some examples to show what this actually means.

In these examples I'm using the test command by itself and then using the echo command to have a look at the value of the $? shell variable. The $? shell variable holds the return status of the previous command.

For the test command if the return status is 0 then the expression was true. If it is 1 then the expression as false.

```
[david@faile 8]$ [ fred ]
[david@faile 8]$ echo $?
0
[david@faile 8]$ [ ]
[david@faile 8]$ echo $?
1
[david@faile 8]$ [ "jamiesob"="mucus.slime.com" ]
[david@faile 8]$ echo $?
0
[david@faile 8]$ [ "jamiesob" = "mucus.slime.com" ]
[david@faile 8]$ echo $?
1
```

In the first example the expression is fred a string with a non−zero length. So the return status is 0 indicating true. In the second example there is no expression, so it is a string with zero length. So the return status is 1 indicating false.

The last two examples are similar to the problem and solution in the student's program. The third example is similar to the students problem. The parameter is a single non−zero length string ("jamiesob"="mucus.slime.com") so the return status is 0 indicating truth.

When we add the spaces around the = we finally get what we wanted. The test command actually compares the two strings and sets the return status accordingly and because the strings are different the return status is 1 indicating false.

So what about the −a operator used in the student's program. Well the −a simply takes the results of two expressions (one on either side) and ands them together. In the student's script there the two expressions are non−zero length strings. Which are always true. So that becomes 0 −a 0 (TRUE and TRUE) which is always true.

Here are some more examples

```
 [david@faile 8]$ [ "jamiesob"="mucus.slime.com" -a
"david"="fred" ]
 [david@faile 8]$ echo $?
 0
 [david@faile 8]$ [ "jamiesob"="mucus.slime.com" -a "" ]
 [david@faile 8]$ echo $?
 1
 [david@faile 8]$ [ "jamiesob" = "mucus.slime.com" -a
"david" = "david" ]
 [david@faile 8]$ echo $?
 1
 [david@faile 8]$ [ "jamiesob" = "jamiesob" -a "david" =
"david" ]
 [david@faile 8]$ echo $?
 0
```

The first example here is what is happening in the student's program. Two non−zero length strings, which are always true, "anded" together will always return true regardless of the strings.

The second example shows what happens when one side of the −a is a zero length string. A zero length string is always false, false and true is always false, so this example has a return status of 1 indicating false.

The last two examples show "working" versions of the test command with spaces in all the right places. Where the two strings being compared are different the comparison is false and the test command is returning false. Where the two strings being compared are the same the comparison operator is returning true and the test command is returning true.

Exercises

What will be the return status of the following `test` commands? Why?

```
["hello"]
[ $HOME ]
[ "'hello'" ]
```

The above code is very inefficient IO wise − for every entry in the **`netwatch`** file, the entire **`netnasties`** file is read in. Modify the code so that the while loop reading the netnasties file is replaced by a for loop. (Hint: what does:

```
BADSITES='cat netnasties'
```

do?)

EXTENSION: What other IO inefficiencies does the code have? Fix them.

Speed and shell scripts

Exercise 8.11 is actually a very important problem in that it highlights a common mistake made by many novice shell programmers. This mistake is especially prevalent amongst people who have experience in an existing programming language like C/C++ or Pascal.

This supplementary material is intended to address that problem and hopefully make it a little easier for you to answer question 11. Online lecture 8,

particularly on slide 21 also addresses this problem. You might want to have a look at and listen to this slide before going much further.

What's the mistake

A common mistake for beginning shell programmers make is to write shell programs as if they were C/C++ programs. In particular they tend not to make use of the collection of very good commands which are available.

Let's take a look at a simple example of what I mean. The problem is to count the number of lines in a file (the file is called the_file). The following section discusses three solutions to this problem

1. A solution in C
2. A shell solution written like the C program.
3. A "proper" shell/UNIX solution

Solution in C

```
#include <stdio.h>

void main( void )
{
  int line_count = 0;
  FILE *infile;
  char line[500];

  infile = fopen( "the_file", "r" );

  while ( ! feof( infile ) )
  {
    fgets( line, 500, infile );
    line_count++;
  }
  printf( "Number of lines is %d\n", line_count-1 );
}
```

Pretty simple to understand? Open the file, read the file line by line, increment a variable for each line and then display the variable when we reach the end of the file.

Shell solution written by C programmer

It is common for new comers to the shell to write shell scripts like C (or whatever procedural language they are familiar with) programs. Here's a shell version of the previous C solution. It uses the same algorithm.

```
count=0
while read line
do
  count=‘expr $count + 1‘
done < the_file

echo Number of lines is $count
```

This shell script reads the file line by line, increment a variable for each line and when we reach the end of the file display the value.

Shell solution by shell programmer

Anyone with a modicum of UNIX experience will know that you don't need to write a shell program to solve this problem. You just use the wc command.

```
wc -l the_file
```

This may appear to be a fairly trivial example. However, it does emphasise a very important point. You don't want to use the shell commands like a normal procedural programming language. You want to make use of the available UNIX commands where ever possible.

Comparing the solutions

Let's compare the solutions.

The C program is obviously the longest solution when it comes to size of the program. The shell script is much shorter. The shell takes care of a lot of tasks you have to do with C and the use of wc is by far the shortest. The UNIX solutions are also much faster to write as there is no need for a compile/test cycle. This is one of the advantages of scripting languages like the shell, Perl and TCL.

What about speed of execution?

As we've seen in earlier chapters you can test the speed of executable programs (in a very coarse way) with the time command. The following shows the time taken for each solution. In the tests each of the three solutions worked on the same file which contained 1911 lines.

```
[david@faile david]$ time ./cprogram
Number of lines is 1911
0.00user 0.01system 0:00.01elapsed 83%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (79major+11minor)pagefaults 0swaps

[david@faile david]$ time sh shsolution
Number of lines is 1911
12.24user 14.17system 0:28.12elapsed 93%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (164520major+109070minor)pagefaults 0swaps

[david@faile david]$ time wc -l /var/log/messages
   1911 /var/log/messages
0.00user 0.01system 0:00.04elapsed 23%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (85major+14minor)pagefaults 0swaps
```

The lesson to draw from these figures is that solutions using the C program and the wc command have the same efficiency but using the wc command is much quicker.

The shell programming solution which was written like a C program is horrendously inefficient. It is tens of thousands of times slower than the other two solutions and uses an enormous amount of resources.

The problem

Obviously using while loops to read a file line by line in a shell program is inefficient and should be avoided. However, if you think like a C programmer you don't know any different.

When writing shell programs you need to modify how you program to make use of the strengths and avoid the weaknesses of shell scripting. Where possible you should use existing UNIX commands.

A solution for scanit?

Just because the current implementation of scanit uses two while loops it

doesn’t mean that your solution has to. Think about the problem you have to solve.

In the case of improving the efficiency of scanit you have to do the following

- for every user entered as a command line parameter
- see if the user has visited one of the sites listed in the netnasties file

To word it another way, you are searching for lines in a file which match a certain criteria. What UNIX command does that?

Number of processes

Another factor to keep in mind is the number of processes your shell script creates. Every UNIX command in a shell script will create a new process. Creating a new process is quite a time and resource consuming job performed by the operating system. One thing you want to do is to reduce the number of new processes created.

Let’s take a look at the shell program solution to our problem

```
count=0
while read line
do
  count=‘expr $count + 1‘
done < the_file

echo Number of lines is $count
```

For a file with 1911 lines this shell program is going to create about 1913 processes. 1 process for the echo command at the end, one process to for a new shell to run the script and 1911 processes for the expr command. Every time the script reads a line it will create a new process to run the expr command. So the longer the file the less efficient this script is going to get.

One way to address this problem somewhat is to use the support that the bash shell provides for arithmetic. By using the shell’s arithmetic functions we can avoid creating a new process because the shell process will do it.

Our new shell script looks like this

```
count=0
while read line
do
  count=$[ $count + 1 ]
done < /var/log/messages

echo Number of lines is $count
```

See the change in the line incrementing the count variable. It’s now using the shell arithmetic support. Look what happens to the speed of execution.

```
[david@faile 8]$ time bash test6
Number of lines is 1915
1.28user 0.52system 0:01.83elapsed 98%CPU (0avgtext+0avgdata
0maxresident)k
0inputs+0outputs (179major+30minor)pagefaults 0swaps
```

We have a slightly bigger file but even so the speed is much, much better. However, the speed is still no where as good as simply using the wc command.

until

The format of the **`until`** construct is:

```
until command
do
  commands
done
```

("commands" are executed until "command" is true)

Example

```
until [ "$1" = "" ]
do
  echo $1
  shift
done
```

`break` and `continue`

Occasionally you will want to jump out of a loop; to do this you need to use the **`break`** command. **`break`** is executed in the form:

```
break
```

or

```
break n
```

The first form simply stops the loop, for example:

```
while true
do
  read BUFFER
  if [ "$BUFFER" = "" ]
  then
    break
  fi
  echo $BUFFER
done
```

This code takes a line from the user and prints it until the user enters a blank line. The second form of **`break`**, **`break n`** (where **`n`** is a number) effectively works by executing **`break`** "**`n`**" times. This can break you out of embedded loops, for example:

```
for file in $*
do
  while read BUFFER
  do
    if [ "$BUFFER" = "ABORT" ]
    then
      break 2
    fi
  echo $BUFFER
  done < $file
done
```

This code prints the contents of multiple files, but if it encounters a line containing the word "**`ABORT`**" in any one of the files, it stops processing.

Like **`break`**, **`continue`** is used to alter the looping process. However, unlike **`break`**, **`continue`** keeps the looping process going; it just fails to

finish the remainder of the current loop by returning to the top of the loop. For example:

```
while read BUFFER
do
  charcount=‘echo $BUFFER | wc −c | cut −f1‘
  if [ $charcount −gt 80 ]
  then
    continue
  fi
  echo $BUFFER
done < $1
```

This code segment reads and echo’s the contents of a file − however, it does not print lines that are over 80 characters long.

Redirection

Not just the `while − do − done` loops can have IO redirection; it is possible to perform piping, output to files and input from files on `if`, `for` and `until` as well. For example:

```
if true
then
  read x
  read y
  read x
fi < afile
```

This code will read the first three lines from `afile`. Pipes can also be used:

```
read BUFFER
while [ "$BUFFER" != "" ]
do
  echo $BUFFER
  read BUFFER
done | todos > tmp.$$
```

This code uses a non−standard command called `todos`. `todos` converts UNIX text files to DOS textfiles by making the EOL (End−Of−Line) character equivalent to CR (Carriage−Return) LF (Line−Feed). This code takes STDIN (until the user enters a blank line) and pipes it into todos, which in turn converts it to a DOS style text file (**`tmp.$$`**) . In all, a totally useless program, but it does demonstrate the possibilities of piping.

Now for the really hard bits

Functional Functions

A symptom of most usable programming languages is the existence of functions. Theoretically, functions provide the ability to break your code into reusable, logical compartments that are the by product of top−down design. In practice, they vastly improve the readability of shell programs, making it easier to modify and debug them.

An alternative to functions is the grouping of code into separate shell scripts and calling these from your program. This isn’t as efficient as functions, as functions are executed in the same process that they were called from; however other shell programs are launched in a separate process space − this is inefficient on memory and CPU resources.

You may have noticed that our **`scanit`** program has grown to around 30 lines of code. While this is quite manageable, we will make some major changes later that really require the "modular" approach of functions.

Shell functions are declared as:

```
function_name()
{
  somecommands
}
```

Functions are called by:

```
function_name parameter_list
```

YES! Shell functions support parameters. **$1** to **$9** represent the first nine parameters passed to the function and **$*** represents the entire parameter list. The value of **$0** isn't changed. For example:

```
#!/bin/bash
# FILE:         catfiles

catfile()
{
  for file in $*
  do
    cat $file
  done
}

FILELIST=‘ls $1‘
cd $1

catfile $FILELIST
```

This is a highly useless example (**cat *** would do the same thing) but you can see how the "main" program calls the function.

local

Shell functions also support the concept of declaring "local" variables. The `local` command is used to do this. For example:

```
#!/bin/bash

testvars()
{
  local localX="testvars localX"
  X="testvars X"
  local GlobalX="testvars GlobalX"
  echo "testvars: localX= $localX X= $X GlobalX= $GlobalX"
}

X="Main X"
GlobalX="Main GLobalX"
echo "Main 1: localX= $localX X= $X GlobalX= $GlobalX"

testvars

echo "Main 2: localX= $localX X= $X GlobalX= $GlobalX"
```

The output looks like:

```
Main 1: localX=  X= Main X GlobalX= Main GLobalX
testvars: localX= testvars localX X= testvars X GlobalX= testvars GlobalX
Main 2: localX=  X= testvars X GlobalX= Main GLobalX
```

The return trip

After calling a shell function, the value of **$?** is set to the exit status of the last command executed in the shell script. If you want to explicitly set this, you can use the **return** command:

```
return n
```

(Where **n** is a number)

This allows for code like:

```
if function1
then
  do_this
else
  do_that
fi
```

For example, we can introduce our first function into our **scanit** program by placing our datafile tests into a function:

```
#!/bin/bash
# FILE:           scanit
#

check_data_files()
{
  if [ −r netwatch −a −r netnasties ]
  then
    return 0
  else
    return 1
  fi
}

# Main Program

if check_data_files
then
  echo "Datafiles found"
else
  echo "One of the datafiles missing − exiting"
  exit 1
fi

# our other work...
```

Difficult and not compulsory

The following section (up to the section titled "Bugs and Debugging") is not compulsory for students studying 85321.

Recursion: (see "Recursion")

Shell programming even supports recursion. Typically, recursion is used to process tree−like data structures − the following example illustrates this:

```
#!/bin/bash
# FILE:           wctree

wcfiles()
{
  local BASEDIR=$1            # Set the local base directory
  local LOCALDIR=‘pwd‘  # Where are we?
  cd $BASEDIR                 # Go to this directory (down)
  local filelist=‘ls‘         # Get the files in this directory
  for file in $filelist
  do
    if [ −d $file ]           # If we are a directory, recurs
    then
      # we are a directory
      wcfiles "$BASEDIR/$file"
    else
      fc=‘wc −w < $file‘      # do word count and echo info
```

```
      echo "$BASEDIR/$file $fc words"
    fi
  done
  cd $LOCALDIR          # Go back up to the calling directory
}

if [ $1 ]               # Default to . if no parms
then
  wcfile $1
else
  wcfile "."
fi
```

Exercise

> What does the `wctree` program do? Why are certain variables declared as `local`? What would happen if they were not? Modify the program so it will only "recurs" 3 times.
>
> **EXTENSION:** There is actually a UNIX command that will do the same thing as this shell script – what is it? What would be the command line? (Hint: **`man find`**)

***`wait`*'ing and *`trap`*'ing**

So far we have only examined linear, single process shell script examples. What if you want to have more than one action occurring at once? As you are aware, it is possible to launch programs to run in the background by placing an ampersand behind the command, for example:

```
runcommand &
```

You can also do this in your shell programs. It is occasionally useful to send a time consuming task to the background and proceed with your processing. An example of this would be a sort on a large file:

```
sort $largefile > $newfile &
do_a_function
do_another_funtion $newfile
```

The problem is, what if the sort hadn't finished by the time you wanted to use $newfile? The shell handles this by providing **`wait`** :

```
sort $largefile > $newfile &
do_a_function
wait
do_another_funtion $newfile
```

When **`wait`** is encountered, processing stops and "waits" until the child process returns, then proceeds on with the program. But what if you had launched several processes in the background? The shell provides the shell variable **`$!`** (the PID of the child process launched) which can be given as a parameter to wait – effectively saying "wait for this PID". For example:

```
sort $largefile1 > $newfile1 &
$SortPID1=$!
sort $largefile2 > $newfile2 &
$SortPID2=$!
sort $largefile3 > $newfile3 &
$SortPID3=$!
do_a_function
wait $SortPID1
do_another_funtion $newfile1
wait $SortPID2
do_another_funtion $newfile2
wait $SortPID3
do_another_funtion $newfile3
```

Another useful command is `trap`. `trap` works by associating a set of commands with a signal from the operating system. You will probably be familiar with:

```
kill –9 PID
```

which is used to kill a process. This command is in fact sending the signal "9" to the process given by PID. Available signals are shown in Table 8.7.

Signal	Meaning
0	Exit from the shell
1	Hangup
2	Interrupt
3	Quit
4	Illegal Instruction
5	Trace trap
6	IOT instruction
7	EMT instruction
8	Floating point exception
10	Bus error
12	Bad argument
13	Pipe write error
14	Alarm
15	Software termination signal

Table 8.7
UNIX signals

(Taken from "*UNIX Shell Programming*" Kochan et al)

While you can't actually trap signal 9, you can trap the others. This is useful in shell programs when you want to make sure your program exits gracefully in the event of a shutdown (or some such event) (often you will want to remove temporary files the program has created). The syntax of using `trap` is:

```
trap commands signals
```

For example:

```
trap "rm /tmp/temp.$$" 1 2
```

will trap signals 1 and 2 − whenever these signals occur, processing will be suspended and the **`rm`** command will be executed.

You can also list every trap'ed signal by issuing the command:

```
trap
```

To "un−trap" a signal, you must issue the command:

```
trap "" signals
```

The following is a somewhat clumsy form of IPC (Inter−Process Communication) that relies on **`trap`** and **`wait`**:

```
#!/bin/bash
# FILE:     saymsg
# USAGE: saymsg <create number of children> [total number of
#        children]

readmsg()
{
  read line < $$ # read a line from the file given by the PID
```

```
  echo "$ID − got $line!"     # of my *this* process ($$)
  if [ $CHILD ]
  then
    writemsg $line      # if I have children, send them message
  fi
}

writemsg()
{
  echo $* > $CHILD      # Write line to the file given by PID
  kill −1 $CHILD        # of my child.  Then signal the child.
}

stop()
{
  kill −15 $CHILD       # tell my child to stop
  if [ $CHILD ]
  then
    wait $CHILD         # wait until they are dead
    rm $CHILD                 # remove the message file
  fi
  exit 0
}


# Main Program

if [ $# −eq 1 ]
then
  NUMCHILD=‘expr $1 − 1‘
  saymsg $NUMCHILD $1 & # Launch another child
  CHILD=$!
  ID=0
  touch $CHILD                # Create empty message file
  echo "I am the parent and have child $CHILD"
else
  if [ $1 −ne 0 ]       # Must I create children?
  then
    NUMCHILD=‘expr $1 − 1‘    # Yep, deduct one from the number
    saymsg $NUMCHILD $2 &     # to be created, then launch them
    CHILD=$!
    ID=‘expr $2 − $1‘
    touch $CHILD        # Create empty message file
    echo "I am $ID and have child $CHILD"
  else
    ID=‘expr $2 − $1‘         # I don’t need to create children
    echo "I am $ID and am the last child"
```

```
  fi
fi

trap "readmsg" 1        # Trap the read signal
trap "stop" 15                # Trap the drop-dead signal

if [ $# -eq 1 ]         # If I have one parameter,
then                    # then I am the parent - I just read
  read BUFFER                 # STDIN and pass the message on
  while [ "$BUFFER" ]
  do
    writemsg $BUFFER
    read BUFFER
  done
  echo "Parent - Stopping"
  stop
else                    # Else I am the child who does nothing -
  while true                  # I am totally driven by signals.
  do
    true
  done
fi
```

So what is happening here? It may help if you look at the output:

```
psyche:~/sanotes[david@faile david]$ saymsg 3
I am the parent and have child 8090
I am 1 and have child 8094
I am 2 and have child 8109
I am 3 and am the last child
this is the first thing I type
1 - got this is the first thing I type!
2 - got this is the first thing I type!
3 - got this is the first thing I type!

Parent - Stopping

psyche:~/sanotes[david@faile david]$
```

Initially, the parent program starts, accepting a number of children to create. The parent then launches another program, passing it the remaining number of children to create and the total number of children. This happens on every launch of the program until there are no more children to launch.

From this point onwards the program works rather like Chinese whispers - the parent accepts a string from the user which it then passes to its child by sending a signal to the child - the signal is caught by the child and **`readmsg`** is executed. The child writes the message to the screen, then passes the message to its child (if it has one) by signalling it and so on and so on. The messages are passed by being written to files - the parent writes the message into a file named by the PID of the child process.

When the user enters a blank line, the parent process sends a signal to its child - the signal is caught by the child and **`stop`** is executed. The child then sends a message to its child to stop, and so on and so on down the line. The parent process can't exit until all the children have exited.

This is a very contrived example − but it does show how processes (even at a shell programming level) can communicate. It also demonstrates how you can give a function name to **`trap`** (instead of a command set).

Exercise

> **`saymsg`** is riddled with problems − there isn't any checking on the parent process command line parameters (what if there wasn't any?) and it isn't very well commented or written − make modifications to fix these problems. What other problems can you see?
>
> **EXTENSION**: Fundamentally **`saymsg`** isn't implementing very safe inter−process communication − how could this be fixed? Remember, one of the *main* problems concerning IPC is the race condition − could this happen?

Bugs and Debugging

If by now you have typed every example program in, completed every exercise and have not encountered one single error then you are a truly amazing person. However, if you are like me, you would have made at least 70 billion mistakes/ typos or TSE's (totally stupid errors) − and now I tell you the easy way to find them!

Method 1 − `set`

Issuing the truly inspired command of:

```
set −x
```

within your program will do wonderful things. As your program executes, each code line will be printed to the screen − that way you can find your mistakes, err, well, a little bit quicker. Turning tracing off is a good idea once your program works − this is done by:

```
set +x
```

Method 2 −

echo

Placing a few **echo** statements in your code during your debugging is one of the easiest ways to find errors − for the most part this will be the quickest way of detecting if variables are being set correctly.

Very Common Mistakes

```
$VAR=‘ls‘
```

This should be `VAR=‘ls‘`. When setting the value of a shell variable you don’t use the $ sign.

```
read $BUFFER
```

The same thing here. When setting the value of a variable you don’t use the $ sign.

```
VAR=‘ls −al"
```

The second ‘ is missing

```
if [ $VAR ]
then
    echo $VAR
fi
```

Haven’t specified what is being tested here. Need to refer to the contents of Tables 8.2 through 8.5

```
if [ $VAR −eq $VAR2 ]
then
   echo $VAR
fi
```

If $VAR and $VAR2 are strings then you can’t use `−eq` to compare their values. You need to use =.

```
if [ $VAR = $VAR2 ] then
  echo $VAR
fi
```

The `then` must be on a separate line.

And now for the really really hard bits

Writing good shell programs

We have covered most of the theory involved with shell programming, but there is more to shell programming than syntax. In this section, we will complete the **scanit** program, examining efficiency and structure considerations.

scanit currently consists of one chunk of code with one small function. In its current form, it doesn’t meet the requirements specified:

"...you will produce a report of people accessing restricted sites, exactly which sites and the number of times they visited them."

Our code, as it is, looks like:

```
#!/bin/bash
# FILE:           scanit
#

check_data_files()
{
  if [ −r netwatch −a −r netnasties ]
  then
    return 0
  else
    return 1
  fi
}

# Main Program

if check_data_files
then
  echo "Datafiles found"
else
  echo "One of the datafiles missing − exiting"
  exit 1
fi

for checkuser in $*
do
  while read buffer
  do
    while read checksite
    do
      user=‘echo $buffer | cut −d" " −f1‘
      site=‘echo $buffer | cut −d" " −f2‘
      if [ "$user" = "$checkuser" −a "$site" = "$checksite" ]
      then
        echo "$user visited the prohibited site $site"
      fi
    done < netnasties
  done < netwatch
done
```

At the moment, we simply print out the user and site combination − no count provided. To be really effective, we should parse the file containing the user/site combinations (**`netwatch`**), register and user/prohibited site combinations and then when we have all the combinations and count per combination, produce a report. Given our datafile checking function, the pseudo code might look like:

```
if data_files_exist
  ...
else
  exit 1
fi
check_netwatch_file
produce_report
exit
```

It might also be an idea to build in a "default" − if no username(s) are given on the command line, we go and get all the users from the **`/etc/passwd`** file:

```
f [ $1 ]
then
  the_user_list=$*
else
  get_passwd_users
fi
```

Exercise

Write the shell function **`get_passwd_users`**. This function goes through the **`/etc/passwd`** file and creates a list of usernames. (Hint: username is field one of the password file, the delimiter is "**`:`**")

`eval` the wonderful!

The use of `eval` is perhaps one of the more difficult concepts in shell programming to grasp is the use of **`eval`**. **`eval`** effectively says "parse (or execute) the following twice". What this means is that any shell variables that appear in the string are "substituted" with their real value on the first parse, then used as−they−are for the second parse.

The use of this is difficult to explain without an example, so we'll refer back to our case study problem.

The real challenge to this program is how to actually store a count of the user and site combination. The following is how I'd do it:

```
checkfile()
{
  # Goes through the netwatch file and saves user/site
  # combinations involving sites that are in the "restricted"
  # list

  while read buffer
  do
    username=‘echo $buffer | cut −d" " −f1‘     # Get the username
    # Remove “.”’s from the string
    site=‘echo $buffer | cut −d" " −f2 | sed s/\\\.//g‘
    for checksite in $badsites
    do
      checksite=‘echo $checksite | sed s/\\\.//g‘
      # Do this for the compare sites
      if [ "$site" = "$checksite" ]
      then
        usersite="$username$checksite"
        # Does the VARIABLE called $usersite exist? Note use of eval
        if eval [ \$$usersite ]
        then
          eval $usersite=\‘expr \$$usersite + 1\‘
        else
          eval $usersite=1
        fi
      fi
    done
  done < netwatch
}
```

There are only two *really* tricky lines in this function:

```
1.    site=‘echo $buffer | cut −d" " −f2 | sed s/\\\.//g‘
```

Creates a variable `site`; if buffer (one line of netwatch) contained

```
rabid.dog.com
```

then `site` would become:

```
rabiddogcom
```

The reason for this is because of the variable `usersite`:

```
usersite="$username$checksite"
```

What we are actually creating is a variable name, stored in the variable usersite − why (you still ask) did we remove the "."'s? This becomes clearer when we examine the second tricky line:

```
2.    eval $usersite=\‘expr \$$usersite + 1\‘
```

Remember eval "double" or "pre" parses a line − after eval has been run, you get a line which looks something like:

```
# $user="jamiesobrabiddogcom"
jamiesobrabiddogcom=‘expr $jamiesobrabiddogcom + 1‘
```

What should become clearer is this: the function reads each line of the `netwatch` file. If the site in the `netwatch` file matches one of the sites stored in `netnasties` file (which has been cat'ed into the variable `badsites`) then we store the user/site combination. We do this by first checking if there exists a variable by the name of the user/site combination − if one does exist, we add 1 to the value stored in the variable. If there wasn't a variable with the name of the user/site combination, then we create one by assigning it to "1".

At the end of the function, we should have variables in memory for all the user/prohibited site combinations found in the netwatch file, something like:

```
jamiesobmucusslimecom=3
tonsloyemucusslimecom=1
tonsloyeboysfunnetcomfr=3
tonsloyewarezundergr=1
rootwarzundergr=4
```

Note that this would be the case even if we were only interested in the users `root` and `jamiesob`. So why didn't we check in the function if the user in the netwatch file was one of the users we were interested in? Why should we!? All that does is adds an extra loop:

```
for every line in the file
  for every site
     for every user
     do check
     create variable if user and if site in userlist,
     badsitelist
```

whereas all we have now is

```
for every line in the file
    for every site
       create variable if site in badsitelist
```

We are still going to have to go through every user/badsite combination anyway when we produce the report − why add the extra complexity?

You might also note that there is minimal file IO − datafiles are only read ONCE − lists (memory structures) may be read more than once.

Exercise

Given the `checksite` function, write a function called `produce_report` that accepts a list of usernames and finds all user/badsite combinations stored by `checkfile`. This function should `echo` lines that look something like:

```
jamiesob: mucus.slime.com 3
tonsloye: mucus.slime.com 1
tonsloye: xboys.funnet.com.fr 3
tonsloye: warez.under.gr 1
```

Step−by−step

In this section, we will examine a complex shell programming problem and work our way through the solution.

The problem

This problem is an adaptation of the problem used in the 1997 shell programming assignment for systems administration:

Problem Definition

Your department''s FTP server provides anonymous FTP access to the **`/pub`** area of the filesystem − this area contains subdirectories (given by unit code) which contain resource materials for the various subjects offered. You suspect that this service isn't being used any more with the advent of the WWW, however, before you close this service and use the file space for something more useful, you need to prove this.

What you require is a program that will parse the FTP logfile and produce usage statistics on a given subject. This should include:

§ Number of accesses per user

§ Number of bytes transferred

§ The number of machines which have used the area.

The program will probably be called from other scripts. It should accept (from the command line) the subject (given by the subject code) that it is to examine, followed by one or more commands. Valid commands will consist of:

§ USERS − get a user and access count listing

§ BYTES − bytes transmitted for the subject

§ HOSTS − number of unique machines who have used the area

Background information

A cut down version of the FTP log will be examined by our program − it will consist of:

```
remote host name
file size in bytes
name of file
local username or, if guest, ID string given (anonymous FTP password)
```

For example:

```
aardvark.com   2345    /pub/85349/lectures.tar.gz    flipper@aardvark.com
138.77.8.8     112     /pub/81120/cpu.gif            sloth@topaz.cqu.edu.au
```

The FTP logfile will be called **`/var/log/ftp.log`** − we need not concern ourselves how it is produced (for those that are interested − look at **`man ftpd`** for a description of the real log file).

Anonymous FTP "usernames" are recorded as whatever the user types in as the password − while this may not be accurate, it is all we have to go on.

We can assume that all directories containing subject material branch off the **`/pub`** directory, eg.

```
/pub/85321
/pub/81120
```

Expected interaction

The following are examples of interaction with the program (**`scanlog`**):

```
[david@faile david]$ scanlog 85321 USERS
jamiesob@jasper.cqu.edu.au 1
b.spice@sworld.cqu.edu.au 22
jonesd 56

[david@faile david]$ scanlog 85321 BYTES
2322323

[david@faile david]$ scanlog 85321 HOSTS
5

[david@faile david]$ scanlog 85321 BYTES USERS
2322323
jamiesob@jasper.cqu.edu.au 1
b.spice@sworld.cqu.edu.au 22
jonesd 56
```

Solving the problem

How would you solve this problem? What would you do first?

Break it up

What does the program have to do? What are its major parts? Let's look at the functionality again − our program must:

- get a user and access count listing
- produce a the byte count on files transmitted for the subject
- list the number unique machines who have used the area and how many times

To do this, our program must first:

- Read parameters from the command line, picking out the subject we are interested in
- go through the other parameters one by one, acting on each one, calling the appropriate function
- Terminate/clean up

So, this looks like a program containing three functions. Or is it?

Look at the simple case first

It is often easier to break down a problem by walking through a simple case first.

Lets imagine that we want to get information about a subject − let''s use the code 85321. At this stage, we really don't care what the action is. What happens?

The program starts.

- We extract the first parameter from the command line. This is our subject. We might want to check if there is a first parameter − is it blank?
- Since we are only interested in this subject, we might want to go through the FTP log file and extract those entries we're interested in and keep this information in a temporary file. Our other option is to do this for every different "action" − this would probably be inefficient.
- Now, we want to go through the remaining parameters on the command line and act on each one. Maybe we should signal a error if we don't understand the action?
- At the end of our program, we should remove any temporary files we use.

Pseudo Code

If we were to pseudo code the above steps, we'd get something like:

```
# Check to see if the first parameter is blank
if  first_parameter = ""
then
  echo "No unit specified"
  exit
fi
```

```
# Find all the entries we're interested in, place this in a TEMPFILE
# Right − for every other parameter on the command line, we perform
# some

for ACTION in other_parameters
do
  # Decide if it is a valid action − act on it or give a error
done

# Remove Temp file
rm TEMPFILE


# Let's code this:
if [ "$1" = "" ]
then
  echo "No unit specified"
  exit 1
fi

# Remove $1 from the parm line

UNIT=$1
shift

# Find all the entries we're interested in
grep "/pub/$UNIT" $LOGFILE > $TEMPFILE

# Right − for every other parameter on the command line, we perform
some
for ACTION in $@
do
  process_action "$ACTION"
done

# Remove Temp file
rm $TEMPFILE
```

Ok, a few points to note:

- Notice the use of the variables **LOGFILE** and **TEMPFILE**? These would have to be defined somewhere above the code segment.
- We remove the first parameter from the command line and assign it to another variable. We do this using the shift command.
- We use **grep** to find all the entries in the original log file that refer to the subject we are interested in. We store these entries in a temporary file.
- The use of **$@** in the loop to process the remaining parameters is important. Why did we use it? Why not **$*** ? Hint: "1 2 3 4 5 6" isn't "1" "2" "3" "4" "5" "6" is it?
- We've invented a new function, **process_action** − we will use this to work out what to do with each action. Note that we are passing the function a parameter. Why are we enclosing it in quotes? Does it matter if we don't? Actually, in this case, it doesn't matter if we call the function with the parameter in quotes or not, as our parameters are expected to be single words. However, what if we allowed commands like:

  find host 138.77.3.4

  If we passed this string to a function (without quotes), it would be interpreted as:

  $1="find" $2="host" $3="138.77.3.4"

  This wouldn't be entirely what we want − so, we enclose the string in

quotes − producing:

$1="find host 138.77.3.4"

As we mentioned, in this case, we have single word commands, so it doesn't matter, however, always try to look ahead for problems − ask yourself the figurative question − "Is my code going to work in the rain?".

Expand function `process_action`

We have a function to work on − **`process_action`**. Again, we should pseudo code it, then implement it. Wait! We haven't first thought about what we want it to do − always a good idea to think before you code!

This function must take a parameter, determine if it is a valid action, then perform some action on it. It is an invalid action, then we should signal an error.

Let's try the pseudo code first:

```
process_action()
  {

    # Now, Check what we have
    case Action in
      BYTES then do a function to get bytes
      USERS then do a function to get a user list
      HOSTS then do a function to get an access count
      Something Else then echo "Unknown command $theAction"
    esac

  }
```

Right − now try the code:

```
process_action()
  {
    # Translate to upper case
    theAction=‘echo $1 | tr [a-z] [A-Z]‘

    # Now, Check what we have
    case $theAction in
      USERS) getUserList ;;
      HOSTS) getAccessCount ;;
      BYTES) getBytes ;;
      *) echo "Unknown command $theAction" ;;
    esac

  }
```

Some comments on this code:

- Note that we translate the "action command" (for example "bytes" , "users") into upper case. This is a nicety − it just means that we'll pick up every typing variation of the action.
- We use the case command to decide what to do with the action. We could have just as easily used a series of **`IF-THEN-ELSE-ELIF-FI`** statements − this becomes horrendous to code and read after about three conditions so case is a better option.
- As you will see, we've introduced calls to functions for each command − this again breaks to code up into bite size pieces (excuse the pun ;) to code. This follows the top−down design style.
- We will now expand each function.

Expand Function `getUserList`

Now might be a good time to revise what was required of our program − in particular, this function.

We need to produce a listing of all the people who have accessed files relating to the subject of interest and how many times they've accessed files.

Because we've separated out the entries of interest from the log file, we need no longer concern ourselves with the actual files and if they relate to the subject. We now are just interested in the users.

Reviewing the log file format:

```
aardvark.com       2345   /pub/85349/lectures.tar.gz
      flipper@aardvark.com
138.77.8.8   112    /pub/81120/cpu.gif              sloth@topaz.cqu.edu.au
```

We see that user information is stored in the fourth field. If we pseudo code what we want to do, it would look something like:

```
for every_user_in the file
do
  go_through_the_file_and_count_occurences
  print this out
done
```

Expanding this a bit more, we get:

```
extract_users_from_file
for user in user_list
do
  count = 0
  while read log_file
  do
    if user = current_entry
    then
      count = count + 1
    fi
  done
  echo user count
done
```

Let's code this:

```
getUserList()
  {
    cut −f4 $TEMPFILE | sort > $TEMPFILE.users
    userList=‘uniq $TEMPFILE.users‘

    for user in $userList
    do
    {
      count=0
      while read X
      do
        if echo $X | grep $user > /dev/null
        then
          count=‘expr $count + 1‘
        fi
      done
     } < $TEMPFILE
    echo $user $count
    done

    rm $TEMPFILE.users
  }
```

Some points about this code:

- The first cut extracts a user list and places it in a temp file. A unique list of users is then created and placed into a variable.
- For every user in the list, the file is read through and each line searched for the user string. We pipe the output into **/dev/null**.
- If a match is made, count is incremented.
- Finally the user/count combination is printed.
- The temporary file is deleted.

Unfortunately, this code totally sucks. Why?

There are several things wrong with the code, but the most outstanding problem is the massive and useless looping being performed − the **while** loop reads through the file for every user. This is bad. While loops within shell scripts are very time consuming and inefficient − they are generally avoided if, as in this case, they can be. More importantly, this script doesn't make use of UNIX commands which could simplify (and speed up!) our code. Remember: don't re−invent the wheel − use existing utilities where possible.

Let's try it again, this time without the while loop:

```
getUserList()
  {
    cut −f4 $TEMPFILE | sort > $TEMPFILE.users      # Get user list
    userList=‘uniq $TEMPFILE.users‘

    for user in $userList                           # for every user...
    do
      count=‘grep $user $TEMPFILE.users | wc −l‘ # count how many times they are
      echo $user $count                             # in the file
    done

    rm $TEMPFILE.users
  }
```

Much better! We've replaced the while loop with a simple grep command − however, there are still problems:

We don't need the temporary file

Can we wipe out a few more steps?

Next cut:

```
getUserList()
  {
    userList=‘cut −f4 $TEMPFILE | sort | uniq‘

    for user in $userList
    do
      echo $user ‘grep $user $TEMPFILE | wc −l‘
    done
  }
```

Beautiful!

Or is it.

What about:

```
echo ‘cut−f4 $TEMPFILE | sort | uniq −c‘
```

This does the same thing...or does it? If we didn't care what our output looked like, then this'd be ok − find out what's wrong with this code by trying it and

the previous segment − compare the results. Hint: **`uniq -c`** produces a count of every *sequential* occurrence of an item in a list. What would happen if we removed the **`sort`**? How could we fix our output "problem"?

Expand Function `getAccessCount`

This function requires a the total number of unique hosts which have accessed the files. Again, as we've already separated out the entries of interest into a temporary file, we can just concentrate on the hosts field (field number one).

If we were to pseudo code this:

```
create_unique_host list
count = 0
for host in host_list
do
   count = count + 1
done
echo count
```

From the previous function, we can see that a direct translation from pseudo code to shell isn't always efficient. Could we skip a few steps and try the efficient code first? Remember − we should try to use existing UNIX commands.

How do we create a unique list? The hint is in the word unique − the **`uniq`** command is useful in extracting unique listings.

What are we going to use as the input to the **`uniq`** command? We want a list of all hosts that accessed the files − the host is stored in the first field of every line in the file. Next hint − when we see the word "field" we can immediately assume we're going to use the **`cut`** command. Do we have to give **`cut`** any parameters? In this case, no. **`cut`** assumes (by default) that fields are separated by tabs − in our case, this is true. However, if the delimiter was anything else, we'd have to use a "**`-d`**" switch, followed by the delimiter.

Next step − what about the output from **`uniq`**? Where does this go? We said that we wanted a count of the unique hosts − another hint − counting usually means using the **`wc`** command. The **`wc`** command (or word count command) counts characters, words and lines. If the output from the **`uniq`** command was one host per line, then a count of the lines would reveal the number of unique hosts.

So what do we have?

```
cut –f1
uniq
wc –l
```

Right − how do we get input and save output for each command?

A first cut approach might be:

```
cat $TEMPFILE | cut –f1 > $TEMPFILE.cut
cat $TEMPFILE.cut | uniq > $TEMPFILE.uniq
COUNT=‘cat $TEMPFILE.uniq | wc –l‘
echo $COUNT
```

This is very inefficient; there are several reasons for this:

- We cat a file **THREE** times to get the count. We don't even have to use cat if we really try.
- We use temp files to store results − we could use a shell variable (as in the second last line) but is there any need for this? Remember, file IO is much

slower than assignments to variables, which, depending on the situation, is slower again that using pipes.

§ There are four lines of code − this can be completed in **one**!

So, removing these problems, we are left with:

```
getAccessCount()
  {
    echo ‘cut −f1 $TEMPFILE | uniq | wc −l‘
  }
```

How does this work?

§ The shell executes what’s between **‘‘** and this is outputted by **echo**.

§ This command starts with the **cut** command − a common misconception is that **cut** requires input to be piped into it − however, **cut** works just as well by accepting the name of a file to work with. The output from **cut** (a list of hosts) is piped into **uniq**.

§ **uniq** then removes all duplicate host from the list − this is piped into **wc**.

§ **wc** then counts the number of lines − the output is displayed.

Expand Function `getBytes`

The final function we have to write (Yes! We are nearly finished) counts the total byte count of the files that have been accessed. This is actually a fairly simple thing to do, but as you’ll see, using shell scripting to do this can be very inefficient.

First, some pseudo code:

```
total = 0
while read line from file
do
  extract the byte field
  add this to the total
done

echo total
```

In shell, this looks something like:

```
getBytes()
  {
    bytes=0
    while read X
    do
      bytefield=‘echo $X | cut −f2‘
      bytes=‘expr $bytes + $bytefield‘
    done < $TEMPFILE
    echo $bytes
  }
```

...which is very inefficient (remember: looping is bad!). In this case, every iteration of the loop causes three new processes to be created, two for the first line, one for the second − creating processes takes time!

The following is a bit better:

```
getBytes()
  {
    list=‘cut −f2  $TEMPFILE ‘
    bytes=0
    for number in $list
    do
      bytes=‘expr $bytes + $number‘
```

```
    done

    echo $bytes
  }
```

The above segment of code still has looping, but is more efficient with the use of a list of values which must be added up. However, we can get smarter:

```
getBytes()
  {
   numstr=‘cut −f2 $TEMPFILE | sed "s/$/ + /g"‘
   expr $numstr 0
  }
```

Do you see what we’ve done? The **cut** operation produces a list of numbers, one per line. When this is piped into **sed**, the end−of−line is substituted with **“ + “** − note the spaces. This is then combined into a single line string and stored in the variable **numstr**. We then get the **expr** of this string − why do we put the **0** on the end?

Two reasons:

After the **sed** operation, there is an extra **“+”** on the end − for example, if the input was:

2
3
4

The output would be:

2 +
3 +
4 +

This, when placed in a shell variable, looks like:

2 + 3 + 4 +

...which when evaluated, gives an error. Thus, placing a **0** at the end of the string matches the final **“+”** sign, and **expr** is happy

What if there wasn’t a byte count? What if there were no entries − **expr** without parameters doesn’t work − **expr** with **0** does.

So, is this the most efficient code?

Within the shell, yes. Probably the most efficient code would be a call to **awk** and the use of some **awk** scripting, however that is beyond the scope of this chapter and should be examined as a personal exercise.

A final note about the variables

Throughout this exercise, we’ve referred to **$TEMPFILE** and **$LOGFILE**. These variables should be set at the top of the shell script. **LOGFILE** refers to the location of the FTP log. **TEMPFILE** is the actual file used to store the entries of interest. This must be a unique file and should be deleted at the end of the script. It’d be an excellent idea to store this file in the **/tmp** directory (just in case your script dies and you leave the temp file laying around − **/tmp** is regularly cleaned out by the system) − it would be an even better idea to guarantee its uniqueness by including the process ID (**$$**) somewhere within its name:

```
LOGFILE="/var/log/ftp.log"
TEMPFILE="/tmp/scanlog.$$"
```

The final program − a listing

The following is the completed shell script − notice how short the code is (think of what it would be like if we hadn't been pushing for efficiency!).

```
#!/bin/sh

#
#  FILE:        scanlog
#  PURPOSE:     Scan FTP log
#  AUTHOR:      Bruce Jamieson
#  HISTORY:     DEC 1997            Created
#
#  To do :      Truly astounding things.
#               Apart from that, process a FTP log and produce stats

#-------------------------
# globals

LOGFILE="ftp.log"
TEMPFILE="/tmp/scanlog.$$"

# functions


  #--------------------------------------
  # getAccessCount
  # − display number of unique machines that have accessed the page

  getAccessCount()
  {
    echo ‘cut −f1 $TEMPFILE  | uniq | wc −l‘
  }

  #-----------------------------------------------------
  # getUserList
  # − display the list of users who have acessed this page

  getUserList()
  {
    userList=‘cut −f4 $TEMPFILE | sort | uniq‘

    for user in $userList
    do
      echo $user ‘grep $user $TEMPFILE | wc −l‘
    done

  }

  #-----------------------------------------------------
  # getBytes
  # − calculate the amount of bytes transferred

  getBytes()
  {
   numstr=‘cut −f2 $TEMPFILE | sed "s/$/ + /g"‘
   expr $numstr 0
  }


  #-----------------------------------------------------
  # process_action
  # Based on the passed string, calls one of three functions
  #

  process_action()
  {
    # Translate to upper case
    theAction=‘echo $1 | tr [a−z] [A−Z]‘

    # Now, Check what we have
```

```
    case $theAction in
      BYTES) getBytes ;;
      USERS) getUserList ;;
      HOSTS) getAccessCount ;;
      *) echo "Unknown command $theAction" ;;
    esac

  }


#----  Main

#

if [ "$1" = "" ]
then
  echo "No unit specified"
  exit 1
fi

UNIT=$1

# Remove $1 from the parm line
shift

# Find all the entries we're interested in
grep "/pub/$UNIT" $LOGFILE > $TEMPFILE

# Right - for every parameter on the command line, we perform some
for ACTION in $@
do
  process_action "$ACTION"
done

# Remove Temp file
rm $TEMPFILE

# We're finished!
```

Final notes

Throughout this chapter we have examined shell programming concepts including:

- variables
- comments
- condition statements
- repeated action commands
- functions
- recursion
- traps
- efficiency, and
- structure

Be aware that different shells support different syntax − this chapter has dealt with bourne shell programming only. As a final issue, you should at some time examine the Perl programming language as it offers the full functionality of shell programming but with added, compiled−code like features − it is often useful in some of the more complex system administration tasks.

Review Questions

8.1

Write a function that equates the username in the `scanit` program with the user's full name and contact details from the `/etc/passwd` file. Modify `scanit` so its output looks something like:

```
*** Restricted Site Report ***

The following is a list of prohibited sites, users who have
visited them and on how many occasions

Bruce Jamieson x9999 mucus.slime.com 3
Elvira Tonsloy x1111 mucus.slime.com 1
Elvira Tonsloy x1111 xboys.funnet.com.fr 3
Elvira Tonsloy x1111 warez.under.gr 1
```

(Hint: the fifth field of the passwd file usually contains the full name and phone extension (sometimes))

8.2

Modify **`scanit`** so it produces a count of unique user/badsite combinations like the following:

```
*** Restricted Site Report ***

The following is a list of prohibited sites, users who have
visited them and on how many occasions

Bruce Jamieson x9999 mucus.slime.com 3
Elvira Tonsloy x1111 mucus.slime.com 1
Elvira Tonsloy x1111 xboys.funnet.com.fr 3
Elvira Tonsloy x1111 warez.under.gr 1

4 User/Site combinations detected.
```

8.3

Modify `scanit` so it produces a message something like:

```
There were no users found accessing prohibited sites!
if there were no user/badsite combinations.
```

References

Kochan S.G. et al "UNIX Shell Programming" SAMS 1993, USA

Jones, D "Shell Programming" WWW Notes

Newmarch, J "Shell Programming" http://pandonia.canberra.edu.au/OS/13_1.html

Source of `scanit`

```
#!/bin/bash
#
# AUTHOR:   Bruce Jamieson
```

```
# DATE:           Feb 1997
# PROGRAM:  scanit
# PURPOSE:  Program to analyse the output from a network
#           monitor.  "scanit" accepts a list of users to
#           and a list of "restricted" sites to compare
#           with the output from the network monitor.
#
# FILES:    scanit            shell script
#           netwatch    output from network monitor
#           netnasties  restricted site file
#
# NOTES:    This is a totally made up example − the names
#           of persons or sites used in data files are
#           not in anyway related to reality − any
#           similarity is purely coincidental :)
#
# HISTORY:  bleak and troubled :)
#


checkfile()
{
  # Goes through the netwatch file and saves user/site
  # combinations involving sites that are in the "restricted"
  # list

  while read buffer
  do
    username=‘echo $buffer | cut −d" " −f1‘
    site=‘echo $buffer | cut −d" " −f2 | sed s/\\\.//g‘
    for checksite in $badsites
    do
      checksite=‘echo $checksite | sed s/\\\.//g‘
      # echo $checksite $site
      if [ "$site" = "$checksite" ]
      then
        usersite="$username$checksite"
        if eval [ \$$usersite ]
        then
        eval $usersite=\‘expr \$$usersite + 1\‘
        else
          eval $usersite=1
        fi
      fi
    done
  done < netwatch
}

produce_report()
{
  # Goes through all possible combinations of users and
  # restricted sites − if a variable exists with the combination,
  # it is reported
  for user in $*
  do
    for checksite in $badsites
    do
      writesite=‘echo $checksite‘
      checksite=‘echo $checksite | sed s/\\\.//g‘
      usersite="$user$checksite"
      if eval [ \$$usersite ]
      then
        eval echo "$user: $writesite  \$$usersite"
        usercount=‘expr $usercount + 1‘
      fi
    done
  done
}

get_passwd_users()
{
  # Creates a user list based on the /etc/passwd file

  while read buffer
  do
    username=‘echo $buffer | cut −d":" −f1‘
```

```
    the_user_list=‘echo $username $the_user_list‘
  done  < /etc/passwd
}

check_data_files()
{
  if [ −r netwatch −a −r netnasties ]
  then
    return 0
  else
    return 1
  fi
}

# Main Program
# Uncomment the next line for debug mode
#set −x


if check_data_files
then
  echo "Datafiles found"
else
  echo "One of the datafiles missing − exiting"
  exit 1
fi

usercount=0
badsites=‘cat netnasties‘

if [ $1 ]
then
  the_user_list=$*
else
  get_passwd_users
fi
echo
echo "*** Restricted Site Report ***"
echo
echo The following is a list of prohibited sites, users who have
echo visited them and on how many occasions
echo
checkfile
produce_report $the_user_list
echo
if [ $usercount −eq 0 ]
then
  echo "There were no users found accessing prohibited sites!"
else
  echo "$usercount prohibited user/site combinations found."
fi
echo
echo

# END scanit
```