Introduction to Shells

What are Shells?

A shell is the interface between OS and you, the user. The shell interprets the text you type, and the keys you press, in order to direct the operating system to take an appropriate action. A shell can also serve as a programming language.

Bourne Shell

Bourne Shell is the oldest shell. It was written by Stephen Bourne at Bell Laboratories. The Bourne Shell has been the default shell for Ubuntu, and has been a de facto standard in the industry. The Bourne Shell has neither the interactive features, nor the complex programming constructs, of the C and Korn shells.

C Shell

C Shell is a shell developed by Bill Joy at the University of California at Berkeley. The C Shell syntax resembles that of the C programming language, It has powerful interactive features like command history and file name completion.

Korn Shell

Korn Shell is a newer shell developed by David Korn at Bell Laboratories, and is upwardly compatible with most Bourne shell features. It has interactive features like C Shell, but executes faster and has extended in-line command editing capability.

Introduction to Shells

Determining Your Login Shell

You can display the file name of the shell you entered when you logged in by typing:

$ echo $SHELL

The echo command displays the contents or value of a variable named SHELL. The command to list the environmental variables is

env

The Bourne Shell

The Bourne Shell is a “command interpreter”; it takes your commands and interprets them to the system.

For example, if you have to execute a series of commands every day, you may get tired of typing the commands each time. By programming the shell, you can create a shell script, a file containing all of the commands that need to be executed each day. To execute the commands, you only need execute the shell script.

UNIX System Structure

The structure of the system consists of several parts which work together to bring you the operating system. Everything in UNIX is either a file or a process.

The **kernel** is the core of the operating system. It controls the computer's resources and allots time to different users and tasks. The kernel keeps track of the programs being run and is in charge of starting each user on the system. However, the kernel does not interact with the user to interpret the commands. The **shell** is a program that the kernel runs for each user which sets up commands for execution. By having several shells and one kernel, the OS is able to support many users at the same time (the user's requests are not actually processed at the same time, but the kernel schedules processing time in a way which simulates concurrent processing). By having the kernel in control, it is also possible for one user to run several shells. The kernel remains in control of all shells and programs.

When you log on to the system, the kernel checks if your login identifer and password are correct. It then runs a shell program for you to interact with it (you never see this, only the shell after successful login). Most systems will start the POSIX Shell (/bin/posix/sh) as a default, but it is possible to run the Bourne Shell (/bin/sh), the C Shell (/bin/csh), or the Korn Shell (/bin/ksh). instead.

To give you an idea of processes and how the kernel schedules them, let's look at the ps command which lists the processes the kernel is currently coordinating. Type:

ps -ef

The UID column refers to the user identifer (the person who executed this process). PID refers to the process identifer. There are several commands which use the PID, such as kill. For example,

kill -9 4465

will kill (terminate) process 4465.

Shell Commands

A command could include options and parameters to the command. Options to a command can be found with the man command under the description of the command. These options are usually preceded by a dash (-) and are separated from the command name, other options, and parameters by blanks. Parameters, or variables, are data the command needs to function properly. If you omit parameters from the ls command, the current directory is listed. But if you include a directory name (or path name) as a parameter, a listing of that directory is printed. Command syntax usually takes the following form:

command [options] [parameters]

Sequential Processing

When you enter commands line by line (pressing [Return] after each command), you are telling the system to complete the command (or program) before executing the next command. Executing:

date

ps -ef

who

will complete each command before going on to the next. You can place all of the commands on the same line by using the “;” separator. For example,

date; ps -ef; who

is equivalent to entering each command on a separate line. This process is called sequential processing. New programs or commands cannot be started until the preceding program or command has completed.

If parameters are required by the program, they are entered as usual. The semicolon is placed after the last parameter. While a program is running as a sequential process, there is no response to keyboard activity until after the program has completed (other than the keyboard buffer delay).

Programs already in progress when a program with sequential processing is executed continue to run as usual. While a program is running as a sequential process, you have the option of waiting for the program to finish.

Nonsequential (Background) Processing

Programs can also be run nonsequentially , in which case, each program runs without waiting for the previous program to complete. This type of execution is more commonly called “running in the background” Follow the program name with & to specify background processing.

program1 & program2 & program3 &

This example runs program1, program2, and program3, in the background, and returns a prompt to the user immediately. Note that programs that write to the terminal or require input are poor choices for background execution, since the output will be intermixed on the screen, or the input may not be read by the correct program.

Redirecting Input and Output

Every program has at least three data paths associated with it: standard input, standard output, and standard error output. Programs use these data paths to interact with you. By default, standard input (stdin) is your keyboard. The default destination for both standard output (stdout) and standard error (stderr) is your screen.

Redirecting input and output is a convenient way of selecting what files or devices a program uses. The output of a program that is normally displayed on the screen can be sent to a printer or to a file. Redirection does not affect the functioning of the program because the destination of output from the program is changed at the *system* level. The program is unaware of the change.

I/O redirection enables you to change a specific data path of a program while leaving its other data paths unchanged. For example, stdout can be stored in a file instead of written to your screen.

How to Redirect Input and Output

I/O redirection symbols are entered on the shell command line, or from a shell program. The program begins executing with the data paths specified by the

redirection symbols. To specify I/O redirection for a program, each file name is

preceded by a redirection symbol, as in:

programA < file name

programB > file name

Spaces between the redirection symbols and the file names are optional. The symbol identifies the name that follows it as a file for input or output. The redirection symbols are listed in Table 1.

Table 1. Redirection Symbols

Symbol	Function	Example
<	Read *standard input* from an existing file.	`program1 <input.data`
>	Write *standard output* to a file.	`program2 >output.data`
>>	Append *standard output* to an existing file.	`sample.prog >>output.data`

Note Using > destroys any previous contents of the file specified to receive the output. If a file's contents must be preserved, use >>.

Note Be careful not to use the same file for standard input and standard output. When input and output operations access the same file, the results are unpredictable.

If a file you specify with a redirection symbol is not in the current directory, you should use a path name to identify it. The following actions are taken when the system does not locate files named with the redirection symbols:

- If a file specified for input with the < symbol is not located, an error message is displayed.
- If a file specified for output with the > or >> symbol is not located, it is created and used for program output.

Examples

The following examples show how the data paths of programs, commands, or utilities can be modified with the redirection symbols.

CHItest < data1

Runs the program CHItest using the file data1 as input.

date >> syslog

Adds the current time and date to the end of the file syslog.

Pipes

Two or more programs or commands can be connected so the output of one program is used as the input of another program. The data path that joins the programs is called a pipe. Pipes allow you to redirect program input and output without the use of temporary files.

When programs are connected with pipes, the shell coordinates the input and output between the programs. The pipes only transfer data in one direction, from the standard output of one program to the standard input of another program.

How to Connect Programs With Pipes

The vertical bar (|) is the “pipe” symbol. Parameters for the program are listed after the program name, but before the | symbol. Spacing between the program names and vertical bars is optional. The syntax used for connecting programs with pipes is as follows:

program a | program b | program c

Here are some examples.

To print the number of files in the current directory, type:

ls | wc

To print a listing of each file in the directory, and paginate it for convenient

screen viewing, type:

ls | more

To send the contents of file to pr, which formats the data and then passes it to lp for printing on the line printer, type:

cat file | pr | lp

Redirection in Pipes

The redirection symbols can be used for programs connected with pipes. However, only the data paths not connected with pipes can be changed. If you specify a change to a data path being used with a pipe, then an error occurs.

The following changes are permitted:

- The standard input of the first program using a pipe can be redirected with the < symbol.
- The standard output of the last program using a pipe can be redirected by using the > symbol or appended to an existing file with the >> symbol.

Examples

The following commands show how programs can be connected with pipes and how additional changes can be made to data paths with redirection symbols.

The first example takes the standard output from test_prog1 and uses it as standard input to /usr/output_prog.

test_prog1 | /usr/output_prog

The next example runs four programs connected with pipes and puts the output of the fourth program in store_file.

get_it | check_it | process_it | format_it > store_file

Pipe Example

The following pipe uses several of the symbols we just discussed. Try to figure out what will happen before you read the description below.

sort pdir; ((pr pdir | lpr)& (sort local)&); cat local >>pdir

This pipeline will run three sets of commands sequentially. The first command is to sort the pdir file. When it is completed, the second command set is executed. The parentheses separate the commands so the shell knows which command to associate with a symbol. Therefore, the two commands (pr and sort) are run nonsequentially. So, at the same time, the pdir file is formatted and sent to the printer, and the local file is sorted. Finally, the cat command is run which appends the local file to the pdir file.

File Name Generation

A helpful way to reduce typing is to use patterns to match file names. If you are in a directory with a file "programming" you can see a listing with either:

ls programming

or you can use a pattern to match:

ls p*

where "*" will match any character or string of characters. If you have another file beginning with "p", it too will be listed. Table 2 shows the file generation symbols you can use:

Table 2. File Generation Symbols

Symbol	Description
*	Matches any string of characters including the null string.
?	Matches any single character.
[...]	Matches any one of the characters enclosed in the brackets. A pair of characters separated by a minus will match any character between the pair (lexically).

[a-z]?cubit*.[ca]

will match a file which begins with any character a through z (lower case), followed by any single character, followed by the string "cubit", followed by any number of characters, and which ends in ".c" or ".a".

Shell Scripts

Introduction to Shell Scripts

Simple Scripts

Stringing commands together on a line with sequential processing, background processing or pipes is an extremely useful tool for a limited number of commands. To save typing the commands repetitively, in the case where you use the same sequence of commands often, you can place the command line(s) into a file. This file is called a shell script. You create a file containing the commands, tell the system you want the file to be executable (so it can be run as a program), and then type the name of the file to execute the commands in the shell script.

A simple shell script could contain the following command line:

date; who; ps -ef; du /users

which executes each command only when the previous command has completed. To create the script, enter an editor (vi for example) and type the above command line. Save the file.

To run the script, you have two methods: the sh command, or changing the

permissions on the file. The sh command will create a new shell to run the script. As mentioned in the beginning of this tutorial, it is possible to have several shells running at the same time (with the kernel in control). The sh command creates a new shell to execute the file you specify (if you don't specify a file, it creates a new shell similar to the one you are already in). To execute the script with the sh command, type:

sh scriptname

Where scriptname is the name of the file you placed the command line in.

The common way to run a script or program, however, is to declare the file executable with the chmod command. chmod is used to alter the permissions on a file. For our purposes, we will declare the file to be executable by everyone on the system, but only you can update the file. Type:

chmod +x scriptname

Now the file is executable, and you only need enter the file name to run the script (simply type the scriptname as if it was a command). Your script will execute, and you will see a large output. Both methods of executing scriptname have the same net effect, they just behave differently at first.

Scripts With More Than One Line

The example above just uses one command line for the script. You can, however, make the script easier to read and contain more than one line of commands. Each line of commands is executed in sequential order (the previous line must complete before the next line is executed). So, we can take the previous example:

date; who; ps -ef; du /users

and spread the command line into four lines which accomplish the same thing:

```
date
who
ps -ef
du /usr
```

When this script is executed, you get the same results as before.

Echo and Redirection in Scripts

If you have a large output from a script like in the above example, you may wish to place some headers or comments in the output and place the output into a file. The echo command will print titles or comments for you. It works in the following manner:

echo "string"

where string is a string of characters.

Modify your example script to look like:

```
echo "Current date and time: \c"
date
echo "Users logged in:\n"
who
echo "\nCurrent processes:"
ps -ef
echo "\nUser disk usage:"
du /usr
```

where "\c" causes the next line of output to be printed on the same line, and "\n" causes an extra carriage return and line feed.

Next you can execute the file using the redirection symbols to append the output to another file. For example, let's say our file is called status1, and the file we wish to place the output in is called status_file:

status1 >> status_file

Each time you monitor the system, you can have the output added to a file.

Basic Shell Programming

All of the constructs of shell programming can be executed in two ways: you can type the commands into a file so they will all be executed when the file name is entered (after changing the permission), or you can enter the commands directly into the shell (just as you enter commands like "date").

When you enter shell constructs directly into the shell, you can either type them on the same line (and press return to execute them), or you can type them over several lines. For example, we can type the following construct two ways.

First on one line:

if test -d /d1; then echo "/d1 is a directory"; fi

Then on several lines:

```
if test -d /d1
then
   echo "/d1 is a directory"
fi
```

Typing the command on one line is simple to do in the shell. If you type the command on several lines, you will receive a secondary prompt. The secondary prompt is usually a \>".

Parameters

In addition to shell parameters, you can create parameters of your own. The format for user-created parameters is:

parameter = value

Note that there must be no blanks between the parameter, equal sign (=), and the value. You can create these parameters while you are in the shell, and they will help you save typing. Look at an example:

x=phantom

When you type in the above statement, the variable x is created and the value "phantom" is assigned. To access the variable x, you will need to precede the variable name with a dollar sign ($). Try this:

echo $x

Using Parameters in Shell Programs

You can use parameters within your shell programs in the same way. On one line, define the variable with the same format. When you wish to refer to the value of the parameter, you precede the parameter name with a dollar sign ($).

One advantage of using parameters in a program is that you can combine them with each other, or with file names, to create a variety of file paths with less typing. Let's say you define a parameter to be the path to a directory:

dir2=/users/dave/projects/memos

If you want to print the contents of a file in the above directory, you would use the cat command as follows:

cat ${dir2}/junememo

where the braces differentiate between the parameter and the characters following it and junememo is the name of a file (note we had to include a slash before the filename or "junememo" would have been concatenated directly to "memos" and we would have received an error message). What has happened is called parameter substitution.

Parameter Substitution

When you wish to include the value of a parameter into a string or statement, you must precede the parameter with a dollar sign ($).

Positional Parameters

When you execute a shell program, you can include parameters on the command line. When you do, each parameter must be separated with a blank, like:

scopy file1 file2 file3

where scopy is a shell program with three parameters.

When the shell program runs, you can access the value of these parameters (each separated by a blank) with positional parameters named $0, $1, $2 . . . $9. If your list of values exceeds nine parameters, the values are placed in a buffer, and you can access the values with the shift command.

If you need to know the number of positional parameters (let's say you wish to see if the user included any parameters at all) you use $#. If you need a parameter which contains all of the positional parameters separated by blanks, use $* (this is useful if the positional parameters constitute a sentence or even a command line).

Positional parameters are accessed within the body of the script. When the script is executed, the parameters are assigned values only for the execution of the script.

The Backslash

The backslash (\) will cancel, or escape, the special meaning of the next character:

echo \$dir1

will echo "$dir1" instead of the parameter value of "dir1" because the dollar sign is told to have no special meaning.

Conditions: The if Statement

Your shell programs may need to execute a command or set of commands only if a certain condition exists. Let's say you want to \execute the sort command only if the file exists, otherwise print an error message". Your statement would look like:

```
if test -f $1
then
   sort $1
else
   echo "file does not exist"
fi
```

where $1 is a filename passed in from the command line. For the case of \if this then that, else if this then that, etc" we can use the elif statement which means "else if". The format for the if construct looks like:

```
if command list1
then command list2
elif command list3
then command list4
.
.
else command listn
fi
```

It is helpful to indent to indicate parts of the if construct. Make sure you end the construct with fi.

Let's look at an example to better clarify this construct:

```
if grep jones personnel
then
   echo "jones" >> available
elif grep castle personnel
   then
      echo "castle" >> available
   else
      echo "empty" >> available
fi
```

Test

An often used command is the test command. You can use the test command in the if construct to test conditions such as equality. Here are two examples to explain the use of the test command:

```
dir1=/usr/bin
if test $dir1 = /usr/bin
then
echo "directory found"
fi
```

This construct "tests" if the value for dir1 (notice how we used parameter substitution) is equal to the string "/usr/bin".

```
if test $# -eq 0
then
   echo "no positional parameters"
fi
```

The -eq option is used to test the numeric equivalence of the $# and the value zero. Remember $# is the number of positional parameters passed to the script.

To make typing easier, you can use an abbreviation for test. The square brackets enclosing the options and parameters do the same as the test command. For example:

```
if [ $# -eq 0 ]
```

has the same meaning as the first line in the above example (if test $# -eq 0). Be sure to separate the square brackets from any characters with a blank.

Read

If you wish to receive input during the execution of a shell program, you can use the read statement with the following format:

```
read [ parameter... ]
```

where [parameter...] means a list of one or more parameters. When the computer executes this statement, it gets input from the keyboard (unless you use redirection symbols to get input from a file). Each word (words are separated by blanks) typed in is assigned to the respective parameter in the list, with the leftover words assigned to the last parameter.

Exit

Each command returns a status when it terminates. If it is unsuccessful, it returns a code which tells the shell to print an error message. You can use the exit command to leave a shell program with a certain exit status. The usual exit statuses are:

Value	Description
0	Success.
1	A built-in command failure.
2	A syntax error has occurred.
3	Signal received that is not trapped

Comments

To add to a shell program comments, simply start the line with a pound sign (#). For example:

```
# this line is a comment
```

Looping

Many times sequential processing in a program is just not enough. We need a mechanism which will allow us to repeat the same set of commands using a different set of parameter values. To accomplish this in shell programming you can choose between three looping constructs: for, while, and until.

For

The for construct allows you to execute a set of commands once for every new value assigned to a parameter. Look at the following format:

```
for parameter [ in wordlist]
do command-list
done
```

where parameter is any parameter name, wordlist is a set of one or more values to be assigned to parameter, and command-list is a set of commands to be executed each time the loop is performed. If the wordlist is omitted (and also "in"), then the parameter is assigned the value of each positional parameter.

The word list is a versatile quantity in the for construct. It can be a list which you specifically type (separated with blanks), or it can be a shell command (using grave accents) which generates a list. Let's look at some examples.

```
for i in `ls`
do
  cp $i /users/rhonda/$i
  echo "$i copied"
done
```

This example will assign one file at a time from the current directory (the values are generated by the `ls` command) to the "i" parameter. The loop's command list will copy the file to another directory, then report the success of the copy.

```
for direc in /dev /usr /users/bin /lib
do
  num=`ls $direc | wc -w`
  echo "$num files in $direc"
done
```

This example lists the values to be given to direc in the loop. The command list then lists each respective directory (the parameters) and assigns a word count (wc) to the num parameter. Then the word count is printed out.

```
for i
do
  sort -d -o ${i}.srt $i
done
```

This final example will assign each positional parameter respectively to "i" (since the in clause was omitted). If the positional parameters are file names, the script will sort the file and place the result in a file having the same name as the unsorted file with ".srt" appended to it. It will then get the next positional parameter until all have been accessed.

While

The while construct repeatedly executes a list of commands in the following

format:

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

All of the commands in command-list1 are executed. If the last command in the list is successful (indicated by an exit status of 0 from the command), then the commands in command-list2 are executed. Then we loop back to execute command-list1 until the last command in the list is unsuccessful, and then the while loop terminates.

```
while [ -r "$1" ]
do
   cat $1 >> composite
   shift
done
```

This example tests the positional parameter to see if it exists and is a readable file. If it is, it appends the contents of the file to the composite file, shifts the positional parameters (what was $2 is now $1) and tests the new file. When the file is not readable, or there are no more positional parameter values ($1 is null) the while loop is terminated.

Until

The until construct is basically the same as the while construct except that the commands in the loop are executed until the conditions are true (instead of false like in the while loop). Here is the format:

```
until command-list1
   do command-list2
done
```

If the last command in command-list1 is unsuccessful , then the commands in command-list2 are executed. When the last command in command-list1 is successful, the until loop is terminated. Let's use the same operation in the while section to illustrate:

```
until [ ! -r $1 ]
do
   cat $1 >> composite
done
```

Notice the subtle difference with the while loop. The ! negates the test conditions. We execute the loop until the condition is true (or successful). The while loop executes the commands while a condition is true (or successful).

Case

The case construct is an expansion of the if construct. If you have a condition which may have several possible responses, you can either string together many if's or you can use the case construct:

```
case parameter in
   pattern1 [ | pattern2...] ) command-list1 ;;
   pattern2 [ | pattern3...] ) command-list2 ;;
   .
   .
   .
esac
```

After the first line (which asks if parameter matches one of the following conditions) is listed all of the possibilities for parameter. Each of these lines contains a pattern (or value for parameter). The brackets ([j| pattern2...]) refer to other values that may be valid. The vertical bar (|) represents \or". Finally, the pattern(s) are followed by a close parenthesis), and then by a list of commands to be executed if the patterns match.

An example may better illustrate:

```
case $i in
   -d | -r ) rmdir $dir1
             echo "option -d or -r" ;;
   -o )      echo "option -o" ;;
   -* )       echo "incorrect response";;
esac
```

The eval Command

The eval command reads its arguments as input to the shell, and the resulting commands are executed. The format is:

eval [arg ...]

where arg ... is one or more arguments which are shell commands or shell programs. Here is an example:

eval "grep jones $p_file | set | echo $1 $2 $4"

eval will execute the pipe contained in double quotes in the shell.

Input/Output

The common redirection symbols can be used in shell programs.

<& digit

This input redirection symbol uses the file descriptor associated with the descriptor digit. Most programs have standard input as 0, standard output as 1, and standard error as 2 (stdin, stdout, and stderr respectively).

>&digit

is the format for using descriptors, where digit can be any single digit (0, . . . ,9). The most commonly used redirection of this form is 1>&2 or 2>&1. For example,

echo File $name not found 1>&2

The output of this line is redirected to the standard error (your terminal).

Special Commands

Expr

The expr command is very useful for performing arithmetic operations in shell programs. It also has other operations useful for string manipulation.

With the form:

expr expression {+ - } expression

you can add or subtract integers.

```
a=15
expr $a + 5
```

will return the string 20.

To modify variables, you can use a similar format to:

a=`expr $a + 1`

using command substitution (grave accents) to place the new value in the variable a.

The symbols for multiplication, division, and remainder of integer-valued arguments are: *, /, and %, respectively. Note the * is preceded by a backslash (\) to escape the shell's interpretation of the asterisk.

To compare integers, use the following format:

expr expression { =,\>,\>=,\<,\<=,!= } expression

where != is \not equal to", and the other symbols represent mathematical comparisons (again, note the backslash before the special characters < and >). The function will return 0 if the comparison is successful, and 1 if it is not. Here is an example of how a comparison might be used:

```
if expr $a \<= $b
then
   echo "$a is less than or equal to $b"
fi
```

Defining Functions

The more complicated your shell programs get, the more you will want to modularize them by using functions. This way you can create generic functions which can be re-used and eliminate repetitive code.

To define a function, use the following syntax:

name() {list;}

where name is the name of the function, and list is a list of commands used in the function.

Here is an example to show how functions are defined:

```
stat() {
   if [ -d $1 ]
   then
      echo "$1 is a directory"
      return 0
   else
      echo "$1 is not a directory"
      return 1
   fi;
}
```

This function tests the filename to see if it is a directory. If it is it returns a status of 0. Otherwise it returns status 1. Do not forget to place the semi-colon (;) at the end of the last line.