# Bash Shell Scripting

C

## SOME OF THE MAIN TOPICS IN THIS CHAPTER ARE

# Introduction

Part of being able to upgrade and configure PCs using Linux is knowing how to write shell scripts. The preferred shell for scripting is the Bash (or Bourne Again) shell, a GNU answer to the classic Bourne shell. Although it is possible to write scripts in Tcsh, I strongly recommend against it. The main advantages of using Bash scripts, as opposed to full-blown compiled programs written in C, are that Bash scripts can be written in a shorter time and that a script requires no compile time. In order to run a script, you need only write your script in a text file and then allow execution permissions on that file with the `chmod` command. Bash scripts also have a lot of built-in file handling capabilities that allow you to do more work with less code. Compare the following examples:

A Hello World Bash script:

```
#!/bin/bash
echo Hello World!
echo What is your name?
read thename
echo Hello $thename, glad to meet you!
```

A Hello World C program that does almost the exact same thing:

```
#include <stdio.h>
main()    {
  char thename[255];
  printf("Hello World!\n");
  printf("What is your name?\n");
  scanf("%s",&thename);
  printf("Hello %s, glad to meet you!\n", thename);
  }
```

In this case I saved a few lines of code. However consider the following example:

```
#!/bin/bash
#
#This program finds directories containing the strings "gnome" and "kde,"
#and puts them in alphabetical order in the file kde-gnome-file.
#
locate gnome > .tmp.gnomefile
locate kde > .tmp.kdefile
sort .tmp.kdefile .tmp.gnomefile > kde-gnome-file
rm .tmp.gnomefile .tmp.kdefile
```

This example took a second-and-a-half to run on my system. Admittedly, this is slow compared to a binary program, written in C and compiled with gcc. On the other hand, that same C program would be considerably longer than nine lines, even without comments, and would definitely take longer than a second-and-a-half to compile. Shell scripting saves time when you have to perform relatively lightweight tasks on not much data. For massive projects, you are better off programming in C (or some other compiled language).

# Why Is Bash Shell Scripting Important for Hardware?

Bash shell scripting is important from a hardware maintenance perspective for two reasons. First, every user account has three configuration files that handle basic setup and shutdown tasking. These are the following:

| | |
|---|---|
| `.bash_profile` | Executed by Bash on initializing login shells |
| `.bashrc` | Executed by Bash on initializing non-login shells |
| `.bash_logout` | Executed by Bash on exiting login shells |

The distinction between `.bash_profile` and `.bashrc` depends on the distinction between a login shell and a non-login shell. A login shell is the first shell that one sees when one logs in to an account that uses Bash as a default shell. A non-login shell is any other instance of a Bash shell. These scripts tell the shell where to find important resources such as printer drivers, mouse drivers, and disks, as well as setting up aliases for different commands. For you, the person performing upgrades and repairs on the PC, knowing how to modify these scripts correctly makes it easier to deliver a more complete, better working system to the client.

The second reason why Bash shell scripting is important is that it provides a handy tool for detecting problems with hardware. For example, if you are responsible for maintaining a RAID array, you can set up a quick shell script to read through the disk message logs every 300 seconds to detect problems, and page you if there is a problem. This cuts down on the amount of work you need to do to detect problems in the first place.

# How to Script, from Writing to Running

Suppose that you want to write a simple script to read through a message log and detect a possible hacker. What you should do is search through the file /etc/group and see whether any intruders have added themselves to the root group—that is, the group that has root permissions—without your authorization.

On most Linux home systems, the file /etc/group looks like this:

```
root:x:0:
daemon:x:1:
bin:x:2:
...
```

In this case, you are concerned with the first line. Anyone in group root has superuser permissions and can act as the system administrator. Assuming that administrators you've created already know the password to root, you can assume that if another person's user ID is attached to the root group, this person gained administrator permissions without your consent. You can write a simple script to detect whether such a change has taken place.

Thus, use a text editor (such as Emacs or vi) to write your script, named `hackfind0`. The script is as follows:

```
#!/bin/bash
okline="root:x:0:"
read line1 < /etc/group
if [ $line1 != $okline ]; then
         echo Hacker detected! ${line1#$okline} ¦ mail root
         echo Hacker detected, check your mail!
         fi;
```

### Note

Note that indentation doesn't matter for individual command lines; you could just as easily have written the script as

```
#!/bin/bash
okline="root:x:0:"
read line1 < /etc/group
if [ $line1 != $okline ]; then
echo Hacker Detected! ${line1#$okline} ¦ mail root
echo Hacker detected, check your mail!
fi;
```

The code has the same function but is harder for a human to read. You should, therefore, indent your code when you're writing functions or flow control structures (such as `if` statements and `for` loops) in order to make your code easier to follow and debug.

When this script is written, you need to set execute permissions. This is done using the `chmod` command. In particular, if you want to have anyone able to execute it, use

```
chmod 711 hackfind0
```

Because you only want the root user to execute it, type

```
chmod 710 hackfind0
```

Now the only people who can read and write the script are those who have the root password, and only persons in the root group can execute it.

Then, whenever you want to execute the script, type the following if `hackfind0` is in your path:

```
hackfind0
```

Otherwise, type the following if it isn't:

```
./hackfind0
```

# The Parts of a Shell Script

Now look at the shell script `hackfind0` to see what it consists of:

```
#!/bin/bash
okline="root:x:0:"
read line1 < /etc/group
if [ $line1 != $okline ]; then
     echo Hacker detected! ${line1#$okline} ¦ mail root
    echo Hacker detected, check your mail!
    fi;
```

I'll analyze this script line by line. The first line is `#!/bin/bash`. This line tells the shell that this is a Bash shell script. If this line is not present, the script can be executed by invoking another instance of Bash with `bash <script name>`. The second line defines a variable `okline`, and then sets its value as a string. In C or Java, you would have to specify the type of this variable. Bash assigns types automatically, although you can set them using the `declare` and `typeset` keywords.

The third line contains two features of shell scripting: a command statement, `read`, and a redirector. The redirector tells `read` to use the file /etc/group as standard input. Normally, `read` would prompt the user for a line of input. Instead, `read` takes the first line of /etc/group and places that line in the variable `line1`.

The fourth line is a condition statement, `if`, which consists of a conditional (inside the brackets) and a set of commands to execute if the conditional is true (between the keywords `then` and `fi`). The fifth and sixth lines contain the keyword `echo`, which outputs a line of text to standard output. The sixth line prints to the screen, but the fifth line directs a line of standard output to another command: `mail`. This line is put into an email message that is sent to the root account. The fifth line also contains a pattern matcher: `${line1#$okline}`. This takes the string in `line1` and compares it against the string in `okline`; if there is text after the string `root:x:0:`, the `echo` command will write that additional text to the email. This example shows some, but not all, of the interesting functionality of Bash scripts.

# List Commands

In most cases, you type one command at a time at the prompt and execute one command at a time in a shell script. There are, however, cases where you want to run one command conditionally depending on how another command runs. For simple cases, this can be done using the operators `;`, `&`, `&&`, and `||`.

| Keyword | Use | Meaning |
|---|---|---|
| `;` | `<cmd1> ; <cmd2>` | Execute `<cmd1>` and `<cmd2>` |
| `&` | `<cmd> &` | Execute `<cmd>` as a background process |
| `&&` | `<cmd1> && <cmd2>` | Execute `<cmd2>` only if `<cmd1>` returns an exit status of 0 (that is, it ran with no problems) |
| `\|\|` | `<cmd1> \|\| <cmd2>` | Execute `<cmd2>` only if `<cmd1>` returns an exit status other than 0 (usually this means something went wrong) |

For example, I commonly run Netscape on my own machine by typing

```
netscape &
```

I also set up my `.bash_profile` file to automatically start X Window, or to tell me that there was a problem with X Window, with this statement:

```
startx || echo Something went wrong starting X.
```

# Declaring and Assigning Variables

Declaring a variable can be done two ways: through `declare` or `typeset`, or by assigning a value. There are also a number of variables built directly in to the Bash shell. These are detailed in Appendix D, "A Quick Guide to Bash Shell Redirection."

## *declare* and *typeset*

The keywords `declare` and `typeset` are functionally equivalent. These set a particular type for a variable. For example, the following command sets `x1` equal to 15 and sets it to the type `integer`:

```
declare -i x1=15
```

Assigning anything but an integer value to `x1` produces a syntax error.

The options for `declare` are as follows:

| Option | Meaning |
|---|---|
| `-a` | Array |
| `-f` | Function names |
| `-F` | Display function names without definitions (to `stdout`) |
| `-i` | Integer |
| `-r` | Read-only |
| `-x` | Export |

Note that without the `-i` flag, the variable can be either a string or an integer.

### *Note*

Using `declare` on its own displays the values of all variables in the environment.

## Assignment

The easiest way to declare a variable is simply to provide a variable name and assign a value to it. (Early BASIC programmers might recognize this convention.) For example, in the earlier script, you declared a variable `okline` by assignment:

```
okline="root:x:0:"
```

Note that when you assign a value, there should be no spaces separating the = sign from either parameter. For example, the following command returns a syntax error:

```
okline = "root:x:0:"
```

## Read-Only Variables

Another way to make a variable read-only is to use the `readonly` keyword.

For example, in your script you could have replaced line 2 with

```
readonly okline="root:x:0:"
```

If later in the script, you had the following declaration, the script would return a syntax error:

```
okline="fubar"
```

Note that `readonly` is equivalent to `declare -r`.

## Accessing Variables

To use a variable in a function, you need to precede it with a dollar sign ($). Otherwise the script can give you a syntax error, simply become confused, or perform an assignment when it needs to perform a comparison. For example,

```
$line1=$okline
```

is much different from

```
line1=$okline
```

In the first case, the two are compared. In the second case, `line1` is assigned the value of `okline`.

# Lists and Arrays

Two other important types of data, besides variables, are lists and arrays. The difference between lists and arrays is subtle but important. A list is a string of values separated by a colon (:). Values are accessed using `for` loops and `select` commands, as well as through other devices (`pushd` and `popd`). Lists are usually declared through assignment. The most common example of a list is the `PATH` variable, which contains a list of directories to search through to access a command. For example,

```
PATH=~/bin:/bin:/sbin:/usr/bin:/usr/local/bin:/usr/local/sbin:/usr/sbin
```

An array is a variable with a series of spaces to hold a series of values. Unlike a list, where only the first and last values are easily accessible without moving through the entire list, every value of an array is instantly accessible. The most straightforward way to declare an array is by declaring each value in the array individually:

```
array1[0]=Monica
array1[2]=Suzanne
array1[1]=Mike
array1[3]=Matt
array1[12]=Gus
array1[13]=Kira
     ...
```

Another way to declare this same array is as follows:

```
array=(Monica Mike Suzanne Matt [12]=Gus Kira ... )
```

Note that in both cases, the same names are assigned to the same spaces—that is, `Monica` to space `0`, `Mike` to space `1`, up to `Kira` to space `13`. If you want to declare an empty array, you can use the `declare` command with the `-a` option—that is, declare `-a array1`.

To access individual elements of an array, use `${array[index]}`. For example,

```
array=(Monica Mike Suzanne Matt [12]=Gus Kira)
echo ${array[1]}
```

returns

```
Mike
```

The same would happen if you substitute `${array[@]}` (no quotes) or `${array[*]}` for `"${array[@]}"`.

If you use `"${arrray[*]}"` instead (note the quotes), Bash interprets this as a string containing all the elements of the list. Thus

```
for name in "${array[*]}"; do
     echo $name
done
```

returns

```
Monica Mike Suzanne Matt Gus Kira     (all on one line).
```

# Declaring Functions

Bash shell scripts can access Linux commands and Bash shell commands. It is also possible for a shell script to contain its own local commands, known as functions.

There are two ways to declare functions:

```
function <function name>
     {
     <commands>
     }
```

and

```
function <function name> ()
     {
     <commands>
     }
```

There is no difference between the two; the parentheses in the second declaration are there to make C programmers feel more at home. When a function is declared, it can be used as a command later in the script, until the `unset` command is used (more on that later). For example,

```
#!/bin/bash
function foo
     {
     echo FOO!
     }
function bar
     {
     echo BAR!
     }
foo
bar
```

This script, when called, would print out:

```
FOO!
BAR!
```

Admittedly this is rather silly, but if function `foo` represented a lot of commands and these commands had to be performed over and over again, `foo` would be more useful. For example,

```
#!/bin/bash
function Gettysburg
     {
     echo Four score and seven years ago our forefathers
      ...
     echo shall not perish from the Earth.
     }
if [ $string1="Abraham" ]; then
     Gettysburg
     fi
if [ $string2="Lincoln" ]; then
     Gettysburg
     fi
```

In this case, the use of function `Gettysburg` saves you from having to type many lines of shell script. It is possible to declare variables within a function. Variables declared within a function are accessible from anywhere in the script after the function call, unless you declare them using the local keyword. For example,

```
function foo
   {
   bar=14
   }
foo
echo $bar
```

will have output

```
14
```

On the other hand, if you replace line 4 with `local bar=14`, `bar` has no value outside of function `foo`, and line 7 will output a blank line.

# Conditionals

The hardest part of getting used to Bash shell scripting is getting used to Bash conditionals. This is part of the reason why C programmers tend to prefer the C or Tcsh shells: Despite the fact that they are poor scripting shells, the conditionals are essentially the same as in C. Conditionals are statements which return a value of `true` or `false`.

There are two formats for a conditional:

```
test <statement>
```

and

```
[ <statement> ];
```

where a statement consists of a conditional operation. A conditional operation checks one or more properties of a file, an integer, or a string.

## File Conditional Operations

A file conditional operation is one that checks the status of a file or compares two or more files. For example, the following operation returns true if <file1> exists

```
[ -e <file1> ];
```

On the other hand, the following operation returns true if <file1> is newer than <file2>:

```
[ <file1> -nt <file2> ];
```

## Integer Conditional Operators

An integer conditional operator compares two integer expressions for equality or inequality. These expressions can be integers, variables storing integer functions, or arithmetic expressions. For example, the following command returns true because 3=3:

```
[ 3 -eq 3 ];
```

Additionally, this command returns true because the expression $((1+2)) evaluates to 3:

```
[ $((1+2)) -eq 3 ];
```

(Note that replacing $((1+2)) with 1+2 produces a syntax error.)

Similarly, the following expression returns true if $var1 = 3:

```
[ 3 -eq $var1 ];
```

Additionally, the following expression returns true if $var1 = 1:

```
[ $(($var1+2)) -eq 3 ];
```

## String Conditional Operators

It is also possible to compare two strings for dictionary order and for equality. For example, the following expression returns true if $string1 and $string2 are exactly identical:

```
[ $string1 = $string2 ];
```

There are three other string comparisons, which are listed in the following script:

```
[ $string1 != $string2 ]; (true if $string1 and $string2 are different)
[ $string1 < $string2 ]; (true if $string1 comes before $string2 lexico-
                graphically; that is, if $string1 would come
                before $string2 in the dictionary.) (Note: not
                available with all versions of bash.)
[ $string1 > $string2 ]; (true if $string1 comes after $string2 lexico-
                graphically; that is, if $string1 would come
                after $string2 in the dictionary.) (Note: not
                available with all versions of bash.)
```

Note that these comparisons are case sensitive. That is, Mike is not equal to mike.

## Compound Comparisons

It is possible to string comparisons together using the AND and OR operators. You do this using the -a operator (*a* stands for *and*) and the -o operator (*o* stands for *or*). You can also group operations with escaped parentheses (\( and \)). For example, the following expression returns true:

```
[ 3 -eq 3 -a 4 -eq 4 ];
```

As does the following expression:

```
[ 3 -eq 4 -o 4 -eq 4 ];
```

Note that the and operation (-a) has precedence over the or operation (-o); in long comparison operations, however, it is easy to get confused by the sheer volume of script code. So, to reduce ambiguity, it might be better to use parentheses to group operations. For example, the following expression evaluates the statement <stmt2> -a <stmt3> first:

```
[ <stmt1> -o \( <stmt2> -a <stmt3> \) ];
```

Whereas the next expression evaluates <stmt1> -o <stmt2> first:

```
[ \( <stmt1> -o <stmt2> \) -a <stmt3> ];
```

Without parentheses, the following statement evaluates the same as the first example:

```
[ <stmt1> -o <stmt2> -a <stmt3> ];
```

# Pattern Matching

Sometimes you need to find a pattern inside a string; the Bash shell provides tools to do this easily. The format for a pattern is a string containing literal characters and three wildcard characters:

| | |
|---|---|
| * | Any string of characters, including the empty string |
| ? | Any one character |
| [...] | Any one of a set of characters |

For example: *.jpg matches filenames mike.jpg, tux.jpg, and lgs.jpg; mike?.jpg matches mike1.jpg, mike2.jpg, and mikes.jpg, but not mike.jpg; and [A-Z][A-Z][A-Z][A-Z] matches any four capital letters in a row.

You can use patterns to perform powerful deletion and replacement operations on a string.

| Operation | Use |
|---|---|
| ${<var>#<pattern>} | If the pattern matches the beginning of the variable's value, delete the shortest part that matches. So if $str = "fnordfnord-lala", then ${str#"fnord"} returns fnordlala. |
| ${<var>##<pattern>} | If the pattern matches the beginning of the variable's value, delete the longest part that matches. Thus, ${str#"fnord"} returns lala. |
| ${<var>%<pattern>} | If the pattern matches the end of the variable's value, delete the shortest part that matches. Thus ${str%"la"} returns fnordfnordla. |
| ${<var>%%<pattern>} | If the pattern matches the end of the variable's value, delete the longest part that matches. Thus ${str%%"la"} returns fnordfnord. |
| ${<var>/<pattern>/<string>} | The first, longest match to <pattern> in <var> is replaced by <string>. If <string> is a null string, the match is deleted. Thus, ${str/"fnord"/"boo"} returns boofnordlala. |
| ${<var>//<pattern>/<string>} | All longest matches to <pattern> in <var> are replaced by <string>. Thus, ${str/"fnord"/"boo"} returns booboolala. |

# Flow Control

Flow control comes in five flavors: if statements, case statements, while and until statements, for statements, and select statements.

## *if* Statements

if statements take the following form:

```
if <boolean> then
     <commands>
fi;
```

In this case the <boolean> can be either a conditional expression (as in this script) or a command. In the first case, <commands> will execute if the conditional expression evaluates true. For example, the following expression executes lines 5 and 6 if and only if $line1 is different from $okline:

```
if [ $line1 != $okline ]; then
   echo Hacker detected! ${line1#$okline} ¦ mail root
   echo Hacker detected, check your mail!
  fi;
```

Similarly this expression will output Directories swapped successfully if the pushd command ran without errors:

```
if pushd
then
     echo Directories swapped successfully
fi;
```

### *if-then-else Statements*

An extension of `if` statements is the `if-then-else` statement, which allows you to take different paths depending on how `<boolean>` evaluates at each step. This takes the form

```
if <condition or command>
then
     <commands>
else
     <other commands>
fi;
```

This statement performs `<commands>` if `<condition or command>` is true and `<other commands>` if `<condition or command>` is false. For example,

```
if pushd
then
     echo Directories swapped successfully — now $PWD
else
     echo Directories not swapped
fi;
```

You can take this one step further using the `elif` keyword, which evaluates a new condition if the previous one failed. For example,

```
if pushd then
 echo Directories swapped successfully -- New directory is $PWD
elif popd then
 echo New directory is $PWD
else
 echo Directories not swapped
fi;
```

If `pushd` operates correctly, this executes line 2. If `pushd` does not operate correctly, it performs `popd`, and if `popd` executes correctly, this executes line 4. Otherwise this executes line 6.

## *case* Statements

If you have a large number of conditions to check, it might be more efficient to use a `case` statement instead of a series of `if-elif-else` statements. The format of a `case` statement is as follows:

```
case <expression> in
     <pattern1> ) <commands> ;;
     <pattern2> ) <commands> ;;
     ...
esac
```

Multiple conditions can be attached to the same set of commands using the operator:

```
case <expression> in
     <pattern1> ¦ <pattern2> ) <commands> ;;
     ...
esac
```

For example,

```
function printfiletype
     {
```

```
case $1 in
*.jpg  ¦ *.jpeg ) echo File $1 is a JPEG
*.c ) echo File $1 is a C source file
    *.cpp ¦ *.C ¦ *.cxx ) echo File $1 is a C++ source file
...
* ) echo File $1 has no known extension
esac
}
```

## *while* and *until* Statements

`while` and `until` statements repeat a set of commands until some condition is satisfied (for `until` statements) or not satisfied (for `while` statements). The form for both is the same:

```
while <condition or command>
    do
    <commands>
    done
until <condition or command>
    do
    <commands>
    done
```

The difference is that a `while` statement repeats `<commands>` until the condition is false or the command encounters an error, whereas the `until` statement repeats `<commands>` until the condition is true or the command executes successfully. For example,

```
while popd
    do
    echo New directory is $PWD
    done
```

This continues to switch to a new directory in the directory stack until the directory stack is empty.

## *for* Statements

A `for` statement repeats a set of commands for each item in a list. Note that in languages such as C, Pascal, or FORTRAN, `for` statements are used to repeat a series of statements a set number of times; for example,

```
(* this is a Pascal for loop *)
for i := 1 to N do
    begin
    writeln("Iteration ",i);
    end;
```

By contrast, `for` loops in shell scripts are not constructed to do this. Instead, you would have to use a `while` loop, such as this:

```
#this does the exact same thing as the Pascal code fragment above
i = 1
while [ $i -le $N ]; do
    echo Iteration $i
```

```
    i=$(($i+1))
    done
```

Instead, a `for` loop takes each item on a list and performs operations on that item. For example,

```
#!/bin/bash
#this code takes a complete short listing of every directory in
#the directory stack and prints it out.
for dir in $DIRSTACK
do
    ls -CF $dir ¦ lpr -Pmonica
done
```

## *select* Statements

`select` statements are a special case, unique to the Korn and Bash shells.

Additionally, only Bash shell versions 2.0 or later have `select` statements.

The format for a `select` statement is as follows:

```
select <var> in <list>
    do
    <commands using <var>>
    done
```

This statement does the following:

1. Generates a menu of each item in `<list>` with a unique number for each selection
2. Prompts the user for a number, using the prompt string in `PS3`
3. Stores the selected choice in `<var>` and the choice number in `REPLY`
4. Executes `<commands using <var>>`
5. Repeats forever, or until it encounters a `break` statement

For example, the following code allows you to select a directory from `CLASSPATH` and give a complete listing of its contents:

```
select dir1 from $CLASSPATH; do
    if [ $dir1 ]; then
        ls -CF $dir1
    else
        echo "Invalid selection; quitting"
        break
    fi
    done
```

# Input and Output

Input and output are easier in Bash than in compiled languages such as C. Input and output are handled using four constructs: redirection, the `echo` keyword, the `read` keyword, and the `select` statement.

## Redirection

Redirection is a UNIX feature that allows files and streams to be used in place of standard input and output. The most commonly used redirector is the pipe (¦), which allows the output of one process to be the input of another. For example, the following command takes the output of the `ls` command and uses `less` to scroll through it:

```
ls -la ¦ less
```

Most redirectors are used for turning a file into standard input, standard output, or error output. This is handy in cases where the size of the required input or output is very large. For example, if you have a standard letter `mail1.txt` that you need to send to multiple users, you can type the following instead of having to invoke `mail` and type the letter out again:

```
mail <user> < mail1.txt
```

The most commonly used redirectors are the following; a complete list is given in Appendix D, "A Quick Guide to Bash Shell Redirection."

| Redirector | Example | Use |
|---|---|---|
| ¦ | `<c1> ¦ <c2>` | Standard output of command `<c1>` is used as input for command `<c2>`. |
| > | `<c1> > <file>` | Standard output of `<c1>` is placed in `<file>`. |
| < | `<c1> < <file>` | `<file>` is used as standard input for `<c1>`. |
| >> | `<c1> >> <file>` | Standard output of `<c1>` is appended to the end of `<file>`. |
| 2> | `<c1> 2> <file>` | Error output of `<c1>` is placed in `<file>`. |
| 2>> | `<c1> 2>> <file>` | Error output of `<c1>` is appended to the end of `<file>`. |

## *echo* Statements

The `echo` keyword takes a line of text and prints it to standard output.

For example,

```
echo Hello World
```

prints out the following:

```
Hello World
```

Some characters are not easily printable using the `echo` command. For example,

```
echo Hello World!!
```

produces

```
Hello World<last command>
```

In order to print the string `Hello World!!`, you need to surround it with single quotes:

```
echo 'Hello World!!'
```

Surrounding with double quotes helps to denote where the string begins and ends, but otherwise the symbols within are read just as if they were shell variables.

The following command places the string "Hello World<last command>" in file1:

```
echo "Hello World!!" > file1
```

In order to print out variable values, you need to use the $ symbol before the variable name. So, for example, the following command returns the string PATH:

```
echo PATH
```

In order to get the value of path, you need to state it as follows:

```
echo $PATH
```

## *read* Statements

read statements are used to take information from standard input and place it into a variable. For example,

```
read var1
```

## *select* Statements

Now suppose that you wanted to give a user a choice of several options at some point in a program. For example,

```
good=0
while [ good -eq 0 ]; do
echo "1) Ice Cream"
echo "2) Chocolate Cake"
echo "3) Fruit Plate"
echo 'What would you like for desert?'
read choice
case $choice in
     0 ) serve "Ice Cream"  ;;
     1 ) serve "Chocolate Cake" ;;
     2 ) serve "Fruit Plate" ;;
     * ) good=1 ;;
      esac
done
```

```
This program may be more easily written as:
options="Ice Cream:Chocolate Cake:Fruit Plate:"
PS3='What would you like for desert?'
select choice in $options; do
     if [ $selection ]; then
          serve $selection
     else
          break
     fi
done
```

More details on select statements can be found in the section “Flow Control,” earlier in this appendix.

# Process Control

Because UNIX (and Linux) are multitasking, multiuser environments, it is quite helpful that Bash has a great deal of multitasking control built in to it. Whereas multitasking control in Windows 95/98 and NT is controlled by dialog boxes, multitasking in the Bash shell is controlled by command line.

## Jobs Listing

In order to obtain a list of currently running jobs, you can use two commands: `jobs` and `ps`.

The `jobs` command gives a list of jobs running from the present shell session. Unless there are background jobs (discussed later in this appendix), this will usually return nothing. If there are background jobs, this will return a list of background jobs, with an index number, a process ID number (PID), and a command line for each. For example,

```
$ jobs -l
[1]        132 Running         netscape &
[2]-       287 Running         /usr/local/bin/kvirc &
[3]+       410 Running         lpr -Pmonica letter4.txt
```

Now examine this closely.

The `[n]` is the index number for each process. Next to the `[n]` is nothing, `-`, or `+`. The `+` indicates that this process is the last command executed. The `-` indicates the second to last command executed. After the `[n]` is a number with no brackets. This usually has three or four digits (or more if the machine has been up for a long time). This number is the PID (process ID) for that process. This appears only when you use the `-l` option with `jobs`. After the PID comes a word to indicate the status of a job. Usually a job will have the status `Running`. Other statuses for a job include the following:

| Status | Meaning |
|---|---|
| `Stopped` | The process has been suspended; resume with `fg` or `bg`. |
| `Done` | The process has completed and exited with no errors. |
| `Exit <num>` | The process has exited with a nonzero signal (which usually means something has gone wrong). |
| `Terminated` | The process has received a `SIGTERM` signal and has ceased. Note that `SIGTERM` can be caught by the process, and instead of terminating, the process will do something else. |
| `Killed` | The process has received a `SIGKILL` signal and has ceased. Note that `SIGKILL` signals cannot be caught; they automatically stop. |

The other way to view jobs is to use the `ps` command. This gives a snapshot of processes running at any particular time. For more information on `ps`, see the man pages.

## Background and Foreground

Processes are divided into two basic types as far as the user is concerned: background processes and foreground processes. Background processes run without tying up the shell command line, allowing you to perform other tasks while the process executes. Foreground processes, on the other hand, tie up the shell until execution is complete.

Certain processes simply are unsuited to running in the background. For example, if you want to edit the file /etc/group, you would type

```
vi /etc/group
```

At this point, the vi editor is in control. It occupies the screen and the keyboard, and continues to do so until the user switches to another terminal (for example, if using the X Window System) or exits vi. On the other hand, some processes can run without tying up a Bash session, and there is no reason to let them tie up a Bash session. For example, the Netscape browser can be invoked using

```
netscape &
```

and run quite well without tying up a Bash session.

## Naming Processes

The Bash shell contains several devices for naming processes. The most basic way to name a process is by process ID; for example, if Netscape has PID 441, you could kill Netscape using the command

```
kill -9 441
```

If a job is a background process, there are other ways to refer to it:

| Name | Meaning | Example |
|---|---|---|
| %n | Job with index number n | %1 |
| %<string> | Job started with command <string> | %netscape |
| %?<string> | Job started with a command containing <string> | %?netsc |
| %+ and %% | Most recently started background job | %+ |
| %- | Second most recently started background job | %- |

Thus `kill -9 %netscape` or `kill -9 %?netsc` are the same as `kill -9 441` in the earlier example.

## Starting and Stopping Processes

To stop a job while it is running, you need to send it a stop signal. This is usually done with Ctrl-C (to exit) or Ctrl-Z (to stop and resume later). Stopping a process to resume later keeps the process in the process list, but the process doesn't occupy any more processor resources. (Resources such as memory and disk space are still used.) Another way to stop a job is with the `kill` command. The `kill` command sends a signal to the named process, causing it to halt or

pause. See the man pages for more information on `kill`. To resume a stopped job, use either the `fg` or `bg` command. The `fg` command resumes the job in the foreground whereas the `bg` command resumes the job in the background. Usually you need to give a name for the process, as in the following:

```
bg %?netsc
```

Otherwise, the shell assumes that you are resuming the last stopped job. So if you have three background jobs, like the following, the `bg` command will resume process `[3]`:

```
[1]    Stopped         netscape &
[2]-    Stopped         /usr/local/bin/kvirc &
[3]+    Stopped         lpr -Pmonica letter4.txt
```

Note that `fg %+` and `bg %+` are thus the same as `fg` and `bg`, respectively.

## Signals and Signal Catching

You've already had some discussion of signals as ways to stop or exit processes. This section discusses how to generate and catch signals.

There are two ways to generate signals. The first is by using the keyboard; the second is with the `kill` command.

If a process is running in the foreground, you can generate a signal with the keyboard, by pressing Ctrl-C, Ctrl-Z, or Ctrl-\ (backslash). These keys have the following meanings:

| | | |
|---|---|---|
| Ctrl-C | `INT` (Interrupt signal) | Halts process |
| Ctrl-Z | `TSTP` (Terminal stop signal) | Freezes a process; available to resume execution later. |
| Ctrl-\ | `QUIT` (Quit signal) | Exits process |

The `kill` command syntax looks like this:

```
kill <options> <process>
```

You can use the `kill` command to kill any process, not just processes running in the foreground.

The usual signal sent by `kill` is the `TERM` signal. For example, if you type

```
kill %netscape
```

you should get back something like this:

```
[1]+ Terminated     netscape
```

Other signals you can send are `QUIT`, `TSTP`, and `KILL`. For example, to send the `KILL` signal, you would type:

```
kill -KILL %netscape
```

and you should get back something like:

```
[1]+ Killed          netscape
```

You can catch signals in shell scripts using the `trap` command. The syntax of a `trap` command is as follows:

```
trap <action> <signal1> (<signal2> ... <signalN>)
```

For example, inserting the following into a script would catch Ctrl-C, Ctrl-Z and Ctrl-\, as well as the regular `kill` command, and print a message to the screen:

```
trap "echo 'Unable to exit right now. Please be patient'" INT TERM TSTS QUIT
```

The script would resume as normal.

Note that the one command you cannot catch is the `KILL` command. No matter what else you try to catch, the `KILL` command will always end a process.

# Sample Shell Scripts

The following are three sample shell scripts for your information. These were constructed over the years by David Yeske, an employee of the Linux General Store, with some comments from other employees of the store. Feel free to borrow from these as you like, in the spirit of open source.

```
================================================================
#
#sample .bashrc
#written by David Yeske, Linux General Store
#
#this sets the backspace key to erase
stty erase ^H
export DISPLAY
export EDITOR=vi
export ENV=~/.bashrc
export FCEDIT=vi
#
#history file information. Note that a history of 50000 lines is seen
#by some as excessive :)
#
export HISTFILE=~/.bash_history
export HISTSIZE=50000
export HISTFILESIZE=100000
#
#checks for mail every second. This, too, is seen by some as excessive :)
#
export MAILCHECK=1
export MANPATH=~/man:~/pub/man:/usr/local/man:/usr/share/man:/usr/X11/man
# This tells the man command what to use to scroll through a file.
export PAGER="less -e -i -M"
#
#The PATH is the list of directories that bash looks through to find a
#command.
#
export PATH=$HOME/script:$HOME/bin:$HOME/pub:/bin:/usr/local/bin:/usr/bin:
➥/usr/sbin:/sbin:/local/bin:/usr/local:/usr/etc:/usr/X11/bin:/usr/local/sbin
#
# The prompt. This provides a heck of a lot of information to the user.
```

```
# For more information on prompt formatting, see the bash man page.
#
export PS1="{\u}:{`uname`}:{`uname -r`}:{\h}:{\t}:{`tty`}:{\w}:{\!}:{\#}\n>"
export SPELL=ispell
export SHELL=/bin/bash
#export TERM=screen
#
# Any time that you execute a command at the prompt, this command is executed.
#
export PROMPT_COMMAND='history -a'
#
# a list of shell options
#
shopt -s histappend cdspell checkwinsize cmdhist
history -n
#
# Note: the lines "history -n", "shopt -s histappend ..." and
# "export PROMPT_COMMAND ..." allow us to preserve the command
# history across multiple bash shell sessions.
#
#
# This allows us to use vi command keys to search through the history,
# perform command line editing, &c. Similarly, "set -o emacs" would
# allow us to use emacs keys.
#
set -o vi
umask 077

#
# The sign of a true bash hacker! This line first checks to see if the
# file .bash_aliases is a text file, and if it does, then it calls the
# script .bash_aliases.
#
[ -f ~/.bash_aliases ] && . ~/.bash_aliases
#
# this is equivalent to:
# if [ -f ~/.bash_aliases ]; then
#    ~/.bash_aliases
#    fi
#                  cool, huh? :)
============================================================
#
# sample .bash_aliases file
# written by David Yeske, Linux General Store
# Contains some useful aliases.
#
alias texclean='rm -f *.toc *.aux *.log *.cp *.fn *.tp *.vr *.pg *.ky'
#
# Notice that this alias consists of several commands, bound by single
# quotes on either side. This is equivalent to defining a function
# to handle all of this and then aliasing the function.
#
alias clean='echo -n "Really clean this directory?";
    read yorn;
    if test "$yorn" = "y"; then
       rm -f \#* *~ .*~ *.bak .*.bak  *.tmp .*.tmp core a.out;
```

```
        echo "Cleaned.";
    else
        echo "Not cleaned.";
    fi'
#
# Notice that a lot of these aliases are simply a matter of cutting down
# on keystrokes for commonly used commands. This is especially useful if
# you're using a bash shell over a very slow or lag-ridden connection.
#
alias h='history'
alias j="jobs -l"
alias l="ls -l "
alias ll="ls -l"
alias ls="ls -F"
alias pu="pushd"
alias po="popd"
#
# One common Linux mistake is to type anagrams of commands. While this
# mistake is not usually damaging, it is rather annoying. The following
# is an example of how one anagram mistake, typing "dc" instead of "cd,"
# becomes corrected.
#
alias dc=cd
alias f=finger
alias go=screen
#
# This prints out a list of processes that the current user is running.
#
alias me="ps -xau ¦ grep $LOGNAME"
alias md="mkdir"
alias more="less -e -i -M"
alias moreman="nroff $1 ¦ more"
alias spy="ps -xau ¦ grep -i $*"
alias rd="rmdir"
alias x="cp -i"
alias z="ls -laF $*"
#
# Csh compatibility:
#
alias unsetenv=unset
function setenv () {
  export $1="$2"
}
# Function which adds an alias to the current shell and to
# the ~/.bash_aliases file.
add-alias ()
{
   local name=$1 value="$2"
   echo alias $name=\'$value\' >>~/.bash_aliases
   eval alias $name=\'$value\'
   alias $name
}
#
# This sets up a special command for editing the bash resource scripts.
#
config ()
```

```
{
chmod 700 ~/.bashrc ~/.bash_profile ~/.bash_aliases
vi ~/.bashrc ~/.bash_profile ~/.bash_aliases
chmod 500 ~/.bashrc ~/.bash_profile ~/.bash_aliases
}
# "repeat" command.  Like:
#
#    repeat 10 echo foo
repeat ()
{
    local count="$1" i;
    shift;
    for i in $(seq 1 "$count");
    do
        eval "$@";
    done
}
# Sub-function needed by `repeat'.
seq ()
{
    local lower upper output;
    lower=$1 upper=$2;
    if [ $lower -ge $upper ]; then return; fi
    while [ $lower -le $upper ];
    do
    echo -n "$lower "
        lower=$(($lower + 1))
    done
    echo "$lower"
}
================================================================================
#
#sample .bash_profile
#written by David Yeske, Linux General Store
#
if [ -f ~/.bashrc ] ; then
        . ~/.bashrc
fi
mesg y
export ENV=~/.bashrc
screen -list
```