Shells and Scripting: Section 1 Transcript

Introduction

Welcome to the Shells and Scripting module. Unlike Windows systems, the primary interface to a UNIX system is the command-line interface, called a shell. UNIX shells are particularly powerful and flexible, capable of handling a variety of needs from a single-user desktop interface to job management on a multi-processing mainframe. Another important feature of shells is the ability to automate many common administrative tasks to inspect or change configurations, and start and manage jobs and Internet services. Remote operations on any UNIX system require the operator to be knowledgeable of and capable with using UNIX shells.

Throughout this module, you'll be presented with opportunities to assess and apply what you've learned.

At the end of this module, you will be able to:

- Explain the purpose of a shell in a UNIX operating system,
- Identify commonly used shells and their unique core characteristics,
- Identify features and functions of the shell environment,
- Recognize the use of patterns in writing commands for different shells, and
- Demonstrate basic familiarity with shell script programming.

Prerequisites

As you are traversing through this module, you are expected to use Internet search engines and UNIX "man" pages to discover and learn about a variety of UNIX commands. Some of the commands you need to research and become proficient with are: `alias`, `bg`, `cd`, `env`, `eval`, `export`, `fg`, `history`, `jobs`, `nohup`, `popd`, `pushd`, `set`, `setenv`, `trap`, `unset`, and `unsetenv`.

You also need to be proficient with some basic UNIX syntax. This includes redirecting input and output, using quote characters, history, and file command substitutions.

Before continuing on, take the time to research the UNIX commands displayed - especially any that may be unfamiliar to you. You may print a copy of the commands (found in Resources for this module) and/or record data you learn during your research.

Click the Next Slide button when you are ready to continue.

Bypass Exam Introduction

If you are already familiar with the subject matter presented in this module, you can choose to take a Bypass Exam to skip this module.

The Bypass Exam option provides a single opportunity to successfully demonstrate your competence with the material presented within the module. If you pass, you'll receive credit for

completing the module, unlocking the content within, and you will be free to proceed to the next module. If you do not pass, you will need to successfully complete the module, including all exercises and the Module Exam, to receive credit.

Click the Next Section button to continue.

Shells and Scripting: Section 2 Transcript

What is a Shell? 1/11

While most UNIX systems are capable of supporting a graphical user interface, the typical user environment for a UNIX system is a command line-oriented process called a shell. There are a number of different shell programs available, whose feature sets we will discuss shortly, but the common functions of a shell allow the user to:

- Execute programs stored in the file system by leveraging appropriate system calls,
- Navigate and work with objects in the filesystem,
- Type input to an executing process, and
- Display the output of an executing process.

Understanding and using shells is a critical operational skill for UNIX systems, and an important part of that understanding is knowledge of how shells are typically created. We begin our investigation with exploring how new processes are spawned, an important function of shells, and then show the typical ways the shell itself is spawned.

But first, we're going to look at different types of shells that occur in UNIX systems.

Different Types of Shells 2/11

There are several different types of shells available in UNIX operating systems. Most systems have more than one type of shell, but you may not always be able to choose what type of shell you can use based on what is installed on the system.

The Bourne shell was developed by Stephen Bourne at Bell Labs, and released in 1977. It was intended as a scripting language. It is not often used today.

The C shell was created by Bill Joy while he was a graduate student at University of California, Berkeley, in the late 1970s. It has been distributed widely, beginning with the 2BSD release of the BSD UNIX system that Joy began distributing in 1978. The C shell uses C programming language as a syntax model.

The Korn shell has two versions distinguished by the year they were released ksh88 and ksh93. Both versions contain command line editing.

The Tenex C shell is a newer version of the C shell, uses a C-like syntax, and can be more convenient than the other shells for interactive users.

The Bourne-Again-Shell (BASH) is the most common shell in use today. BASH was developed as a GNU project and incorporates features from the Bourne shell, the Korn shell, and the C shell. It contains command line editing.

The Z shell is an interactive login shell and a shell script command processor. The Z shell is an

extended version of the BASH shell, and closely resembles the Korn shell. The Z shell has many improvements of some features from the BASH, Korn, and C shells. It also has enhancements including command line editing, built in spelling correction, command completion, shell functions, and module loading.

It's important to know which shell you are in because the commands and syntax for different shells are different.

Identifying Your Shell

The `/etc/passwd` (commonly referred to as "etc password") file determines which shell takes effect during your UNIX session. When you log in, the system checks your entry in `/etc/passwd`. The last field of each entry names a program to run as the default shell.

To determine what shell you are currently using, run the following command: `ps -p $$`

You can change to another shell by typing the program name at the command line. For example, to change from Bourne shell to the Korn shell type: `exec /bin/ksh`.

Using a different shell is not always permitted. Sometimes changing shells is just a personal preference. Other times, using a specific type of shell makes certain kinds of interactions easier. For example, a C shell is often recommended for interactive use, and the Bourne shell for scripting.

You can find links to the source codes for shells online. Let's look at what happens when you first start a UNIX session by logging in.

Fork and Exec

On a UNIX system, every process except the first, usually called `init`, is started by another process. There are two key system calls that a process uses to create another process: *fork()* and *exec()*.

The *fork()* system call uses the kernel to create an identical copy of the currently running process. This new process, called the child, has the same running code and the same memory contents as the original process, called the parent. The main difference between the two processes is the return value of the *fork()* system call: the parent receives an integer, which is the process ID (PID) of the child, while the child receives the value 0.

The return values allow the process to know whether it is the parent (the return value is not zero) or the child (the return value is zero).

In isolation the *fork()* system call simply causes a process to create copies of itself. But a typical, though not universal, follow up to a *fork()* is for the child process to run the *exec()* system call. The *exec()* system call uses the calling process to load a new program into its memory and start executing the new program from the beginning. This combination of *fork()* and *exec()* is what allows UNIX systems to run a broad array of programs in a controlled manner.

Linux and Solaris both use *fork()* and *exec()* to initiate login.

Typical Console Login for Linux

For systems such as Linux that use the System Five (SYS V) `init` for booting, the first process to start is `init`. A typical function for `init` is to start up consoles, for example the physical terminals used to interact with the system using the keyboard and screen. Classically, this function is called **getty**, which stands for "get TTY." The modern Linux implementation is called **mingetty**. `Init` performs this using *fork()* then *exec()* system calls to create mingetty.

After the user enters their username and presses enter, mingetty makes an *exec()* system call, and loads the login program. Since there is no *fork()*, the former mingetty process simply becomes the login process, and there is really no way to tell at this point that it used to be mingetty.

When the user enters their password, login verifies the password is correct. If so, login determines the user's default shell, as defined in the `/etc/passwd` file and performs a *fork()* and *exec()* to start that shell program. Control of the keyboard and screen passes to the shell, but the login process remains running for the duration of the user's session.

When the user runs a command such as `ls`, the shell performs a *fork()* and *exec()*, creating a child that loads and runs the `ls` program. When ls completes, the process simply ends, and control is passed back to the shell.

Now let's look at a typical console login for Solaris 10.

Typical Console Login for Solaris 10

Solaris uses the System Five (SYSV) `init` for booting, but only to start up its own **Service Management Framework (SMF)**. SMF is in charge of starting all other services, including **ttymon**, using *fork()* and *exec()*.

Much like Linux, when the user enters their username, ttymon makes an *exec()* system call and loads the login program.

On Solaris 10, when the user types their password, login does not *fork()* and *exec()* to create the shell; rather, it simply calls *exec()* and replaces itself with the shell. The login program is no longer running.

Another variation of the login process occurs during a remote login.

Typical Login for SSH Solaris 10

Just like ttymon, the sshd service is started by SMF using *fork()* and *exec()*. When a client connects to the **SSH** service, the sshd service performs two *fork()* and *exec()*s to create three sshd processes. Once the user is authenticated, the sshd process creates the shell for the user using *fork()* and *exec()*.

Notice that the login program was never executed during this login procedure. This indicates that sshd itself is responsible for checking the user's password and authenticating the user.

After users are logged into the system, they can interact with files.

File Descriptors

Whenever a process wants to access a file, it requests access from the kernel using the *open()* system call. If the request is successful, the kernel returns a file descriptor, which is an integer value. The process uses the file descriptor for subsequent reads or writes to the file.

By default, when any process on a UNIX system is started, it is connected to three standard file descriptors. They are:

Standard Input, `STDIN`, which has a file descriptor value of 0. By default, standard input is connected to the keyboard so that keystrokes are fed as input into the process. Usually only one process is attached to the keyboard input at one time, so only that active process receives input from the keyboard on standard input.

Standard Output, `STDOUT`, has a file descriptor value of 1. By default, standard output is directed to the terminal so that output from the process is printed to the screen. Unlike `STDIN`, multiple processes can print to the screen at the same time. Having multiple processes output to the same screen might be confusing at times.

`STDERR`, or Standard Error, has a file descriptor value of 2. By default, standard error is also directed to the terminal so that errors from the process are also printed to the screen. This separation of standard output and standard error output is useful when redirecting one or both of these data streams, so that even if a user has stopped standard output from being printed to the screen, the user can still see errors.

Now that you know which file resources a process opens by default, let's examine how the shell can redirect these file descriptors into files and devices, and even to or from other processes.

Input/Output Redirection

Most commands in UNIX systems use the terminal to output their result and print to the screen as `STDOUT`. UNIX commands also use the terminal as source for their input. When the terminal is used as the input source, it is the standard input or `STDIN`.

You can bypass that behavior by using Input/Output or I/O redirections. I/O redirection lets you use a file for input or output. It also allows you to make the output of one command the input for another command. To redirect `STDOUT` use a greater than sign, referred to as "output" or "output to."

For example, if you wanted to redirect the output of the echo command into a file, you would type:
`echo hello > output.txt`

If the specified output file doesn't exist, using "output to" will create it. If the file does exist, it will be completely overwritten by the command. To avoid this, you can use two greater than signs, which is known as "append" or "append to" which will simply add the output from `STDOUT` to the file.

For example, `echo world >> output.txt`

To redirect `STDIN`, use the lesser than sign, referred to as "input" or "input from."

For example, `grep hello < output.txt`

Now that you know how the shell can redirect these file descriptors into files and devices, and even to or from other processes, we are going to examine what happens when a shell starts a new process.

Job Control

When a shell starts a new process, the shell keeps track of the process as a job. All of the jobs the shell starts are associated with the shell's terminal so that each of its jobs can potentially use the keyboard as `STDIN`, and the screen as `STDOUT`.

To keep things orderly, the shell usually allows only a single job, including the shell itself, to receive input from the keyboard at any time. This is called the foreground job. By default, when you run a command in the shell, that command becomes the foreground job.

If a job is running in the foreground and you want the shell to take control back, you need to stop the job. To stop a job, press Ctrl+Z on the keyboard. This causes the foreground job to stop execution, but the process does not terminate. It is now in a suspended state.

If you want this job to continue running, but not have control of the terminal, you can run it in the background. When a background job is running, it cannot receive input from `STDIN`. However, it can print `STDOUT` and `STDERR` to the screen. To start a stopped job in the background, use the `bg` command. Similarly, to move a stopped job to the foreground, use the `fg` command.

It is possible to start a job directly in the background by appending an ampersand (&) to the end of the command invocation, making sure to leave a space after the last argument.

To view a list of all jobs tracked by the shell, you can issue the `jobs` command. The job numbers can be used with the `fg` and `bg` commands to manage multiple jobs.

When a shell is terminated, through either a logout or other mechanism, all of the jobs controlled by that shell are terminated. For certain long running jobs, this may not be desirable. If you are starting a new job and don't want it to be tracked by the shell, you can preface your command invocation with the `nohup` command, which you'll want to put in the background. On the other hand, if you have a currently running job you would like to detach from the shell, place it in the background and then use the `disown` command to remove it from the controlling shell.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Shells and Scripting: Section 3 Transcript

Shell Expansion

Every time you type a command line and press enter, the shell takes several actions on the input before it carries out your command. The actions are referred to as expansion. What happens is that what you type is interpreted by the shell and expanded into a literal representation of command before the shell acts on it.

For example, if you type a command using a wildcard character, the shell literally creates strings for every variation that matches and then executes on each one before outputting the result.

Shells can expand text, pathnames, and by using command substitution. Click each tab to learn more about the different ways that shell expansion occurs. Whenever possible, try out each example on your VM or make up your own to discover what happens. When you have finished reviewing each set, click the Next Slide button to continue with the module.

Text: The shell can expand to include text strings when they are enclosed within curly brackets called braces. Brace expansion can have a preamble, in this example "r" followed by multiple text strings within the braces separated by commas, and ending with postscript, in this example "d". The preamble is prefixed to each string contained within the braces, and the postscript is appended to each resulting string, expanding from left to right.

Brace expansions can be nested. Brace expansion is performed before any other expansions. Any characters special to other expansions are preserved in the result.

Click another tab to continue learning about shell expansion. If you have finished shell expansion, click the Next Slide button to continue with the module

Pathnames: The shell can expand to include filenames using square brackets. Pathname expansion uses the directory structure to locate filenames. Pathname expansion matches any one of the characters enclosed within square brackets. You can use a hyphen between characters to specify a range. Pathname expansion can have a preamble, in the top example "a" followed by a range of characters, and ending with postscript, in the top example "e". The preamble is the first part of the filename, and the postscript is the last part of the file name.

Click another tab to continue learning about shell expansion. If you have finished shell expansion, click the Next Slide button to continue with the module

Command Substitution: The shell can expand using command substitution making the output of a command replace the command itself. The output is then stored to a variable or displayed using `echo`. The classic form of command substitution uses backquotes called "backticks" (`...`). In the BASH shell, `$( )` can be used instead of backticks. The following formats are equivalent: `$(command)` and `` `command` `` in a BASH shell.

To get the date to display on the screen, type `echo "today is $(date)"` or type `echo`

```
"today is `date`"
```

Before moving to the next slide, take a few moments to try some command substitutions of your own.

Click another tab to continue learning about shell expansion. If you have finished shell expansion, click the Next Slide button to continue with the module

Pattern Expansion

To review, shell expansion uses special characters and syntax as shortcuts that are expanded by the shell before executing a command. You can make shell expansion even more powerful by using pattern matching expressions in the BASH shell to include lists.

To do so, you must first set the `extglob` shell option by typing the following command: `shopt -s extglob`

After that command runs, it becomes possible to match patterns within a list in addition to standard asterisks, question marks, and square brackets. Patterns in a list need to be separated by a vertical bar (or pipe) and grouped with parenthesis.

The first example shows calls for everything except what is included in the list. Because the list has only one file, the command will execute against all files but that one.

The second example calls for a list of all the files that start with `install` or `uninstall` and end with the `.log` or `.sh` extensions.

The third example calls for a list of all files except those with `.jpg` and `.gif` extensions.

As powerful as pattern expansion is, there are times when you need to find files and folders on a system and may not know where to look. This is where a tool called GREP comes in handy.

Using GREP

The GREP program is a tool that can search through files and folders and check which lines in those files match a given string. "GREP" then outputs the filenames and the line numbers or the actual lines where the match was found. "GREP" is a useful tool for locating information stored anywhere on your computer, especially if you do not know where to look.

"GREP" also supports the use of patterns called regular expressions. Those regular expressions closely resemble the extended pattern matching syntax that is normally used on the shell with minor differences. The usage syntax consists of typing "grep"; followed by options; then the pattern; and finally the file, folder, or list of files and folders to search.

There is a list of regular expressions you can use with GREP in the Resources for this module.

Now that you have a solid understanding of shell and pattern expansion, let's look at another expansion feature of shells, the command history.

Shell Command History

The shell command history is a list that contains all the commands that were typed during a session. In most shells, that list is saved in a file, called a history file, whenever a user logs out of his session or exits the shell. The same history file is used to populate the history list when the user logs in. Many environment variables are used to control how the command history is handled by the shell.

For the BASH shell for instance, the following environment variables are used:

- `HISTFILE` - The name of the file in which command history is saved. The default value is `~/.bash_history`. If this variable is undefined, the command history is not saved when an interactive shell exits.
- `HISTFILESIZE` - The maximum number of lines contained in the history file. When this variable is assigned a value, the history file is truncated by removing the oldest entries to contain no more than the number of lines specified.
- `HISTSIZE` - The number of commands to remember in the command history list.

The C shell uses different environment variables to affect the behavior of the history:

- The "history" environment variable is used to define the maximum size of the history list.
- The "savehist" environment variable determines if the history is saved when the user logs out. Since savehist is not set by default, the C shell does not record the command history by default.
- The "histfile" environment variable sets the default location for the history file. When the variable is undefined or unset, `~/.history` is the default location of the history file.

The BASH and C shells do have similar commands to interact with the history list:

- The "history" command displays the command line history list.
- Two !'s relays the last command you typed, whereas a single "!" followed by a number executes the command with that number in the history list.

Shell command history allows previous command information to be used in current execution. You can also interact with the environment variables to modify command history settings. For example, in BASH, `unset HISTFILE` prevents the history file from being saved when the user logs out of a session. In the C shell, this can be accomplished by running `unset savehist`.

Now let's look at shell variables, which allow you to remember many other types of information as you execute on the command line.

Shell Variables

A shell variable is a dynamic named string that can be used to store a value. Most shells use shell variables for session customization purposes, but they can also be used to temporarily hold values for future usage. One very powerful use of shell variables is for programming in shell scripts, covered later in this module.

Shell variables are contained exclusively within the shell in which they were set or defined. There are different types of shell variables:

- **Built-in shell variables** - Built in shell variables are automatically set up by the shell. The shell defines those by default when it is started without any external input. A few examples of BASH built-in variables are:
    - `BASH`: Which expands to the full file name used to invoke the current instance of BASH,
    - `HOSTNAME`: Which is automatically set to the name of the current host, and
    - `PWD`: The present working directory.

  Additionally, the BASH shell is equipped with various special variables that have special meaning with the shell or can be used to customize the user environment.
- **Global or environmental shell variables** - After the login program invokes your shell, the shell sets up your environment variables by reading and executing startup files and scripts. These files normally set the type of terminal in the variable `$TERM`, the default path that is searched for executable files in the `$PATH` variable, the command prompt string, and so forth. Global or environmental shell variables are inherited by child processes. Notice that the Bourne family uses the `export` command to mark variables as global after they have been defined, whereas the C shell family uses `setenv` to define global variables from the beginning.
- **Local shell variables** - Local variables are only accessible within the current shell. You can edit startup scripts to manipulate your startup environment or add additional variables as needed. You can also create conditional environment settings with shell programming. When a variable name exists in both local and global lists, references will use the local variable.

We are now going to discuss different uses for shell variables and their syntax.

Manipulating Variables Using Bourne and BASH Shells

The syntax and commands to set variables are slightly different depending on the type of shell.

In a BASH shell, defining a variable is simple. Type the name of the variable followed by the equal sign. Next, type the value of the variable. After a variable is set, you can retrieve its value by typing a dollar sign ($) before its name. This is referred to as shell variable expansion. To make variables global, just use the `export` command followed by the list of variables you wish to make global. You can also use the `export` command to define and make one or more variables global within a single command.

To undefine a variable, use the `unset` command followed by the name of the variable. After doing so, the variable will no longer be available.

In the attached example, we demonstrate how a local variable and a global variable are defined and displayed. We also show the availability of each type of variable depending on when we move to a subshell, and we finally show the unavailability of each type of variable after they have been unset.

Other operations can be performed by using variants of the variable expansion syntax such as:

- Determining the number of characters in the variables (or string length),
- Turning the string held by the variable to upper case,
- Turning the string held by the variable to lower case,
- Setting the variable to a default value, and
- Perform pattern matching for substring expansion, substring removal, or for substring search

and replace operations.

See the Supplemental Material for examples of variable expansion syntax. Now, let's look the syntax for variables within a C shell.

Manipulating Variables Using the C Shell

Manipulating variables in the C shell environment is slightly different than the Bourne and BASH shell environments. For instance, to define a local variable, use the `set` command. To define a global variable, use the `setenv` command instead. The variable name and the value follow respectively. After a variable is set, you can retrieve its value by prepending a dollar sign ($) to its name just like in BASH.

The C shell uses the same syntax as BASH to undefine a variable but just in the case of local variables. To undefine global variables, use the `unsetenv` instead.

In the attached example, we demonstrate how a local variable and a global variable are defined and displayed. We also show the availability of each type of variable depending on when we move to a subshell, and we finally show the unavailability of each type of variable after they have been unset. Notice that using the `unset` command to undefine global variables does not work in the C shell.

Unlike BASH, the C shell offers a more limited set of variable expansion operations that can be performed. Take a few minutes to conduct some research on the different variable expansion operations offered in the C shell.

Next, we are going to examine special variables that are set internally by the shell.

Special Variables

Special variables are set internally by the shell and are available to users. Special variables are typically used inside shell scripts. Some special variables are common to all shells and many more are shell specific. Important special variables to know are:

- $1 to $9 - Positional argument parameters,
- $? - Return status of last sub-process,
- $$ - Process ID of the shell,
- $0 - Name of command, including path, that executed the current shell.

All of the variables and substitutions we have covered rely on a variety of special characters that the shell interprets. Next, we look at different ways to use special characters in a "shell safe" manner, so that they are not interpreted by the shell.

Quoting and Escaping

In UNIX shells, certain characters are reserved as special characters that expand expressions and patterns such as the "?" question mark, "!" exclamation mark, and the "*" asterisk. But what if you need to use these characters as themselves, and not have them represent their special roles in the shell?

For example, if you want the question "Is this your file?" to print on to the screen. If you enter it as it appears here, the system interprets the string as part of a script. If the current working directory contains filenames that match the pattern file?, you will print the list of matching files instead of the question. To use special characters as themselves, you must "escape" them. In UNIX systems, you escape a special character by preceding it with a backslash "\".

A backslash tells the shell to treat that character as itself and not as an instruction for shell processing. A common use for escaping is when filenames contain special characters. This often occurs when transferring files from a Windows system. Windows file names allow spaces in them, but spaces in UNIX are argument separators. You can use the backslash to tell the shell the space is literally a space instead.

In other situations you may find a line that contains many special characters. Rather than escaping every character, you can quote an entire string. Surrounding a string with single or double quotation marks tells the shell to treat everything contained inside as a single argument. Quoting also escapes characters within the string using these rules:

- Strings within single quotes escape every character in quotes without exception. In the example, the asterisk, dollar sign, and question mark all remain themselves when printed to the screen.
- Strings within double quotes escape every character except dollar sign $, backtick `, backslash \, and bang !. Strings within double quotes allow both variable and command substitution. These strings also expand based on history, but do not allow wildcard expansion. In the example, dollar sign HOME and backtick pwd expand to the path of the home and current working directory respectively.

Understanding how the shell uses variables and quoting and escaping characters allows you to analyze your shell environment.

The Shell Environment

By default, when a process is created, it inherits a duplicate environment of its parent process, except for explicit changes made by the parent when it creates the child. Similarly, every shell gets its initial environment from a combination of key **environmental initialization files** and existing **environment variables** handed down by the login process.

You can change environment variables for a particular command invocation by indirectly invoking it using `env`. Before moving on, take a few moments to research the syntax for the `env` command.

Shell Modes

Your shell type determines which configuration files are read. Login shells are those executed as part of an initial account logon to the system. Non-login shells typically run and inherit from an existing environment.

An interactive shell takes user input from a command line. A non-interactive shell (typically running a script) does not require user input unless programmatically requested.

You can tell the difference between the login and non-login shells on a process list. Login shells will either have the appropriate login option (`-l` or `--login`), or a "-" (tac) in the front of it. The first three lines in the image are login shells.

Interactive shells are more difficult to discern. There's a command line option that may provide an indicator; otherwise, you need to look at the children of that shell to make a determination. The next two lines are interactive shells.

The last line is a non-interactive, non-login shell. It is running the "tst" script. The clue here is the name `/bin/bash` with a command line argument. The user could have started the shell with `/bin/bash`, as well.

Since you are now familiar with the basics of shell modes, let's take a few moments to examine login and exit scripts, which vary for both shell type and shell mode.

Login and Exit Scripts

Login and exit scripts are scripts that the shell runs right after login and right before exit respectively. Those files vary by shell type and shell mode (interactive and non-interactive).

When BASH is executed as an interactive login shell, or as a non-interactive shell with the `--login` option, it first reads and executes commands from the file `/etc/profile`, if that file exists. After reading that file, it looks for `~/.bash_profile`, `~/.bash_login`, and `~/.profile`, in that order, and reads and executes commands from the first one that exists and is readable. The `--noprofile` option may be used when the shell is started to inhibit that behavior.

When a login shell exits, BASH reads and executes commands from the files `~/.bash_logout` and `/etc/bash.bash_logout`, if the files exist.

When an interactive shell that is not a login shell is started, BASH reads and executes commands from `~/.bashrc`, if that file exists. This may be inhibited by using the `--norc` option. The `--rcfile` option forces bash to read and execute commands from the specified file instead.

Notice that the C Shell makes no distinction between interactive and non-interactive modes for both login and non-login shells. The BASH shell does not designate between interactive and non-interactive for the login shell either. However, non-login shells can have an interactive or non-interactive mode.

Some logon scripts are not particular to any given user but available system wide for a given shell. For instance, the `/etc/profile` script will run for all the users that login and have BASH as the default shell.

The Z shell is an extended Bourne shell with some features of BASH, Korn, and C Shell. It would be easy to assume that the shell modes would be similar, but they are actually very different.

Before continuing on to aliases, take a few moments to conduct some research to discover the differences between the interactive and non-interactive modes of both the login and non-login Z shells.

Aliases

An alias is the name of a shortcut to launch a command or group of commands. Aliases are used for many reasons. To display a list of all the defined aliases on a system, type the alias command at the command prompt.

It is important to note that aliases are recognized only by the shell in which they are created, and they apply only for the user who creates them. However, the root user can create aliases for any or all users.

Additionally, aliases are typically added in the startup files so that they persist and are available to interactive subshells. For BASH use `~/.bashrc`. For Korn use `~/.kshrc`. For C shell use `~/.cshrc`.

A key takeaway about the use of aliases is the importance of using full pathnames to execute commands when operating on an unfamiliar system. While many of the alias examples show the addition of arguments to a command, there is nothing to prevent a user from typing arbitrary actions to simple commands like 'ls'. Always know the consequences of your actions and never make any assumptions about an unfamiliar system. Using full pathnames is always the safest route through the unknown.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Shells and Scripting: Section 4 Transcript

Shell Scripting

Now that you have an understanding of the shell environment and how to use patterns and variables, it's time to apply that knowledge to analyzing shell scripts. Shell scripts are short programs that are written in a shell programming language and interpreted by a shell process. They are created by writing shell commands, line by line, in a text file. **Conditional statements** and **loops** can also be added to alter the flow of execution of the program.

It is important to note that a shell script should start with a "#!" followed by the path of the shell program that runs the rest of the code. This guarantees that the script runs in a separate subshell. If that line is omitted, the script will run in the current shell.

Invoking or running a shell script is just the same as running any shell command. Arguments can even be passed to the shell script by listing them after the shell script name and separating them by spaces. Those arguments are available within the script though the positional parameters $1 through $9 and ${10} and up.

Shell scripts are extremely useful for automating tasks on UNIX operating systems. Typical operations performed by shell scripts include manipulating files, executing programs, and printing text. If a user or a system administrator runs a set of commands to perform certain actions regularly, a shell script is a very good way to cut down the typing involved. If the frequency of those actions is more deterministic, the user can even register the shell script as a `cron` job, forcing the script to run repeatedly at the specified frequency.

The following examples discuss scripting as it applies to the BASH shell. For more information about other types of shells, use your man pages or the Internet.

Conditionals

In a shell script, you can specify different actions based on the success or failure of a command using conditionals. Conditionals are mostly used to determine program flow through "if" statements and "while" loops.

What is actually being checked in a conditional statement is the return code or exit status of the command or expression being evaluated. In other words, the value of the special variable "$?" (last successful command) is being checked. If the exit status is zero, then the expression evaluates to be true. If the exit status is not zero, then the expression evaluates to be false. Note that the exit status and the output of a command are two different things. Only the latter gets displayed to the screen.

"Test" is a command line utility that can be used in conditionals to check file types and compare values. The test syntax is used to pass the expression to evaluate as an argument of the command. The use of the test command results in changes to the "$?" variable depending on the veracity of the expression. Alternatively, you can use square brackets in the syntax instead of typing "test".

The syntax for the if conditional uses the "if" keyword followed by the conditional expression, notably the "test" command. This is followed by a semi-colon, the "then" keyword, a list of commands on separate lines, another semi-colon, and finally the "fi" keyword.

We will discuss more about the details of the syntax as we apply conditionals to files, numbers, and strings. Let's look at how to apply a conditional to a file.

Shell Scripting Using File Conditionals

You can apply a conditional on a file to check the file mode (readable, writable, executable), the type of the file, to compare files sizes, and so forth.

You will often see this when a script is looking for a file to modify or when trying to read a configuration file. A regular file means not a device (block or character), symlink, socket, or named pipe (for example `fifo`). There are tests for those types as well.

The table shows some common options for file conditionals. For a larger list of file conditionals, see page 408 and 436 in *UNIX In A Nutshell*, or the `help test`, `man test`, or `info test` commands.

Next, we are going to look at applying a conditional to a number.

Shell Scripting Using Number Conditionals

You can apply a conditional on numbers to compare values. You can also use them to perform math functions; however, this is not the most efficient use of number conditionals. Conditionals applied to numbers are most often seen in scripts to check the size of files such as log files. They are also frequently used to verify disk space usage limits for users.

The table shows some common options for number conditionals. More information is available in your reading materials. Finally, let's look at applying a conditional to a string.

Shell Scripting Using String Conditionals

String conditionals greatly differ from number conditionals as you can see on the attached table. Furthermore, like any other shell command, shell expansion including words splitting and pathname expansion occur whenever the test command is used (even with the square bracket syntax). That default shell behavior can lead to unexpected results whenever we are trying to compare strings as is. This is why strings comparison is usually done with quoted strings.

There is an alternative syntax for comparing values that uses double square brackets and prevents words splitting and pathname expansion, that syntax is discussed in more detail in the provided Supplemental Material.

Next, we are going to explore how to get shell commands to repeat automatically, which is a powerful way to expand scripts.

Shell Scripting Using Loops

Loops execute a set of commands repeatedly. There are several different types of loops available. Which you use depends on the situation. Click each tab to learn about the different types of loops. When you have finished reviewing each set, click the Next Slide button to continue with the module.

The **while loop** executes a set of commands repeatedly as long as an expression or condition is true. Use it when you need to manipulate the value of a variable repeatedly. This loop displays the numbers zero though nine. Each time this loop executes, the variable "a" is checked to see whether it has a value that is less than 10. If the value of "a" is less than 10, this test condition has an exit status of 0. In this case, the current value of "a" is displayed and then a is incremented by 1. Click another tab to continue learning about loops. If you have finished common shell features, click the Next Slide button to continue with the module.

The **for loop** operates on a list of items or a list of words in a string. It repeats a set of commands for every item in that list. This loop displays all the files starting with `.bash` and available in your home directory. Click each tab to learn about the different types of loops. When you have finished reviewing each set, click the Next Slide button to continue with the module.

The **until loop** executes a set of commands until an expression or condition is true. This loop displays the numbers zero to nine. Click each tab to learn about the different types of loops. When you have finished reviewing each set, click the Next Slide button to continue with the module.

Arithmetic Operations in BASH

Another powerful way to create shell scripting is by using the `expr` tool to perform basic integer operations like addition, subtraction, division, multiplication, and modulo. The `expr` tool is so powerful because it can perform all the basic arithmetic operations, and it can perform all the comparison operations available with the `test` program (and so much more). The output value of the `expr` program can be captured using command substitution.

In this module, we are showing only the arithmetic operations, but there are many more features of the `expr` tool. Use the `man` pages to find out more.

Video

Welcome to Analyzing a Script. I will be your guide for this video presentation.

Having a basic understanding of simple scripting languages and how scripts work is a very useful skill for a computer operator to have.

In this presentation, we will walk through a bash script, highlighting some of the various code snippets and what they do. Let's get started!

When executed, this bash script will create a simple web server that can receive input and respond with the appropriate output depending on what was received as input. Let's go over some of the individual sections of this code.

At the beginning of a script, it is common to set some variables that will be used throughout the rest of the script code. On line 2 we see a variable named `base` being set to equal the value of

`/var/www`. Line 3 sets a variable named `default_port` to equal `8080`. So anytime in the code you see $base (such as line 14), it will translate to "var/www" and `$default_port` will equal 8080 (as seen on line 38). The beginning of the script is not the only place a variable can be set. They can be set throughout the script and seen here by lines 12 and 13 among others.

After the initial variables are set, we move down to line 4 which starts a function that continues down to line 34. A function is used to group pieces of code in a logical way. To create a function just write `function` followed by whatever you want to name that function (in this case "serve") and then anything in between the function brackets (lines 4 and 34) will be considered a part of that same group (or function). Anything in between those 2 lines support a common action. We can see that in this function there are several items such as read, a while loop and a conditional if statement. All supporting the same logical action. Let's look at what this function is doing.

Within the function `serve`, the first line of code is a `read` followed by the variable name `request`. The `read` command reads one line of data from standard input. In this case, the `read` command will store the contents of that one line of data in a variable named `request`. Next, on lines 7 through 10 we see a while loop. A while loop is designed to repeat an execution of some command as long as the same condition remains true. So in this case, the while loop is going to read input from standard in and store that data in a variable named `header` until a carriage return (\r) is received. Once the carriage return is received, the code moves to the next action, which is setting up some additional variables to be used within the rest of the function.

Lines 12 through 14 are setting several variables and also performing some string substitutions. The data that was being read in from the previous while loop was expected to be an HTTP request, so these variables are basically cleaning up the expected HTTP request so that it can be used later on. Specifically, line 12 sets a variable called `url`. The data for this variable is the contents of the $request variable from line 5. The pound sign (#) is a string modifier that tells bash to remove the first match of whatever follows (in this case GET) from the beginning of a string. If you are familiar with the hypertext transfer protocol, you know that `GET` is a standard method to send a request to a web server. This script is stripping that `GET` off of the data that will be stored in the variable `url`. The next line (13) is taking the data that now resides in the variable `$url` and is performing a string substitution (the percent sign) to remove a match from the end of the string. In this case, it is removing the first instance of HTTP slash anything (the asterisk being a wild card meaning anything else). After these 2 commands are executed, the variable $url will contain the data from the variable $request, minus GET and HTTP slash. Line 14 creates another variable. This one named "filename". The contents of that variable will be the contents of the variable $base (which is /var/www) and the contents of the variable $url, which we just covered.

The next part of the code (lines 16 - 31) is an if/else statement. This is a basic conditional statement that can be translated as "if this occurs (in this case the variable filename exists) do something (in this case, send an HTTP 200 response) otherwise (or "else" in this context) do this instead (the "else" action in this statement would be to send an HTTP 404 response)." So, this part of the program is evaluating the received data to determine if it was a proper HTTP request or not. Most of the lines in this conditional are self-explanatory. Line 19 returns the content-length, which it derives from performing an ls command which list information about files or directories. In this case, content-length is the value of the 5th field of the `ls -l` response. The fifth field is the file size.

After the if/else conditional finishes running, the code steps down to lines 32 through 34. Line 32

sends an end of file (ascii 004) to standard out. Line 33 gracefully exits from the function and line 34 denotes the end of commands inside the function "serve."

The next part of the code being read-in is lines 36 and 37. Line 36 is removing any instance of a file named `www_fifo`. Line 37 is creating a special file named `www_fifo`. This is created using the command `mkfifo`. `Mkfifo` creates a special file that can be opened by multiple processes for reading and writing.

The next set of commands in our script on lines 38 through 40 is setting a variable for a port. Remember that currently our default_port value equals 8080. Line 38 sets the initial value of the variable `port` to the value found in the variable `default_port` (8080). The next line, line 39 states that if the command line arguments when starting this script consists of more than 1 argument, then take the second argument as the new value for the variable "port". There are a couple special bash variables being introduced here. Dollar sign pound ($#) is a variable that notes the number of arguments passed. So, on line 39, dollar sign pound will return true if the number of arguments being passed on the command line is greater than 1. Right after that evaluation, we have two ampersands and `port=$2`. The dollar sign two ($2) is our next special variable. Arguments passed from the command line are expressed in bash scripting as variables starting at $0 (which is the name of the script itself) and incrementing by one per argument. So, `$1` is the first argument after the script name, `$2` is the second argument after the script name and so on. So line 39 can be read as "if the command line arguments are greater than 1, use the second argument as the new value for the port variable, if not, then the port variable stays set at 8080."

The final 3 lines of the script are a while loop, which we discussed earlier. This while loop prints the contents of the file `www_fifo` to standard out using the cat command, the output of that command is piped to an instance of netcat which is listening on the value of the variable `$port`. The netcat session is piped to the function serve which will received the data from netcat and redirect it to the www_fifo file. In essence, the script is a web server designed to read in HTTP GET requests and output appropriate HTTP responses.

We have walked through a bash script showing how the various code snippets perform their given tasks, and explained some commonly used commands such as variables, functions, conditionals, loops and pipes. You now have the tools to analyze a script.

Exercise Introduction

It is time for an Exercise. In order to successfully complete the Exercise, you are expected to use your notes and any available resources presented throughout the course, in addition to conducting your own Internet research.

Shells and Scripting: Section 5 Transcript

Summary

You have completed the Shells and Shell Scripting module. During this module we explored shells and shell scripting. You discovered the power and flexibility of shells and why it is important that an operator is knowledgeable of and capable with using UNIX shells. You should now be able to:

- Explain the purpose of a shell in a UNIX operating system,
- Identify commonly used shells and their unique core characteristics,
- Identify features and functions of the shell environment,
- Recognize the use of patterns in writing commands for different shells, and
- Demonstrate basic familiarity with shell script programming.

To receive credit and advance to the next module, you must achieve a passing score on the Module Exam. Click the Next Section button to begin the Module Exam.