University of Technology of Mauritius

MSc Software Engineering 2012

Student ID : 120463

Module: Unix Programming

Assignment on different Unix shells.

Describe what a shell is.

A Unix shell is a command-line interpreter or shell that provides a traditional user interface for the Unix operating system and for Unix-like systems. Users direct the operation of the computer by entering commands as text for a command line interpreter to execute or by creating text scripts of one or more such commands.

The most influential Unix shells have been the Bourne shell and the C shell. The Bourne shell, sh, was written by Stephen Bourne at AT&T as the original Unix command line interpreter; it introduced the basic features common to all the Unix shells, including piping, here documents, command substitution, variables, control structures for condition-testing and looping and filename wildcarding. The language, including the use of a reversed keyword to mark the end of a block, was influenced by ALGOL 68.[1]

The C shell, csh, was written by Bill Joy while a graduate student at University of California, Berkeley. The language, including the control structures and the expression grammar, was modeled on C. The C shell also introduced a large number of features for interactive work, including the history and editing mechanisms, aliases, directory stacks, tilde notation, cdpath, job control and path hashing.

Users typically interact with a modern Unix shell using a terminal emulator. Common terminals include xterm and GNOME Terminal.

Describe what a shell does

A shell hides the details of the underlying operating system with the shell interface and manages the technical details of the operating system kernel interface, which is the lowest-level, or 'inner-most' component of most operating systems.

In Unix operating systems users typically have many choices of command-line interpreters for interactive sessions. When a user logs in to the system, a shell program is automatically executed. It is both an interactive command language as well as a scripting programming language, and is used by the operating system as the facility to control (shell script) the execution of the system.

Describe how a shell relates to the overall system.

Each user has a default shell, which will be launched when a command prompt is opened. The default shell is specified in the configuration file /etc/passwd in the last field of the line corresponding to the user. The shell is initialized by reading its overall configuration (in a file of the directory /etc/), followed by reading the user's own configuration (in a hidden file the name of which starts with a dot, located in the basic user directory, i.e. /home/user_name/.configuration_file). Then, a command prompt window or prompt is displayed as follows:

machine:/directory/current.

By default, for most shells, the prompt consists of the name of the machine, followed by a colon (:), the current directory, then a character indicating the type of user connected:

- "$" specifies a normal user
- "#" specifies the administrator, called "root"

Store data in variable

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix Shell variables would have their names in UPPERCASE.

The following examples are valid variable names:

_FIRSTNAME

LASTNAME_1

VAR_1

Following are the examples of invalid variable names:

- 2_VAR
- -FIRSTNAME
- VAR1-VAR2
- VAR!

The reason you cannot use other characters such as !,*, or - is that these characters have a special meaning for the shell.

Storing data from file into a variable in bash shell

#!/bin/bash

value=`cat sources.xml`

echo $value

other examples

Color=red

echo $color

Read-only Variables

The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed. For example, following script would give error while trying to change the value of NAME:

#!/bin/sh

NAME=”username”

readonly NAME

NAME=”password”

This would produce the following result:

/bin/sh: NAME: This variable is read only.

Variable Types:

When a shell is running, three main types of variables are present:

1. Local Variables: A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.
2. Environment Variables: An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.
3. Shell Variables: A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Environment Variables

The Bourne shell predefines a set of global variables known as environment variables. They define default values for the current account's home directory, the strings to use as the primary and secondary prompts, a list of directories to search through when attempting to find an executable, the name of the account, and the current shell, to name a few. The values for these variables are generally set at login when the shell reads the .login script. This script resides within the account's home directory. Environment variables are by no means set in stone. They may be changed from session to session, but generally, they remain constant and keep a user's account configured for the system.

Some environment variables:

$PATH

Contains a colon-separated list of directories that the shell searches for commands that do not contain a slash in their name (commands with slashes are interpreted as file names to execute, and the shell attempts to execute the files directly). It is equivalent to the Windows %PATH% variable. See: Path (computing).

$HOME

Contains the location of the user's home directory. Although the current user's home directory can also be found out through the C functions getpwuid and getuid, $HOME is often used for convenience in various shell scripts (and other contexts). Using the environment variable also gives the user the possibility to point to another directory.

$PWD

This variable points to the current directory. Equivalent to the output of the command pwd when called without arguments.

$DISPLAY

Contains the identifier for the display that X11 programs should use by default.

$LD_LIBRARY_PATH

On many Unix systems with a dynamic linker, contains a colon-separated list of directories that the dynamic linker should search for shared objects when building a process image after exec, before searching in any other directories.

$LANG, $LC_ALL, $LC_...

LANG is used to set to the default locale. For example, if the locale values are pt_BR, then the language is set to (Brazilian) Portuguese and Brazilian practice is used where relevant. Different aspects of localization are controlled by individual LC_-variables (LC_CTYPE, LC_COLLATE, LC_DATE etc.). LC_ALL can be used to force the same locale for all aspects.

Customizing the environment

You can use the :set command to set environment variables, and the :map command to map a key sequence to a vi command key. Environment variables are set either by assigning them as option or no option for Boolean variables, or by assigning them as option=value.

To display the line numbers of your example file enter the following command:

:set number Return 2 222222 7 777777

To remove the line numbers, enter the following command:

:set nonumber Return 2 222222 7 777777

The :map command sets a single vi command key to a vi command sequence. The syntax for the :map command follows:

:map key sequence Return 2 222222 7 777777

Identify specific features of each shell

The operating system provides the following shells that have both command execution and programming capabilities:

• The Bourne shell (sh)

This is a simple shell that is easily used in programming. It is usually represented by a dollar sign ($) prompt. This shell does not provide either the interactive features or the complex programming constructs(arrays and integer arithmetic) of the C shell or the Korn shell.

The Bourne shell also provides a restricted shell (Rsh).

• The C shell (csh)

This shell is designed for interactive use. It is usually represented by a percent sign (%) system prompt. The C shell provides some features for entering commands interactively:

– A command history buffer

– Command aliases

– Filename completion

– Command line editing

The Korn shell (ksh)

This shell combines the ease of use of the C shell and the ease of programming of the Bourne shell. The system prompt is usually a dollar sign ($) prompt. The Korn shell provides these features:

– The interactive features of the C shell

– The simple programming syntax of the Bourne shell

– Command line editing

– The fastest execution time

– Upward compatibility with the Bourne shell (that is, most Bourne shell programs will run under the Korn shell)

Bourne Shell

The Bourne shell (sh) was the default Unix shell of Unix Version 7. Most Unix-like systems continue to have /bin/sh—which will be the Bourne shell, or a symbolic link or hard link to a compatible shell—even when other shells are used by most users.

Developed by Stephen Bourne at Bell Labs, it was a replacement for the Thompson shell, whose executable file had the same name—sh. It was released in 1977 in the Version 7 Unix release distributed to colleges and universities. Although it is used as an interactive command interpreter, it was always intended as a scripting language and contains all the features that are commonly considered to produce structured programs.

The Bourne shell was once standard on all branded Unix systems, although historically BSD-based systems had many scripts written in csh. Bourne shell scripts can typically be run with bash or dash on GNU/Linux or other Unix-like systems. On many Linux systems, /bin/sh is a symbolic or hard link to Bash. However, for efficiency, some Linux systems (such as Ubuntu) link /bin/sh to dash instead.

Along with the Korn shell and the C shell, the Bourne shell remains among the three most widely used and is included with all UNIX systems. The Bourne shell is often considered the best shell for developing scripts.

Command-line arguments

The most useful of these variables are the ones referring to the command-line arguments. $1 refers to the first command-line argument (after the name of the script), $2 refers to the second one, and so forth, up to $9.

If you have more than nine command-line arguments, you can use the shift command: this discards the first command-line argument, and bumps the remaining ones up by one position: $2 becomes $1, $8 becomes $7, and so forth.

The variable $0 (zero) contains the name of the script (argv[0] in C programs).

Often, it is useful to just list all of the command-line arguments. For this, sh provides the variables $* (star) and $@ (at). Each of these expands to a string containing all of the command-line arguments, as if you had used $1 $2 $3...

The difference between $* and $@ lies in the way they behave when they occur inside double quotes: $* behaves in the normal way, whereas $@ creates a separate double-quoted string for each command-line argument. That is, "$*" behaves as if you had written "$1 $2 $3", whereas "$@" behaves as if you had written "$1" "$2" "$3".

Finally, $# contains the number of command-line arguments that were given.

Other special variables

$? gives the exit status of the last command that was executed. This should be zero if the command exited normally.

$- lists all of the options with which sh was invoked. See sh(1) for details.

$$ holds the PID of the current process.

$! holds the PID of the last command that was executed in the background.

$IFS (Input Field Separator) determines how sh splits strings into words.

Quasi-variable constructs

The ${VAR} construct is actually a special case of a more general class of constructs:

${VAR:-expression}

Use default value: if VAR is set and non-null, expands to $VAR. Otherwise, expands to expression.

${VAR:=expression}

Set default value: if VAR is set and non-null, expands to $VAR. Otherwise, sets VAR to expression and expands to expression.

${VAR:?[expression]}

If VAR is set and non-null, expands to $VAR. Otherwise, prints expression to standard error and exits with a non-zero exit status.

${VAR:+expression}

If VAR is set and non-null, expands to the empty string. Otherwise, expands to expression.

${#VAR}

Expands to the length of $VAR.

The above patterns test whether VAR is set and non-null. Without the colon, they only test whether VAR is set.

The Korn shell

The Korn shell, or ksh, was invented by David Korn of AT&T Bell Laboratories in the mid-1980s. It is almost entirely upwardly compatible with the Bourne shell, [1] which means that Bourne shell users can use it right away, and all system utilities that use the Bourne shell can use the Korn shell instead. In fact, some systems have the Korn shell installed as if it were the Bourne shell.

Although the Bourne shell is still known as the "standard" shell, the Korn shell is becoming increasingly popular and is destined to replace it. In addition to its Bourne shell compatibility, it includes the best features of the C shell as well as several advantages of its own. It also runs more efficiently than any previous shell.

The Korn shell's command-line editing modes are the features that tend to attract people to it at first.

With command-line editing, it's much easier to go back and fix mistakes than it is with the C shell's history mechanism-and the Bourne shell doesn't let you do this at all.

The other major Korn shell feature that is intended mostly for interactive users is job control. The rest of the Korn shell's important advantages are mainly meant for shell customizers and programmers. It has many new options and variables for customization, and its programming features have been significantly expanded to include function definition, more control structures, built-in regular expressions and integer arithmetic, advanced I/O control, and more.

The Korn shell is a backward-compatible extension of the Bourne shell. Features that are valid only in the Korn shell are so indicated.

- Command-line editing (using vi or emacs).
- Access to previous commands (command history).
- Integer arithmetic.
- More ways to match patterns and substitute variables.
- Arrays and arithmetic expressions.
- Command name abbreviation (aliasing).

Variables is used by the Korn shell to store values. Setting variables within a script or at a shell prompt is pretty simple. Define a variable called NAME and SURNAME as follows:

CAR=”bmw”

COLOR=”black”

To display variable, use:

echo $CAR

echo $COLOR

Restricted Shells

Restricted shells can also be set up by supplying rksh and rsh in the shell field of /etc/passwd or by using them as the value for the SHELL variable.

Restricted shells act the same as their non-restricted counterparts, except that the following are prohibited:

- Changing directory (i.e., using cd).
- Setting the PATH variable.rksh also prohibits setting ENV and SHELL.
- Specifying a /for command names or pathnames.
- Redirecting output (i.e., using>and>>).

Shell scripts can still be run, since in that case the restricted shell will call ksh or sh to run the script.

The C shell

C shell is the UNIX Shell(command execution program, often called a command interpreter) created by Bill Joy at the University of California at Berkeley as an alternative to UNIX's original shell, the Bourne shell. These two UNIX shells, along with the Korn shell, are the three most commonly used shells.

The C shell program name is csh, and the shell prompt (the character displayed to indicate readiness for user input) is the % symbol. The C shell was invented for programmers who prefer syntax similar to that of the C programming language

The other popular member of the C shell family is called tcsh (for Tab C shell) and is an extended version of C shell. Some of tcsh's added features are: enhanced history substitution (which allows you to reuse commands you have already typed), spelling correction, and word completion (which allows you to type the first couple of letters in a word and hit the tab key to have the program complete it).

The C shell is designed for interactive use. Below are some features for entering command interactively

- A command history buffer
- Command aliases
- Filename completion
- Command line editing

The basic command for declaring a C shell variable is the set command. For example,

set car = "toyota"

This will initialize the string variable name to contain the value "Kazuya" The method for initializing a wordlist variable is slightly different. The set command is used, but the list of values is enclosed in parentheses, as in

set vehicles = (nissan Honda Mercedes Renault Mitsubishi)

For displaying the value in the variable we just have to put a $ sign in from of the variable for example echo $car will display Toyota on the screen.

Sample shell scripting

```
#!/bin/sh
myname=`whoami`

if [ $myname = root ]; then
        echo "Welcome to FooSoft 3.0"
else
        echo "You must be root to run this script"
        exit 1
fi
```

Differences between Bourne and C Shell Quoting

As in the Bourne shell, the overall idea of C shell quoting is: quoting turns off (disables) the special meaning of characters. There are three quoting characters: a single quote ('), a double quote ("), and a backslash (\ ).

Quoting Character Explanation

' xxx ' Disable all special characters in xxx except ! .

" xxx " Disable all special characters in xxx except $,`, and ! .

\ x Disable special meaning of character x . At end of line, a \ treats the newline character like a space (continues line).

In the Bourne shell, single and double quotes include newline characters. Once you open a single or double quote, you can type multiple lines before the closing quote.

Both the C and the Korn shells offer the following interactive features:

• Command history

The command history buffer stores the commands you enter and allows you to display them at any time. As a result, you can select a previous command, or parts of previous commands, and then re-execute them. This feature may save you time because it allows you to reuse long commands instead of retyping them. In the C shell, this feature requires some setup in the .cshrc file; in the Korn shell this feature is automatically provided.

•Command aliases

The command aliases feature allows you to abbreviate long command lines or rename commands. You do this by creating aliases for long command lines that you frequently use. For example, assume that you often need to move to the directory /usr/chang/reports/status.

You could create an alias status that could move you to that directory whenever you enter status on the command line. In addition, aliases allow you to make up more descriptive names for operating system commands. For example, you could define an alias named rename for the mv command.

•Filename completion

In the C shell, the filename completion feature saves typing by allowing you to enter a portion of the filename. When you press the Escape key, the shell will complete the filename for you.

C, Bourne, and Korn Shell Features

Feature	Description	C	Bourne	Korn
Shell programming	A programming language that includes features such as loops, condition statements, and variables.	Yes	Yes	Yes
Signal trapping	Mechanisms for trapping interruptions and other signals sent by the operating system.	Yes	Yes	Yes
Restricted shells	A security feature that provides a controlled shell environment with limited features.	No	Yes	No
Command aliases	A feature that allows you to abbreviate long command lines or to rename commands.	Yes	No	Yes
Command history	A feature that stores commands and allows you to edit and reuse them.	Yes	No	Yes
Filename completion	A feature that allows you to enter a portion of a filename and the system automatically completes it or suggests a list of possible choices.	Yes	No	Yes
Command line editing	A feature that allows you to edit a current or previously entered command line.	Yes	No	Yes
Array	The ability to group data and call it by a name.	Yes	No	Yes
Integer arithmetic	The ability to perform arithmetic functions within the shell.	Yes	No	Yes

Job control	Facilities for monitoring and accessing background processes.	Yes	No	Yes