Korn shell scripting

A beginner’s guide

Kurt A. Riley
IT Consultant

17 June 2008

Korn shell scripting is something all UNIX® users should learn how to use. Shell scripting provides you with the ability to automate many tasks and can save you a great deal of time. It may seem daunting at first, but with the right instruction you can become highly skilled in it. This article will teach you to write your own Korn shells scripts.

What is a shell?

The IBM® AIX® operating system and other UNIX-like operating systems need a way to communicate with the kernel. This is done is through the use of a shell. There are a few different shells that you can use, but this article focuses on the Korn shell. The Korn shell is the default shell used with AIX.

When you log into AIX, you are started at a prompt in a certain directory. The default directory is usually your home directory. The reason it's called a home directory is because the directory structure is usually something like this:

```
$/home/jthomas:
```

When you log in, you are said to be at the command line or command prompt. This is where you enter UNIX commands. You enter shell commands that interact with the UNIX kernel. These commands can be as simple as one line to check the date or multiple lines long, depending on what you're doing. Listing 1 provides some sample commands.

Listing 1. Sample commands

```
$date
Fri May  1 22:59:28 EDT 2008
$uptime
10:59PM   up 259 days,   9:44,   5 users,   load average: 3.81, 14.27, 13.71
$hostname
gonzo
```

The great thing about shell commands is that you can combine them in a file called a script that allows you to run multiple commands one after another. This is great when you have to repeat the same commands over and over again. Instead of repeatedly typing these commands, you can put them in a Korn shell script.

Writing your first Korn shell script

The first line in a Korn shell script is the shell itself. It is denoted as follows:

```
#!/bin/ksh
```

To write Korn shell scripts on AIX, you need to use a text editor. The most widely used and most readily available is vi. It can be a little intimidating at first, but the more you use vi, the more comfortable you will become. People have written whole books just on how to use the vi text editor.

To begin writing your first Korn shell script, you need to open the vi editor and add the shell name as the first line. After that, you need to build some type of script header telling users who wrote the script, what the script does, and when it was written. You can name your script anything you want, but you usually use the extension .ksh to refer to a Korn shell script. You do not have to do this, but it's good practice. The pound symbol (#) is used to comment with scripts, as shown in Listing 2.

Listing 2. Example of a script header

```
$vi my_first_script.ksh
#!/bin/ksh
###################################################
# Written By: Jason Thomas
# Purpose: This script was written to show users how to develop their first script
# May 1, 2008
###################################################
```

This script header is pretty basic, but it does the trick.

Variables

Setting variables within a script is fairly simple. I usually capitalize all variables within my scripts, as shown in Listing 3, but you don’t have to.

Listing 3. Example of variables

```
#Define Variables
HOME="/home/jthomas" #Simple home directory
DATE=$(date) # Set DATE equal to the output of running the shell command date
HOSTNAME=$(hostname) # Set HOSTNAME equal to the output of the hostname command
PASSWORD_FILE="/etc/passwd" # Set AIX password file path
```

Korn shell nuts and bolts

So far, you’ve learned how to start writing a Korn shell script by writing a basic script header and defining some variables. Now it's time to start writing some Korn shell code.

Start by reading some lines from a file. In this case, use the /etc/passwd file you already defined in your script, and print just the usernames, as shown in Listing 4.

Listing 4. The for loop

```
$vi my_first_script.ksh
#!/bin/ksh
###################################################
# Written By: Jason Thomas
# Purpose: This script was written to show users how to develop their first script
# May 1, 2008
###################################################

#Define Variables
HOME="/home/jthomas" #Simple home directory
DATE=$(date) # Set DATE equal to the output of running the shell command date
HOSTNAME=$(hostname) # Set HOSTNAME equal to the output of the hostname command
PASSWORD_FILE="/etc/passwd" # Set AIX password file path

#Begin Code

for username in $(cat $PASSWORD_FILE | cut -f1 -d:)
do

       print $username

done
```

This syntax is called a *for loop*. It allows you to open the /etc/passwd file and read each line one at a time, cutting out just the first field in the file and then printing that line. Notice this special character: `|`. This is called a pipe. Pipes allow you to redirect output from one command into another.

After you save the script, you run it as shown in Listing 5.

Listing 5. Running the script

```
$./my_first_script.ksh
root
daemon
bin
sys
adm
uucp
nobody
lpd
```

The script begins to print the output to the screen. Alternatively, you can just print the output to a file by doing the following:

```
print $username >> /tmp/usernames
```

The `>>` tells the print command to append each username one after another to a file only. By doing this, you never see the text display on your terminal. You can also print the output to both the screen and a file by using this command:

```
print $username | tee –a /tmp/usernames
```

The `tee` command allows you to redirect output to your terminal and to a file at the exact same time.

You just learned how to read from a file with a for loop and how to cut out just a username and redirect the output to a file or to your terminal.

Error checking

What happens if the /etc/passwd file didn't exist to begin with? The short answer is the script would fail. Listing 6 shows the syntax that checks to see if the files exist.

Listing 6. Syntax for checking to see if a file exists

```
#Begin Code
if [[ -e $PASSWORD_FILE ]]; then #Check to see if the file exists and if so then continue

     for username in $(cat $PASSWORD_FILE | cut -f1 -d:)
     do

         print $username

     done
else

         print "$PASSWORD_FILE was not found"
         exit
fi
```

This little piece of code shows the conditional `if` statement. If the /etc/passwd file exists, then the script continues. If the file doesn't exist, then the script prints `"/etc/passwd file was not found"` to the terminal screen and then exits. Conditional `if` statements start with `if` and end with the letters reversed (`fi`).

The dollar question mark ($?)

Every time you run a command in AIX, the system sets a variable that is often referred to as dollar question. AIX sets this to either 0 for successful or non-zero for failure. This is excellent for Korn shell scripting. Listing 7 shows how the `$?` is set when you run valid and invalid AIX commands.

Listing 7. How the $? is set for valid and invalid AIX commands

```
$date
Sat May 10 00:02:31 EDT 2008
$echo $?
0
$uptime
  12:02AM   up 259 days,  10:47,  5 users,  load average: 4.71, 10.44, 12.62
$echo $?
0
$IBM
ksh: IBM:  not found.
$echo $?
127
$aix
ksh: aix:  not found.
$echo $?
127
$ls -l /etc/password
ls: 0653-341 The file /etc/password does not exist.
$echo $?
2
```

This is helpful when writing Korn shell scripts because it gives you another way to check for errors. Here's a different way to see if the /etc/passwd file exists:

```
#Begin Code
PASSWORD_FILE="/etc/passwd"

ls –l $PASSWORD_FILE > /dev/null 2>&1
```

This command allows you to list the file. However, you don’t really care if the file is there or not. The important thing is for you to get the return code of the command. The greater than sign (`>` allows you to redirect output from the command. You will learn more about redirecting output later in this article.

Listing 8 shows how to use `$?` in a script.

Listing 8. Using `$?` in a script

```
#Begin Code
PASSWORD_FILE="/etc/passwd"

ls –l $PASSWORD_FILE > /dev/null 2>&1
if [[ $? != 0 ]]; then

       print “$PASSWORD_FILE was not found"
       exit

else

   for username in $(cat $PASSWORD_FILE | cut -f1 -d:)
   do

       print $username

   done
fi
```

Instead of actually checking to see if the file exists, I tried to list the file. If you can list the file, then the file exists. If you can't list it, then it doesn't exist. You list a file in AIX by using the `ls –l` *`filename`* command. This gives you a way to test to see if your AIX command was successful by checking the `$?` variable.

Standard in, out, and error

You really need to understand these. You basically have three sources of input and output. In AIX, they are referred to as `STDIN`, `STDOUT`, and `STDERR`. `STDIN` refers to the input you might get from a keyboard. `STDOUT` is the output that prints to the screen when a command works. `STDERR` prints to the screen when a command fails. The `STDIN`, `STDOUT`, and `STDERR` file descriptors map to the numbers 0, 1, and 2, respectively.

If you want to check to see if a command was a success or a failure, you do something like Listing 9.

Listing 9. Redirecting output to `STDOUT` and `STDERR`

```
$date > dev/null 2>&1  # Any output from this command should never be seen

if [[ $? = 0 ]]; then
      print "The date command was successful"
else
      print "The date command failed
fi
```

This code runs the AIX date command. You should never see any output from `STDOUT` (file descriptor 1) or `STDERR` (file descriptor 2). Then you use the conditional `if` statement to check the return code of the command. As you learned previously, if the command returns a zero, then it was successful; if it returns a non-zero, then it failed.

Functions

In Korn shell scripting, the word function is a reserved word. Functions are a way to divide the script into segments. You only run these segments when you call the function. Create an error-check function based on the code you've already written, as shown in Listing 10.

Listing 10. Error-check function

```
##################
function if_error
##################
{
if [[ $? -ne 0 ]]; then # check return code passed to function
    print "$1" # if rc > 0 then print error msg and quit
exit $?
fi
}
```

If I want to run a simple command from inside a script, I can easy write some code similar to the error checking of the `$?` above. I can also just call the `if_error` function every time I want to check to see if something failed, as shown in Listing 11.

Listing 11. Calling the `if_error` function

```
rm –rf /tmp/file #Delete file
if_error "Error: Failed removing file /tmp/file"

mkdir /tmp/test #Create the directory test
if_error "Error: Failed trying to create directory /tmp/test"
```

Each time one of the above commands is run, a call is made to the `if_error` function. The message you want to display for that particular error check is passed to the `if_error` function. This is great because it allows you to write the shell script code one time, but you get to leverage it over and over. This makes writing shell scripts quicker and easier.

The `case` statement

The `case` statement is another conditional statement that you can use in place of using an `if` statement. The `case` statement begins with the word `case` and ends with the reverse (`esac`). A `case` statement allows you to quickly build upon it as your script evolves and needs to perform different tasks. Listing 12 provides an example.

Listing 12. The `case` statement

```
case value in
"Mypattern") commands to execute
 when value matches
 Mypattern
 ;;
esac
```

So, say that you want to delete a file at different times of the day. You can create a variable that checks to see what time it is:

```
TIME=$(date +%H%M)
```

The code shown in Listing 13 will delete a file at 10:00 p.m. and 11:00 p.m. So, each time this section of code is executed, the `$TIME` is checked to see if it matches the times of the `case` statement. If so, then the code is executed.

Listing 13. `case` statement to check time

```
case $TIME in
                 "2200") #This means 10:00
                  rm –rf /tmp/file1
                        ;;
                  "2300")#This means 11:00
                  rm –rf /tmp/file1
                        ;;
                    "*")
                        echo "Do nothing" > /dev/null
                        ;;

 esac
```

Putting a whole script together

So far, you've created a script header and some simple variables and added a function, as shown in Listing 14.

Listing 14. Example Korn shell script

```
$vi my_second_script.ksh
#!/bin/ksh
###################################################
# Written By: Jason Thomas
# Purpose: This script was written to show users how to develop their first script
# May 1, 2008
###################################################

#Define Variables
HOME="/home/jthomas" #Simple home directory
TIME=$(date +%H%M) # Set DATE equal to the output of running the shell command date
HOSTNAME=$(hostname) # Set HOSTNAME equal to the output of the hostname command

```

```
##################
function if_error
##################
{
if [[ $? -ne 0 ]]; then # check return code passed to function
    print "$1" # if rc > 0 then print error msg and quit
exit $?
fi
}

if [[ -e /tmp/file ]]; then  #Check to see if the file exists first
   rm –rf /tmp/file #Delete file
   if_error "Error: Failed removing file /tmp/file"
else
   print "/tmp/file doesn’t exist"
fi

if [[ -e /tmp/test ]]; then
     mkdir /tmp/test #Create the directory test
     if_error "Error: Failed trying to create directory /tmp/test"
else
     print "Directory exists, no need to create directory"
fi

case $TIME in
                 "2200")
                  rm –rf /tmp/file1
                        ;;
                  "2300")
                  rm –rf /tmp/file1
                        ;;
#End Script
esac
```

To run the script, you simply type `./scriptname.ksh`, like this:

```
$./my_second_script.ksh
```

Feeding input into a script from the command line

You can create scripts that allow you to feed input into the script. Look at Listing 15.

Listing 15. Feeding input into a script

```
#!/bin/ksh

OPTION=$1

print "I love $OPTION"

$./scriptname milk
I love milk
$./scriptname tea
I love tea
$./scriptname "peanut butter"
I love peanut butter
```

Any time you feed something into a script, the first option after the script name is called `$1`. The second option after the script name is called `$2`, and so on. This is a great way to write a script so that it is more like a UNIX command with switches or options.

E-mail from a script

You can use your scripts to generate some type of report. For instance, maybe a script was written to keep track of new users added to the system daily. This script could write the output to a file, and then you could send it to yourself. This way, you could get a copy of all the new users added to the system every day. To do this, run the following command:

```
$REPORT="/tmp/users"

cat $REPORT | mailx –s "User admin report from server XYZ" Jason_Thomas@kitzune
```

This will send you an e-mail of the contents in the $REPORT file. The `-s` is what the subject of the e-mail will be. This can come in really handy.

Conclusion

Korn shell scripting can save you a lot of time and make your job so much easier. It can seem intimidating at first, but remember to always start out simple and build upon each and every script. Always follow the same steps: build your script header, define your variables, and error check your work. You just might find yourself trying to write a script for everything you do.

RELATED TOPICS: UNIX and Linux forums Linux Journal Kornshell

About the author

Kurt A. Riley

Kurt Riley has been working with UNIX and Linux systems since 2000. He's provided consulting services to a number of Fortune 500 companies. Kurt holds a bachelor's degree in Information Technology and currently works at Bank of America as a security architect.