9. Basic scripting

- **Turning Commands into a Script**
    - Weight: 4
    - Description: Turning repetitive commands into simple scripts.
    - Key Knowledge Areas:
        - Basic text editing
        - Basic shell scripting
    - The following is a partial list of the used files, terms, and utilities:
        - /bin/sh
        - Variables
        - Arguments
        - for loops
        - echo
        - Exit status
    - Things that are nice to know:
        - pico, nano, vi (only basics for creating scripts)
        - Bash
        - if, while, case statements
        - read and test, and [commands

9.3 Shell Scripts in a Nutshell

A *shell script* is a file of executable commands that has been stored in a text file. When the file is run, each command is executed, Shell scripts have access to all the commands of the shell, including logic. A script can therefore test for the presence of a file or look for particular output and change its behavior accordingly. You can build scripts to automate repetitive parts of your work, which frees your time and ensures consistency each time you use the script. For instance, if you run the same five commands every day, you can turn them into a shell script which reduces your work to one command.

A script can be as simple as one command:

```
echo “Hello, World!”
```

The script, `test.sh`, consists of just one line that prints the string “Hello, World!” to the console.

Running a script can be done either by passing it as an argument to your shell or by running it directly:

```
bob:tmp $ sh test.sh
Hello, World!
bob:tmp $ ./test.sh
```

```
-bash: ./test.sh: Permission denied
bob:tmp $ chmod +x ./test.sh
bob:tmp $ ./test.sh
Hello, World
```

In the example above, first, the script is run as an argument to the shell. Next, the script is run directly from the shell. It is rare to have the current directory in the binary search path ($PATH) so the name is prefixed with "./" to indicate it should be run out of the current directory.

The error "Permission denied" means that the script has not been marked as executable. A quick `chmod` later and the script works. `chmod` is used to change the permissions of a file, which will be explained in detail in a later chapter.

There are different shells with their own language syntax so once you get to more complicated scripts you will want to make sure that your script knows which shell it should run under by specifying the path to the interpreter as the first line, prefixed by "#!" as shown:

```
#!/bin/sh
echo "Hello, World!"
```

The two characters, "#!" are traditionally called the hash and the bang respectively, which leads to the shortened form of *"shebang"* when they're used at the head of a script.

Incidentally, the shebang is used for traditional shell scripts and other text-based languages like Perl, Ruby, and Python. Any text file marked as executable will be run under the interpreter specified in the first line as long as the script is run directly, such as the second invocation shown in the second example in this section (that is, ./script). If you pass the script as an argument to an interpreter, such as "sh script", the given shell will be used no matter what's in the shebang line.

Before you can write shell scripts, you must become comfortable using a text editor. Traditional office tools like LibreOffice aren't appropriate for this task as shell scripts are text files; this job calls for a text editor.

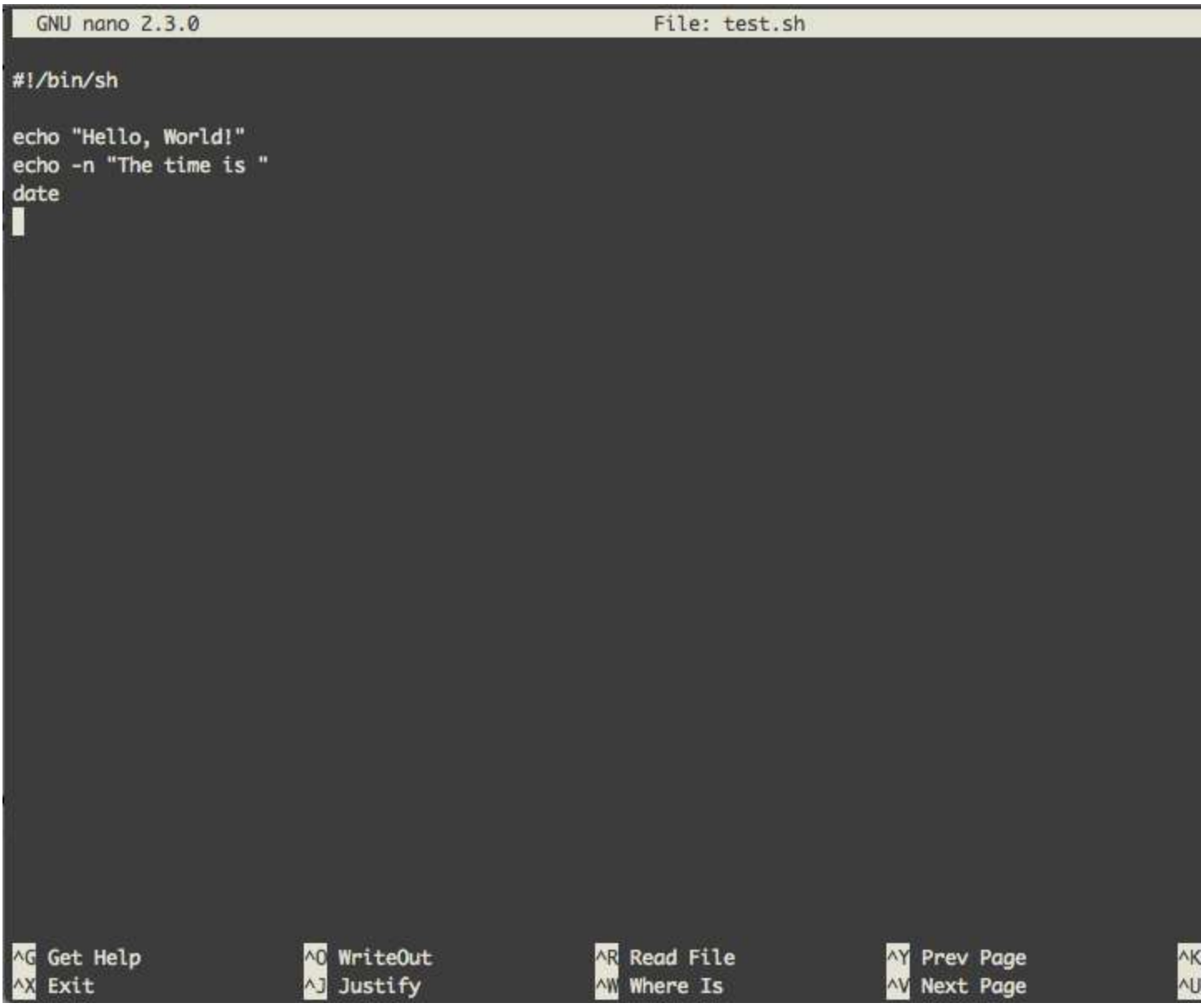
9.4 Editing Shell Scripts

UNIX has many text editors, the merits of one over the other are often hotly debated. Two are specifically mentioned in the LPI Essentials syllabus: The GNU nano editor is a very simple editor well suited to editing small text files. The Visual Editor, vi, or its newer version, VI improved (vim), is a remarkably powerful editor but has a steep learning curve. We'll focus on nano.

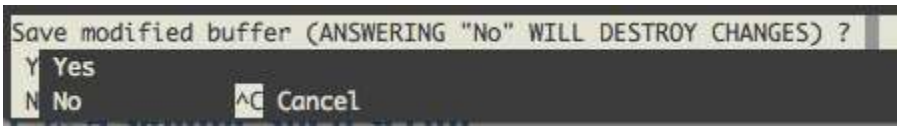
Type `nano test.sh` and you'll see screen similar to this:

Nano has few features to get in your way. You simply type with your keyboard, using the arrow keys to move around and the delete/backspace button to delete text. Along the bottom of the screen you can see some commands available to you, which are context sensitive and change depending on what you're doing. If you're directly on the Linux machine itself, as opposed to connecting over the network, you can also use the mouse to move the cursor and highlight text.

To get familiar with the editor, start typing out a simple shell script while inside nano:



Note that the bottom-left option is **^X Exit** which means “press control and X to exit”. Press Ctrl and X together and the bottom will change:



At this point, you can exit the program without saving by pressing the **N** key, or save first by pressing **Y** to save. The default is to save the file with the current file name. You can press the Enter key to save and exit.

You will be back at the shell prompt after saving. Return to the editor. This time press Ctrl and O together to save your work without exiting the editor. The prompts are largely the same, except that you’re back in the editor.

This time use the arrows keys to move your cursor to the line that has “The time is”. Press Control and K twice to cut the last two lines to the copy buffer. Move your cursor to the remaining line and press Control and U once to paste the copy buffer to the current position. This makes the script echo the current time before greeting you and saved you needing to re-type the lines.

Other helpful commands you might need are:

Command	Description
Ctrl + W	search the document
Ctrl + W, then Control + R	search and replace
Ctrl + G	show all the commands possible
Ctrl + Y/V	page up / down
Ctrl + C	show the current position in the file and the file’s size

9.5 Scripting Basics

You got your first taste of scripting earlier in this chapter where we introduced a very basic script that ran a single command. The script started with the shebang line, telling Linux that /bin/bash (which is Bash) is to be used to execute the script.

Other than running commands, there are 3 topics you must become familiar with:

- Variables, which hold temporary information in the script
- Conditionals, which let you do different things based on tests you write
- Loops, which let you do the same thing over and over

9.5.1 Variables

Variables are a key part of any programming language. A very simple use of variables is shown here:

```
#!/bin/bash


ANIMAL="penguin"
echo "My favorite animal is a $ANIMAL"
```

After the shebang line is a directive to assign some text to a variable. The variable name is ANIMAL and the equals sign assigns the string “penguin”. Think of a variable like a box in which you can store things. After executing this line, the box called “ANIMAL” contains the word “penguin”.

It is important that there are no spaces between the name of the variable, the equals sign, and the item to be assigned to the variable. If you have a space there, you will get an odd error such as “command not found”. Capitalizing the name of the variable is not necessary but it is a useful convention to separate variables from commands to be executed.

Next, the script echos a string to the console. The string contains the name of the variable preceded by a dollar sign. When the interpreter sees that dollar sign it recognizes that it will be substituting the contents of the variable, which is called *interpolation*. The output of the script is then “My favorite animal is a penguin”.

So remember this: To assign to a variable, just use the name of the variable. To access the contents of the variable, prefix it with a dollar sign. Here, we show a variable being assigned the contents of another variable!

```
#!/bin/bash


ANIMAL=penguin
SOMETHING=$ANIMAL
echo "My favorite animal is a $SOMETHING"
```

ANIMAL contains the string “penguin” (as there are no spaces, the alternative syntax without using quotes is shown). SOMETHING is then assigned the contents of ANIMAL (because ANIMAL has the dollar sign in front of it).

If you wanted, you could assign an interpolated string to a variable. This is quite common in larger scripts, as you can build up a larger command and execute it!

Another way to assign to a variable is to use the output of another command as the contents of the variable by enclosing the command in back ticks:

```
#!/bin/bash
CURRENT_DIRECTORY=`pwd`
echo "You are in $CURRENT_DIRECTORY"
```

This pattern is often used to process text. You might take text from one variable or an input file and pass it through another command like `sed` or `awk` to extract certain parts and keep the result in a variable.

It is possible to get input from the user of your script and assign it to a variable through the `read` command:

```
#!/bin/bash


echo -n "What is your name? "
read NAME
echo "Hello $NAME!"
```

Read can accept a string right from the keyboard or as part of command redirection like you learned in the last chapter.

There are some special variables in addition to the ones you set. You can pass arguments to your script:

```
#!/bin/bash
echo "Hello $1"
```

A dollar sign followed by a number *N* corresponds to the *Nth* argument passed to the script. If you call the example above with `./test.sh` Linux then the output will be “Hello Linux”. The $0 variable contains the name of the script itself.

After a program runs, be it a binary or a script, it returns an *exit code* which is an integer between 0 and 255. You can test this through the $? variable to see if the previous command completed successfully.

```
bob:tmp $ grep -q root /etc/passwd
bob:tmp $ echo $?
0
bob:tmp $ grep -q slartibartfast /etc/passwd
bob:tmp $ echo $?
1
```

The `grep` command was used to look for a string within a file with the `-q` flag, which means “quiet”. Grep, while running in quiet mode, returns 0 if the string was found and 1 otherwise. This information can be used in a conditional to perform an action based on the output of another command.

Likewise you can set the exit code of your own script with the `exit` command:

```
#!/bin/bash
# Something bad happened!
exit 1
```

The example above shows a comment (#). Anything after the hash mark is ignored which can be used to help the programmer leave notes. The “exit 1” returns exit code 1 to the caller. This even works in the shell, if you run this script from the command line and then type “echo $?” you will see it returns 1.

By convention, an exit code of 0 means “everything is OK”. Exit codes greater than 0 mean some kind of error happened, which is specific to the program. Above you saw that grep uses 1 to mean the string was not found.

9.5.2 Conditionals

Now that you can look at and set variables, it is time to make your script do different functions based on tests, called *branching*. The if statement is the basic operator to implement branching.

A basic if statement looks like this:

```
if somecommand; then
  # do this if somecommand has an exit code of 0
fi
```

The next example will run “somecommand” (actually, everything up to the semicolon) and if the exit code is 0 then the contents up until the closing “fi” will be run. Using what you know about grep, you can now write a script that does different things based on the presence of a string in the password file:

```
#!/bin/bash


if grep -q root /etc/passwd; then
  echo root is in the password file
else
  echo root is missing from the password file
fi
```

From previous examples, you might remember that the exit code of grep is 0 if the string is found. The example above uses this in one line to print a message if root is in the password or a different message if it isn’t. The difference here is that instead of an “fi” to close off the if block, there’s an

“else”. This lets you do one action if the condition is true, and another if the condition is false. The else block must still be closed with the fi keyword.

Other common tasks are to look for the presence of a file or directory and to compare strings and numbers. You might initialize a log file if it doesn’t exist, or compare the number of lines in a file to the last time you ran it. “If” is clearly the command to help here, but what command do you use to make the comparison?

The test command gives you easy access to comparison and file test operators. For example:

Command	Description
test –f /dev/ttyS0	0 if the file exists
test ! –f /dev/ttyS0	0 if the file doesn’t exist
test –d /tmp	0 if the directory exists
test –x `which ls`	substitute the location of ls then test if the user can execute
test 1 –eq 1	0 if numeric comparison succeeds
test ! 1 –eq 1	NOT – 0 if the comparison fails
test 1 –ne 1	Easier, test for numeric inequality
test “a” = “a”	0 if the string comparison succeeds
test “a” != “a”	0 if the strings are different
test 1 –eq 1 –o 2 –eq 2	-o is OR: either can be the same
test 1 –eq 1 –a 2 –eq 2	-a is AND: both must be the same

It is important to note that `test` looks at integer and string comparisons differently. 01 and 1 are the same by numeric comparison, but not by string comparison. You must always be careful to remember what kind of input you expect.

There are many more tests, such as `-gt` for greater than, ways to test if one file is newer than the other, and many more. Consult the test man page for more.

`test` is fairly verbose for a command that gets used so frequently, so there is an alias for it called '[' (left square bracket. If you enclose your conditions in square brackets, it's the same as running`test`. So, these statements are identical.

```
if test –f /tmp/foo; then
if [ -f /tmp/foo]; then
```

While the latter form is most often used, it is important to understand that the square bracket is a command on its own that operates similarly to test except that it requires the closing square bracket.

"if" has a final form, that lets you do multiple comparisons at one time using "elif" (short for else if).

```
#!/bin/bash

if [ "$1" = "hello" ]; then
  echo "hello yourself"
elif [ "$1" = "goodbye" ]; then
  echo "nice to have met you"
  echo "I hope to see you again"
else
  echo "I didn't understand that"
fi
```

The code above compares the first argument passed to the script. If it is hello, the first block is executed. If not, the script checks to see if it is goodbye and echos a different message if so. Otherwise, a third message is sent. Note that the $1 variable is quoted and the string comparison operator is used instead of the numeric version (-eq).

The if/elif/else tests can become quite verbose and complicated. The *case* statement provides a different way of making multiple tests easier.

```
#!/bin/bash

case "$1" in
hello|hi)
```

```
  echo "hello yourself"
  ;;
goodbye)
  echo "nice to have met you"
  echo "I hope to see you again"
  ;;
*)
  echo "I didn't understand that"
esac
```

The case statement starts off with a description of the expression being tested: case *EXPRESSION*in. The expression here is the quoted $1.

Next, each set of tests are executed as a pattern match terminated by a closing parenthesis. The previous example first looks for “hello” or “hi”; multiple options are separated by the vertical bar (|) which is an OR operator in many programming languages. Following that are the commands to be executed if the pattern returns true, which are terminated by two semicolons. The pattern repeats.

The * pattern is the same as an else because it matches anything. The behavior of the case statement is similar to the if/elif/else statement in that processing stops after the first match. If none of the other options matched the * ensures that the last one will match.

With a solid understanding of conditionals you can have your scripts take actions only if necessary.

9.5.3 Loops

Loops allow code to be executed repeatedly. They can be useful in numerous situations, such as when you want to run the same commands over each file in a directory, or repeat some action 100 times. There are two main loops in shell scripts: the for loop and the while loop.

For loops are used when you have a finite collection over which you want to iterate, such as a list of files, or a list of server names:

```
#!/bin/bash


SERVERS="servera serverb serverc"
for S in $SERVERS; do
  echo "Doing something to $S"
done
```

The script first sets a variable containing a space separated list of server names. The for statement then loops over the list of servers, each time it sets the S variable to the current server name. The choice of S was arbitrary, but note that the S has no dollar sign but the $SERVERS does, showing that $SERVERS will be expanded to the list of servers. The list does not have to be a variable. This example shows two more ways to pass a list.

```
#!/bin/bash


for NAME in Sean Jon Isaac David; do
  echo "Hello $NAME"
done


for S in *; do
  echo "Doing something to $S"
done
```

The first loop is functionally the same as the previous example, except that the list is passed to the for loop directly instead of using a variable. Using a variable helps the clarity of the script as someone can easily make changes to the variable rather than looking at a loop.

The second loop uses a * which is a *file glob*. This gets expanded by the shell to all the files in the current directory.

The other type of loop, a *while* loop, operates on a list of unknown size. Its job is to keep running and on each iteration perform a test to see if it should run another time. You can think of it as “while some condition is true, do stuff.”

```
#!/bin/bash


i=0
while [ $i -lt 10 ]; do
  echo $i
  i=$(( $i + 1))
done
echo “Done counting”
```

The example above shows a while loop that counts from 0 to 9. A counter variable, i, is initialized to 0. Then a while loop is run with the test being “is $i less than 10?” Note that the while loop uses the same notation as an if statement!

Within the while loop the current value of i is echoed and then 1 is added to it through the $((*arithmetic*)) command and assigned back into i. Once i becomes 10 the while statement returns false and processing continues after the loop.