SCR

Scripting

Goals of this lab:

- To learn the basics of writing shell scripts.
- To gain insight in the gains with automating administrative work.
- To gain practical experience of automating administrative work.

Prerequisites: LXB

Table of Contents

PRELAB

Exercise 1: Review and preparation

Automation typically involves writing scripts in the command shell language. Familiarize yourself with the scripting language; in particular control constructs like if statements and loops.

Report: No report required.

Exercise 2: Planning

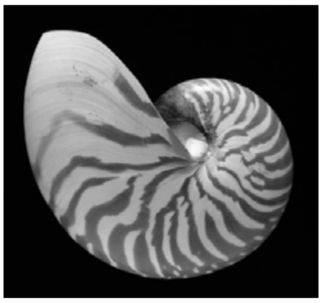
Select one of the tasks listed in part 2 or design your own according to the instructions.

2-1 Write down exactly what the script needs to do (requirements specification).

2-2 Write the script using any kind of pseudocode you like.

Report: Your requirements and your pseudocode.

MAIN LAB



A skilled system administrator automates the work as much as possible. You are to automate some aspect of the administration by writing your own scripts. You are more than welcome to come up with alternatives to any of the ideas below, but discuss it with your lab assistant before you start, to make sure that your idea is suitable for the course. If you do not want to come up with something of your own, you may choose one of the areas below. Although there are scripts that already do at least parts of the following task, you are to write your own.

Time taken 2006: Average 7 hours.

Past problems: This is really a programming lab, using a new language, so you should be familiar with the kinds of problems you might encounter.

Part 1: Scripting primer

Unix system administrators often write scripts to automate tasks. The Unix philosophy of building many small tools with well-defined functionality that can be connected by pipes makes Unix scripting a very powerful tool.

Simple scripts that tie together existing tools with fairly straightforward control are usually written in the shell command language, and are called shell scripts, while scripts that require complex control flow, I/O and networking are increasingly being written in full-fledged programming languages such as Perl or Python.

The rule of thumb used to be that any shell script should be written for the Bourne shell, the simplest shell provided with any Unix system (today, the POSIX shell, a derivative of the Bourne shell, is the corresponding target). Writing for the Bourne shell ensured maximum compatibility. Today's Linux system administrators have become used to the luxury of the Bourne Again Shell (commonly known as bash), which is shipped standard with nearly every Linux distribution. Bash has a more powerful command language, with support for simple data structures, more advanced control flow, arithmetic and a host of other useful features.

In this course, you will be expected to write scripts that are compatible with the normal Bourne shell. These can be read and executed by all Bourne shell derivatives (such as the Korn Shell, Bash, zsh and others).

A simple script

Let's look at a simple script (you can find the script in /home/TDDI05/scr/targrep):

```
#!/bin/sh

function verbose() {
  test “$VERBOSE” && echo “$@” >&2
}

function usage() {
  echo “Usage: $0 [-v] PATTERN FILE...” >&2
  exit 1
}

if [ “$1” = ‘-v’ ] ; then
  VERBOSE=1
  shift
fi

PATTERN=”$1” ; shift
test “$PATTERN” -a “$1” || usage

TMPDIR=`mktemp -d` || exit 1
for file in “$@” ; do
  if grep “$PATTERN” $file >& /dev/null ; then
    verbose “Copying $file”
    cp $file $TMPDIR
  fi
done
(cd $TMPDIR ; tar cf - .)
rm -rf $TMPDIR
```

This script creates a `tar` archive (which is output to stdout) containing all the files specified on the command line that contain a particular pattern. The script contains a number of common elements that you will find in simple scripts.

The first line

```
#!/bin/sh
```

The first line of the script indicates the interpreter that will be used to run the script. In this case, the bourne shell will be used. The hash-bang (#!) at the beginning of the first line indicates that this is a script to be run under some kind of interpreter. The rest of the line is the interpreter command line (in this case `/bin/sh`.

Function definitions

Bourne shell supports simple function definitions. This script contains two of them:

```
function verbose() {
  test “$VERBOSE” && echo “$@” >&2
}

function usage() {
  echo “Usage: $0 [-v] PATTERN FILE...” >&2
  exit 1
}
```

Functions are started with “function *name*() {“ and end with a right curly bracket. There is no explicit parameter list; even if the function takes parameters, it is declared with an empty pair of parentheses.

The first function prints a message (all arguments to the function, as indicated by “$@”) to stderr (file descriptor 2) if the VERBOSE flag is set. The definition uses logical operators for sequencing.

The second function prints an error message including the program name ($0) to stderr, then exits with an error condition.

Logical sequencing operators

The first function is used to output a string to stderr (file descriptor 2) is the VERBOSE flag is set:

```
  test “$VERBOSE” && echo $@ >& 2
```

The `test` command is run with a string containing the value of VERBOSE as its sole argument. When run like this, `test` will return a zero exit status if the string is not empty. The && that follows the invocation of test is a form of sequencing. If the command to the left of the ampersands returns a zero exit status, the command to the right is executed. So in this case, if VEROBSE is non-empty, all arguments to the function ("$@" will be printed to file descriptor 2 (stderr). There is also an operator "||", which runs the right-hand command if the left-hand command returns a non-zero exit status. You can think of "&&" and "||" as logical operators "and" and "or".

The idiom "*test* && *command*" is very useful as an alternative to a far clumsier if statement. Similarly, the idiom "*test* || *command*" is very useful when the condition of the test needs to be negated. You will also see things like "*command* || exit 1" in scripts. This idiom means "run *command*", and if anything goes wrong, exit this script with a non-zero (error) exit status.

If statements

The Bourne shell has a number of control structures. The simplest on is the "if" statement. A simple "if" statement has the following structure:

```
if list ; then
  list
else
  list
fi
```

A *list* is simply a list of pipelines (commands connected by pipes) separated by sequencing operators. In the simplest case, a *list* is a single command. The "if" statement will execute the *list* following "then" if the condition (the *list* after the "if" keyword) returns a zero exit status. If the condition returns a non-zero exit status, the *list* following the "else" keyword is executed.

In the example script, the first if statement looks like this:

```
if [ "$1" = '-v' ] ; then
  VERBOSE=1'
  shift
fi
```

When you look at this, it seems that there is no command following the "if" keyword. Instead, there is a bracket, and a logical-looking condition. In actual fact, a left square bracket is shorthand for the `test` command, and the right-hand bracket is actually a parameter to `test`. In this example, the condition tests if the first *positional parameter* equals "-v".

Positional parameters

Positional parameters are arguments to a function or a script. They are stored in the variables $1, $2, $3 and so forth (up to $9). The entire list of positional parameters can be accessed through $* and $@.

The example script uses positional parameters in several places. The previous if statement is one, and the following statements are another:

```
PATTERN="$1" ; shift
test "$PATTERN" -a "$1" || usage
```

First, the value of the first positional parameter is assigned to the variable PATTERN. Next, that positional parameter is shifted out. The next statement is an example of idiomatic use of "||", and checks that PATTERN is set to something and that there is at least one more positional parameter. If that is *not* the case, the usage function is run.

The shift command (built in to the shell) shifts all positional parameters one step, so $1 takes on the value of $2, $2 the value of $3 and so on.

Exercise 3: $* and $@

3-1 What is the difference between using $* and $@ to access all parameters to a script or function? Hint: it has to do with how spaces in parameters are handled.

Report: An example that demonstrates the difference between $* and $@.

Using backquotes

The following line in the script uses backquotes (also known as backticks) which indicate command substitution:

```
TMPDIR=`mktemp -d` || exit 1
```

The shell executes the command inside the backquotes and replaces the entire backquoted part of the command line with the output from the command. In this case, the `mktemp` command creates a directory and prints the name of the directory to stdout. The invocation of `mktemp` on the command line is replaced with the output of `mktemp`, and the command line evaluated. The result is that TMPDIR will be assigned the name of the directory created by `mktemp`.

For loops

The Bourne shell supports two kinds of looping constructs. The for loop, which loops over a set of parameters, and the while loop, which loops until a condition has been met. A simple for loop has the following structure:

```
for variable in parameters ; do
  list
done
```

The for loop is often used to iterate over positional parameters, as in this script:

```
for file in “$@” ; do
  if grep “$PATTERN” $file >& /dev/null ; then
    verbose “Copying $file”
    cp $file $TMPDIR
  fi
done
```

This loop simply loops over all positional parameters (that haven't been shifted), setting the file variable to each parameter in turn. The body of the for loop is an if statement. Here is an example of using a command other than test as the condition. The grep command searches files for a particular pattern, and returns a zero exit status if the pattern is found. It also prints messages to stdout and stderr, but since the messages are of no concern to this script, they are redirected to /dev/null.

Note that loops are commands, and thus can appear in pipelines. You can, for example, pipe the contents of a file into a while loop and use the `read` command to read one line at a time.

Exercise 4: Positional parameters in for loops

4-1 Why is $@ quoted (inside quotation marks) in the for loop?

4-2 Could $* have been used instead of $@? Explain your answer.

Report: Answers to the questions.

Exercise 5: While loops

5-1 How do while loops work?

Report: An example that demonstrates how a while loop works.

Sub-shells

The penultimate line of our script uses a feature known as a subshell:

```
(cd $TMPDIR ; tar cf - .)
```

Sub-shells are recursive invocations of the shell. They may be created in several ways. The script explicitly creates a sub-shell by using parenthesis around a list of commands. In this case, a subshell is used so the change to the current directory (the `cd` command) does not affect the main script.

Since variable assignments in a sub-shell do not affect the parent shell, constructions such as the following will not work:

```
test “$1” = “-v” && (VERBOSE=1;shift)
```

Debugging scripts

There really aren't any good debugging tools for scripts, but there is one trick that everyone should be aware of. If `/bin/sh` is executed with the –x option, it will print every command it executes. The same effect can be achieved by executing `set -x` in the shell.

To debug the example script, simply add the –x option to the command line on the first line:

```
#!/bin/sh -x
```

Changing the script every time can be impractical. To execute the script just one with the –x option, run /bin/sh explicitly:

```
$ sh -x SCRIPTNAME
```

Exercise 6: Tracing scripts

6-1 Copy the example script and add the –x option to the command line, then run the script.

6-2 Remove the –x option and run the example script again, this time invoking `sh` explicitly.

Report: No report required.

Part 2: Scripting for automation

System administrators write scripts to automate tasks all the time. Some scripts eventually evolve into full-fledged tools that are used by administrators around the world. While writing your own automation scripts, keep the following in mind:

- Make your scripts portable. Not everyone uses bash. If you *do* use bash-specific features (a common occurrence among Linux users), at lease make sure the script is run by `/bin/bash` and not `/bin/sh`.
- Make sure your script handles parameters and file names that contain spaces correctly.
- If your script outputs data to stdout, keep an eye out for commands that output data to stdout, so you don't get a mix of your intended output and other command's outputs.
- If you do I/O to files, make sure your code is secure. Use `mktemp` for temporary files.
- If you output data to a file, consider outputting to stdout instead, so the user can decide where the output goes (by piping or redirecting it).
- Automation scripts should require *no* user interaction whatsoever!

Here are some other hints:

- Programs that read passwords often explicitly read from a terminal, so normal pipes won't let you feed them passwords. Tools such as `expect` (for complex cases) and `socat` (for simple cases) help.
- For simple text processing, the `cut`, `sed` and `awk` commands are very useful (if somewhat cryptic).

- If your script handles files, make sure it can handle files with space characters in the name. Although Unix folks hate file names with spaces, Windows and MacOS users often use file names with spaces, and Unix systems often support Windows and MacOS clients. Don't forget to check files with multiple consecutive spaces in their names.

A little scripting knowledge goes a long way towards automating tasks. Some of the tasks that people often find a need to automate are listed here, and all can be solved using fairly rudimentary scripts (they require fairly simple I/O and control structures). You will choose one of these options, or come up with one of your own (check with your lab assistant that your idea is adequate) to implement.

You may implement your scripts in Bourne shell (/bin/sh, and recommended), Bash (/bin/bash), Perl or Python. If you want to use some other language, ask the lab assistant first. Whichever language you choose, your script must be well-written, well-tested and maintainable.

Option 1: Automate creation of new users

Create a tool that reads a file with lines in the following (or similar) format:

```
username:fullname
```

and creates the user, generates and sets a randomly chosen initial password, adds the user to your directory server, sets up everything required by your e-mail system, creates the home directory and copies basic configuration files to it, updates the automounter. Finally, your script is to generate a report (on the screen) with newly added users together and their respective passwords.

Demonstrate the script by adding at least 50 users to your system. There is a test file available on the course home page.

Option 2: Backup of user files

Write a script that backs up files from user directories. The script should scan all home directories for a file named .backuprc. This file contains one file or directory name per line (note that names may include space characters). The script is to collect all these files in a tar archive (one tar archive for all users), which is then stored elsewhere on the file system (e.g. /var/backup). The script is to keep only the last ten backups. The backup file name must include the date and time when the backup was made.

Option 3: System monitoring scripts

Write a script that reports the health of a Linux system. Health is defined by a set of parameters such as free memory, free disk space, load average and so on. Each parameter can have three ranges: one range that is acceptable, one which causes the script to signal a warning and one which causes the script to signal a critical error. For example, one might say that anything more than 16MB of free memory is acceptable, from 12 to 16MB causes a warning and less than 12MB causes a critical error.

Your script must produce a nicely formatted report that clearly summarizes the most important information at the top and includes a complete status report at the end and e-mails the report to root with a suitable subject line. The script should be configurable somehow (e.g. limits easily changed). It must be completely non-interactive. It must monitor at least the following parameters: free memory (including buffer memory), free disk space, number of processes, number of spamd processes, load average and CPU use.

Exercise 7: Write a script to automate a task.

7-1 Choose *one* of the alternatives above, or come up with your own.

Report: A printout of the script and a set of test cases that show how it works.

FEEDBACK FORM SCR

Complete this feedback form **individually** at the end of the lab and hand it to the lab assistant when you finish. Your feedback is essential for improving the labs. Each student should hand in a feedback form. Do not cooperate on completing the form.

You do not need to put your name on the feedback form. Your feedback will be evaluated the same way regardless of whether your name is on it or not. Your name is valuable to us in case you have made and comments in the last section that need clarifications or otherwise warrant a follow-up.

For each section, please rate the following (range 1 to 5 in all cases).

- ❖ **Difficulty:** Rate the degree of difficulty (1=too easy, 5=too difficult)
- ❖ **Learning:** Rate your learning experience (1=learned nothing, 5=learned a lot).
- ❖ **Interest:** Rate your interest level after completing the part (1=no interest, 5=high interest).
- ❖ **Time:** How long did the part take to complete (in minutes)?

	Difficulty	**Learning**	**Interest**	**Time (minutes)**
Part 1: Scripting primer				
Part 2: Scripting for automation				
Overall				

Please answer the following questions:

- ❖ What did you like about this lab?

- ❖ What did you dislike about this lab?

- ❖ Make a suggestion to improve this lab.

FEEDBACK FORM SCR

Complete this feedback form **individually** at the end of the lab and hand it to the lab assistant when you finish. Your feedback is essential for improving the labs. Each student should hand in a feedback form. Do not cooperate on completing the form.

You do not need to put your name on the feedback form. Your feedback will be evaluated the same way regardless of whether your name is on it or not. Your name is valuable to us in case you have made and comments in the last section that need clarifications or otherwise warrant a follow-up.

For each section, please rate the following (range 1 to 5 in all cases).

- ❖ **Difficulty:** Rate the degree of difficulty (1=too easy, 5=too difficult)
- ❖ **Learning:** Rate your learning experience (1=learned nothing, 5=learned a lot).
- ❖ **Interest:** Rate your interest level after completing the part (1=no interest, 5=high interest).
- ❖ **Time:** How long did the part take to complete (in minutes)?

	Difficulty	**Learning**	**Interest**	**Time (minutes)**
Part 1: Scripting primer				
Part 2: Scripting for automation				
Overall				

Please answer the following questions:

- ❖ What did you like about this lab?
- ❖ What did you dislike about this lab?
- ❖ Make a suggestion to improve this lab.

FEEDBACK FORM SCR

Complete this feedback form **individually** at the end of the lab and hand it to the lab assistant when you finish. Your feedback is essential for improving the labs. Each student should hand in a feedback form. Do not cooperate on completing the form.

You do not need to put your name on the feedback form. Your feedback will be evaluated the same way regardless of whether your name is on it or not. Your name is valuable to us in case you have made and comments in the last section that need clarifications or otherwise warrant a follow-up.

For each section, please rate the following (range 1 to 5 in all cases).

❖ **Difficulty:** Rate the degree of difficulty (1=too easy, 5=too difficult)

❖ **Learning:** Rate your learning experience (1=learned nothing, 5=learned a lot).

❖ **Interest:** Rate your interest level after completing the part (1=no interest, 5=high interest).

❖ **Time:** How long did the part take to complete (in minutes)?

	Difficulty	**Learning**	**Interest**	**Time (minutes)**
Part 1: Scripting primer				
Part 2: Scripting for automation				
Overall				

Please answer the following questions:

❖ What did you like about this lab?

❖ What did you dislike about this lab?

❖ Make a suggestion to improve this lab.