澳 門 理 工 學 院
Instituto Politécnico de Macau
Macao Polytechnic Institute

School of Applied Sciences (B.Sc. In Computing)

COMP 225: Network and System Administration Notes #2: Shell Scripting

K. L. Eddie Law, PhD

Associate Professor

Computing, MPI

Academic Year 2nd Semester, 2019-2020

Topics

- Typical kernel starting up process
- Bash
- More commands
- Shell scripting

Typical Kernel Starting Procedure

- The initialization process starts by initializing the peripherals it needs, typically the serial port, network, and, maybe, USB
- If the auto boot sequence is not interrupted, it usually copies the kernel from Flash to RAM and starts it executing
- The Linux kernel, a compressed image, decompresses itself into the appropriate location in RAM
- It then initializes all of the kernel subsystems, such as memory management and drivers for all of the peripheral devices in the system (must initialize the devices to suit its own requirements)

Typical Startup Operations (cont'd)

- During the initialization process the kernel spews out a large number of messages describing what it is doing
- Next it mounts the root file system. Up to this point, the kernel has been running in kernel space. Finally it starts the `init` process, which makes the transition to user space
- The last thing the kernel boot does is start a process with PID 1 (`/sbin/init`). This then becomes the ultimate parent of every other process in the system

Systemd

- ***systemd***
  - The latest design of a kernel initialization mechanism encompassing something like 900 files
  - Manages and acts on objects called ***units***
  - Its most common type of units is "***service***," represented by a file that ends in `.service` (check those `*.service` files in `/lib/system/system`)
  - Open up, for example, `iscsid.service`; if there is a parameter `WorkingDirectory`, then it indicates the location of support files for the executable, which is identified by `ExecStart`

Systemd

- Use `systemctl` command to manage `systemd`, it understands

start <unit>	Start the specified unit(s)
stop <unit>	Stop the specified unit(s)
restart <unit>	Restart the specified unit(s). If not running they will be started
reload <unit>	Ask specified unit(s) to reload their configuration files
status <unit>	Display the status of the specified unit(s)
enable <unit>	Create symlinks to allow unit(s) to be automatically started at boot
disable<unit>	Remove the symlinks that cause the unit(s) to be started at boot
deamon-reload	Reloads the systemd manager's configuration. Run this any time you make a change to a systemd file

- First, try command `systemctl` with no arguments to see the display

Shell – The Interface

- Several shell programs in common use, all serve the same basic purpose, yet differ in details of syntax and features
  - ***Bourne Again SHell*** – `bash`, a "reincarnation" of the Bourne shell, `sh`, written by Stephen Bourne at Bell Labs for Unix 7; the default on most Linux distributions; should use it unless there is a good reason to switch to another shell
  - ***Korn Shell*** – `ksh`, developed by David Korn at Bell Labs in the early 1980s, more advanced programming facilities than `bash`, but nevertheless maintains backward compatibility
  - ***Tenex C Shell*** – `tcsh`, a successor to the C shell, `csh` that was itself a predecessor to the Bourne shell. Tenex was an OS that inspired some of the features of `tcsh`
  - ***Z Shell*** – `zsh`, described as an extended Bourne shell with a large number of improvements, including some of the most useful features of `bash`, `ksh`, and `tcsh`

About Shell Commands

- A few commands were experienced, let's refresh
- A command with other info on the same line are run together, e.g.,

  `$ ls -la`
  - There are two entries on this "***line of script***"
  - Since $ is for prompting the user, the first input entry is "`ls`" ← the input command (represented by $0 in bash)
  - The second entry is "`-la`" ← the first argument to the command (represented by $1 in bash)
  - For each input line, Bash can intake numerous parameters, but only $0 to $9 values (see how we use them later on…)

More Shell Commands (1)

- `$ cat <filename>`
  - Displays a text File, the name of this command is derived from “concatenate,” which means to join together, one after the other, e.g.,
    ```
    $ cat test1 test2 > test3
    ```
  - If both `test1` and `test2` have data content, they are concatenated and saved in file `test3`. The new file will be generated, if it does not exist
    ```
    $ cat test1 test2 >> test3
    ```
  - Append both `test1` and `test2` to the end of file `test3`
- `$ [ less | more ] <textfile>`
  - Displays the content of the file on screen, the command `less` permits to use arrow keys to scroll up and down
  - Press the key “q” to exit from the utility

Shell Commands (2)

- `$ whoami`
  - Who you are
- `$ logname`
  - who you are logged in as
- `$ id`
  - Info about you and your groups
- `$ groups`
  - Info about groups
- `$ uname -a`
  - Info about the operating system
- `$ hostname`
  - Info about the name of the computer

Shell Commands (3)

- `$ date`
  - Displays the date
- `$ cal`
  - The text calendar (can check out command `ncal`), can run such as
    ```
    $ cal 2_digit_month 4_digit_year
    ```
  - For example, `$ cal 01 2020`
- `$ uptime`
  - Since the time that the computer was booted up
- `$ wc <filename>`
  - Counts number of words or lines in a file
    ```
    $ wc -l <fn>          counts lines
    $ wc -w <fn>          counts words
    ```

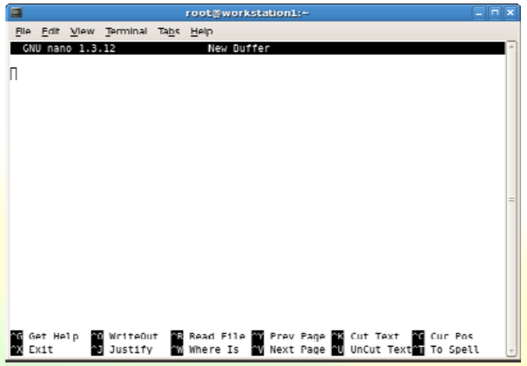
Shell Commands (4)

- `$ file <filename>`
  - Provides information about a file type
- `$ stat <filename>`
  - Provides more detailed information about a file
- `$ tail -x`
  - Displays the last *x* lines from a file
- `$ head -x`
  - Displays the first *x* lines from a file
- `$ clear`
  - Clears the screen

More on commands later!!

Text Editors on Terminals

- `$ vi` (or `vim`) `<filename>`
  - A bit complicated for beginners… hence
- `$ nano <filename>`



```
  GNU nano 1.3.12              New Buffer

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Nano

- Available options are listed at the bottom bar, the ^ is the "control" key, commonly used keys
  - `cntl-x` = exit, will prompt you to save if you have changes
  - `cntl-o` = save
  - `cntl-r` = insert file
  - `cntl-k` = delete a line
  - `cntl-u` = undelete previous contiguous deletions
- Note: the editor nano is available in Red Hat systems on rescue disk
- Let's try it out **now**!!

What is Shell Scripting?

- We have been doing Linux commands on input prompts in a live terminal
- Can we run many of them together in one shot?
- Putting all these lines of scripts together into one file, and the file is a ***script*** file
  - Bash is a scripting language
  - Scripts can automate numerous administrative works

What to Cover in Shell Scripting?

- Basics of Linux shell scripting
  - Definition of shell scripts
  - Uses of shell scripts
- Writing shell scripts
  - Control statements
  - Command lines

Running a Script File

- `$ echo ls > lsfile`
  - echo is a command to show data on screen, but “>” redirects it into a file named `lsfile`
- Run the file with
  - `$ . lsfile` (at least a space between a “dot” and the filename)
  - `$ source lsfile`
  - `$ bash lsfile`
  - What did you see? They work!?
- Try `$ ./lsfile`
  - Seeing anything??

Running a Script File (cont’d)

- Make a script file running by itself
- begin the first line with a #! and a shell command (full path of shell executable)

  ```
  $ which bash
  ```
  - Gives the full path of the executable bash
  - Add it as the first line of script in the file, i.e.,

    ```
    #!/usr/bin/bash
    ```
  - Using a text editor to add it the first line in file “`lsfile`”, then run it with

    ```
    $ ./lsfile
    ```

Running a Script File (cont’d)

- In fact, it was not the issue of having the line “#!...” but the mode of the file, make it executable
  - `$ chmod +x lsfile`
  - `$ ./lsfile`
- If no shell (`#!`) is specified in the script file, the default is to choose the current executing shell
- Different scripting languages may act differently
  - Hence the `#!` line is important
  - The bash is popular
- **Recalling**: a script doesn’t need to be executable if it is an input argument to the `bash` command

Let’s Do More Complicated Scripts

Why Writing Script Files?

- To avoid repetition:
  - If you do a sequence of steps with standard commands over and over, why not do it all with just one command?
  - Or in other words, store all these commands in a file and execute them one by one
- To automate difficult tasks:
  - Many commands have subtle and difficult options that you don't like to remember or figure out every time

Environment Macro Variables

- There are environment variables, run

```
$ env
```

  - Shows all variables in system

```
$ echo $PATH
```

  - Showing the $PATH variable in Linux (the %PATH% in Windows)

```
$ echo $HOME
```

  - Showing the home directory of a user

Example

- Suppose we are running out of space, and there are files no longer useful anymore
- Removing them with

```
rm -rf $HOME/.netscape/cache
rm -f $HOME/.netscape/his*
rm -f $HOME/.netscape/cookies
rm -f $HOME/.netscape/lock
rm -f $HOME/.netscape/.nfs*
rm -f $HOME/.pine-debug*
rm -rf $HOME/nsmail
```

Remark: these files not in our systems, example only

Example (cont'd)

- Put all those commands into a shell script, called `myscript`
- Add the **#!** on the first line in file

```
elaw@s1:~$ cat myscript
#!/usr/bin/bash
rm -rf $HOME/.netscape/cache
rm -f $HOME/.netscape/his*
rm -f $HOME/.netscape/cookies
rm -f $HOME/.netscape/lock
rm -f $HOME/.netscape/.nfs*
rm -f $HOME/.pine-debug*
rm -rf $HOME/nsmail
```

Example (cont'd)

- Run the script with
- Step 1: make it executable

```
$ chmod u+x myscript
```

- Step 2: run it

```
$ ./myscript
```

- Each line of the script is processed in order

Shell Variables

- Declare and assign value to a variable:

  `varname=varvalue`

  - `varname` is the name of the variable
  - `varvalue` is the value of the variable
  - No spaces on both sides of the "="
- Make it an environment variable, using "`export`"

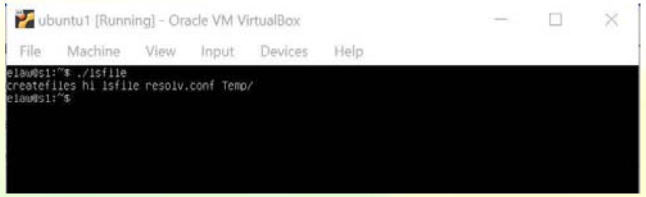
  `export varname=varvalue`

Shell Variables (cont'd)

- Use a text editor to type in

```
#!/bin/bash
filelist=`ls -F`
echo $filelist
```

- `` ` ` `` backquotes, i.e., usually the key on the left of the number 1
- Backquotes deliver the execution output of the content enclosed
- The output is assigned to the variable `filelist`
- Run it with $ ./lsfile


ubuntu1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

```
elaw@s1:~$ ./lsfile
createfiles hi lsfile resolv.conf Temp/
elaw@s1:~$
```

Evaluate Expressions

- Two expressions for evaluations – calculate results
  - The **`$(( ))`** expression operator
  - The **`expr`** command
- Examples:

```
elaw@s1:~$ value=$((1+2))
elaw@s1:~$ echo $value
3
elaw@s1:~$
```

Space

```
elaw@s1:~$ value=`expr 1 + 2`
elaw@s1:~$ echo $value
3
elaw@s1:~$
```

Notes 1

- Why do we need the **expr** command or **$((…))** expansion operator ?
- Example

```
elaw@s1:~$ value=1+2
elaw@s1:~$ echo $value
1+2
elaw@s1:~$
```

NOTES: 1+2 is copied as is into *value* and not the result of the expression, to get the result, we need **$((…))** or **expr**

Notes 2

- Variables as arguments

```
elaw@s1:~$ count=5
elaw@s1:~$ count=$(($count+1))
elaw@s1:~$ echo $count
6
elaw@s1:~$
```

NOTES: *count* is replaced with its value by the shell !

Notes on expr and $((…))

- expr or $((…)) supports the following operators:
  - Arithmetic operators: +,-,*,/,%
  - Comparison operators: <, <=, ==, !=, >=, >
  - Boolean/logical operators: &, |
  - Parentheses: (,)
  - Precedence is the same as C, Java
- N.B. for expr, the multiplication * may have to be added with an escape \ character

Control Statements

- Without control statements, execution within a shell scripts flows from one statement to the next in succession
- Control statements control the flow of execution in a programming language
- 3 most common types of control statements:
  - **Conditionals**: if/then/else, case, ...
  - **Loop statements**: while, for, until, do, ...
  - **Branch statements**: subroutine calls (good programming practice), goto (usage not recommended)

`for` Loops

- `for` loops allow repetition of a command for a specific set of values
- Syntax:

```
for var in value1 value2 ...
do
   command_set
done
```

- `command_set` is executed with each value of `var`, i.e., the sequence of (`value1`, `value2`, …)

Notes on the `for` Loop

- Example: listing all files in a directory

```
#!/bin/bash
for i in *
do
    echo $i
done
```

NOTES: * is a wildcard that stands for all files in the current directory, and `for` will go through each value in `*`, which are all the files and `$i` is the filename

Notes on the `for` Loop

- Output of the example

```
elaw@s1:~$ chmod u+x listfiles
elaw@s1:~$ ./listfiles
a
b
c
html
listfiles
elaw@s1:~$
```

Conditionals

- Conditionals are used to “test” something
  - In Java or C, usually test if a Boolean expression is true or false.
  - In bash shell script, the only thing you can test is whether or not a command is “***successful***”
- Every well behaved command returns back a **return code**
  - **0** if it was successful
  - Non-zero if it was unsuccessful (actually 1..255)
  - Different from Java or C

The `if` Command

- Simple form:

```
if decision_command_1
then
    command_set_1
fi
```

- The importance of having "`then`"
  - Each line of a shell script is treated as one command
  - The "`then`" is a command in itself
  - Though it is part of the "`if`" structure, it is treated separately

The `if` Example

grep returns 0 if it finds something
returns non-zero otherwise

```
if  grep unix myfile > /dev/null
then
    echo "It's there"
fi
```

redirect to /dev/null so that "intermediate" results do not get printed

Using `else` with `if` Expression

- Example

```
#!/bin/bash
if grep "UNIX" myfile > /dev/null
then
    echo  UNIX occurs in myfile
else
    echo  No!
    echo  UNIX does not occur in myfile
fi
```

Using `elif` with `if` Expression

- Example

```
#!/bin/bash
if grep "UNIX" myfile > /dev/null
then
    echo  UNIX occurs in myfile
elif grep “DOS” myfile > /dev/null
then
    echo DOS appears in myfile not UNIX
else
    echo  nobody is here in myfile
fi
```

Using `:` in Shell Scripts

- Sometimes, we do not want a statement to do anything.
- In this case, a colon ':' can be used

```
if grep UNIX myfile > /dev/null
then
:
fi
```

- Does not do anything when the word "UNIX" is found in `myfile`

The `test` Command

- For checking validity, 3 common types of checking
  - Check on files
  - Check on strings
  - Check on integers

The `test` on Files

- **`test -f file`**: does file exist and is not a directory?
- **`test -d file`**: does file exist and is a directory?
- **`test -x file`**: does file exist and is executable?
- **`test -s file`**: does file exist and is longer than 0 bytes?

The `test` on Files: Example

```
#!/bin/bash
count=0
for i in *; do
if test -x $i
then
    count= $(($count+1))
fi
done
echo Total of $count files executable
```

NOTE: *$count+1* serves the purpose of count++

The **test** on Strings

- **`test -z string`**: is `string` of length 0?
- **`test string1 = string2`**: does `string1` equal `string2`?
- **`test string1 != string2`**: not equal?

Space

The **test** on Strings: Example

```
#!/bin/bash
if test -z $REMOTEHOST
then
:
else
    DISPLAY="$REMOTEHOST:0“
    export DISPLAY
fi
```

NOTES: This example tests to see if the value of REMOTEHOST is a string of length > 0 or not, and then sets the DISPLAY to the appropriate value.

The **test** on Integers

- **`test int1 -eq int2`**: is `int1` equal to `int2`?
- **`test int1 -ne int2`**: is `int1` not equal to `int2`?
- **`test int1 -lt int2`**: is `int1` less than to `int2`?
- **`test int1 -gt int2`**: is `int1` greater than to `int2`?
- **`test int1 -le int2`**: is `int1` less than or equal to `int2`?
- **`test int1 -ge int2`**: is `int1` greater than or equal to `int2`?
- Alternatively, can use **`(( … ))`** to test mathematical expressions
  - **`(( int1 == int2))`**
  - **`(( int1 != int2))`**
  - …

The **test** on Integers: Example

```
#!/bin/bash
smallest=10000
for i in 5 8 19 8 7 3
do
    if test $i -lt $smallest
    then
        smallest=$i
    fi
done
echo $smallest
```

NOTES: This program calculates the smallest among the numbers 5, 8, 19, 8, 3.

Alias of **test**: [ ]

- The **[ ]** is the alias of the **test** command
- Expression to test is in the brackets, and each bracket must be surrounded by spaces

```
#!/bin/bash
smallest=10000
for i in 5 8 19 8 7 3
do
    if [ $i -lt $smallest ]
    then
        smallest=$i
    fi
done
echo $smallest
```

The **while** Loop

- **while** loops repeat statements as long as the next Unix command is successful
- Works similar to the **while** loop in Java or C

```
#!/bin/bash
i=1
sum=0
while [ $i -le 100 ]
do
    sum=$(($sum+$i))
    i=$(($i+1))
done
echo The sum is $sum.
```

NOTES: The value of i is tested in the while to see if it is less than or equal to 100.

The **until** Loop

- **until** loops repeat statements until the next Unix command is successful
- Works similar to the **do-while** loop in C

```
#!/bin/bash
x=1
until [ $x -gt 3 ]
do
    echo x = $x
    x=$(($x+1))
done
```

NOTES: The value of x is tested in the until to see if it is greater than 3

Command Line Arguments

- Shell scripts would not be very useful if we could not pass arguments to them on the command line
- Without arguments, restricts the usefulness of the script
- Parameters to any program, e.g.,
  $ ls -t foo
- The '**-t**' and **foo** are parameters to the program **ls**
- This command line has three parameters: **ls**, **-t** and **foo**.

NOTES: command is also part of the command line

Command Line Arguments (cont'd)

- Shell script arguments are "numbered" from left to right
  - **$1** - first argument after command.
  - **$2** - second argument after command.
  - ... up to **$9**
- They are called "positional parameters"
- Their **position** in the command line determines their value

Command Line Arguments (cont'd)

- Ex: Find out if a string appears in file
- Run **`$ mystr string file`**

```
elaw@s1:~$ cat myTime
#!/bin/bash
grep $1 $2
elaw@s1:~$ ./myTime DateTime myTime.py
From DateTime import DataTime
elaw@s1:~$
```

NOTES: $1 has value ***DateTime*** and $2 has value ***myTime.py***

Command Line Arguments (cont'd)

- Other variables related to arguments:
  - **$0** → Name of the command running
  - **$*** → All the arguments (even if there are more than 9)
  - **$#** → The number of arguments
- Example using these special variables

```
elaw@s1:~$ cat cmd_line
#! /bin/bash
echo "$0 = name of the command"
echo "$* = list of arguments"
echo "$# = total number of arguments"
elaw@s1:~$
```

Command Line Arguments (cont'd)

- Example output

```
elaw@s1:~$ ./cmd_line
./cmd_line = name of the command
0 = total number of arguments

elaw@s1:~$ ./cmd_line 1 2 3 4 5
./cmd_line = name of the command
1 2 3 4 5 = list of arguments
5 = total number of arguments

elaw@s1:~$
```

More on the `bash` Variables

- 3 basic types of variables in a shell script
- Positional variables: $1, $2, $3, ..., $9
- Keyword variables: e.g., $PATH, $HOME, and anything else we may define
- Special variables:
  - $! ← return process id of last background process to finish
  - $? ← return status of last foreground process to finish
  - $$ ← the process id of the current shell
  - There are many more others that we can find out about using `$ man sh`

Reading Input

- All this while, we have talked about shell scripts that do useful work and write some output
- What about reading input???
- Using the **`read`** command
  - Reads one line of input and assigns it to variables given as arguments
  - Data type of variable does not matter, shell has no concept of data types

Notes on *read*

- Syntax:
  - **read var1, var2, var3 ....**
  - Reads a line of input from standard input.
  - Assigns first word to var1, second word to var2, ...
  - The last variable gets any excess words on the line.

Reading Input (cont'd)

- Syntax:
  - **`read var1, var2, var3 …`**
  - Reads a line of input from standard input
  - Assigns first word to **`var1`**, second word to **`var2`**, ...
  - The last variable gets any excess words on the line
- Example:

```
elaw@s1:~$ read var1 var2 var3
this is to test the read
elaw@s1:~$ echo $var1
this
elaw@s1:~$ echo $var2
is
elaw@s1:~$ echo $var3
to test the read
elaw@s1:~$
```

NOTES: `var3` has the rest of the string "to test the read"

Remark

- We have done a lot of works on shell scripting!!