csce215 — UNIX/Linux Fundamentals
Spring 2022 — Lecture Notes: Automating Stuff

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with some supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

(8.1) *Last time*

Last time, we covered three important but assorted topics:

- Running shell commands in the background using **job control**.
- Listing and terminating **processes**.
- Understanding and modifying **permissions** on files and directories.

Today, we'll pull together many of the ideas from throughout the semester to see how to create **shell scripts**.

- What are shell scripts?
- How can shell scripts be created?
- How can shell scripts be executed?
- Some shell features that are mostly only useful inside scripts.

(8.2) *What is a shell script?*

A **shell script** is text file containing shell commands, intended to be executed in order.

*Recall from Chapter 1 that all of the commands we've been learning in this course are read and executed by a special program called the **shell**, specifically the* `bash` *shell.*

You already know a lot about shell scripting!

- Everything that works at the command line is fair game in a shell script. Commands, pipes, redirection, etc. ☺
- Everything we'll see here in Chapters 8 and 9 in the context of scripts would also work fine on the command line. ☺

(8.3) *Why bother with scripting?*

Creating scripts can help you **save time** and **avoid mistakes**.

- Avoid typing things again and again.
- Get it right once and not worry about the details in the future.

Remember: One of the advantages of a command-based (rather than 'clicking-based') system is that it makes it easier to let the computer handle repetitive things for us.

(8.4) *A small example*

Here's a small example that you saw in Lab 7.

execute-me

```
#!/bin/bash

num_procs="$(ps -A | wc -l)"
num_users="$(who | wc -l)"

echo There are currently "$num_procs" processes and "$num_users" users.
```

To make a file usable as a shell script, we need to do three things:

1. Make it executable.
2. Specify the interpreter.
3. Put it where the shell can find it.

```
$ ./execute-me
There are currently 347 processes and 2 users.
```

(8.5) *Reminder: Execute permission*

Shell scripts are programs, so they must have execute permission if we want to run them directly.

```
$ chmod -v u+x execute-me
mode of 'execute-me' changed from 0644 (rw-r--r--) to 0744 (rwxr--r--)
```

(8.6) *Interpreter directives*

A script is a program intended to be read and executed by a separate program called an **interpreter**.

To execute a script, the system needs to know which interpreter to use. In the Linux world, we do this by including an **interpreter directive** as the first line of the file:

```
#!/bin/bash
```

This line, also called a **'sharp-bang' line**, has two parts:

1. The exact characters '`#!`'.
2. The full path to the interpreter program. The executable for the interpreter we want is a file called `bash` in the directory `/bin`.

We'll focus in this course on shell scripts for `bash`, but interpreter directives can be used for other interpreters (i.e. other scripting languages) as well.

Here's an example from Chapter 7, which refers to the Python 3 interpreter:

alive.py

```
#!/usr/bin/python3

# This is an example of a program designed to be
# run in the background.

import time
import datetime

while True:
    print(f'Still alive at {datetime.datetime.now()}!')
    time.sleep(3)
```

(8.7) *Finding executables*

To execute programs (including scripts), your shell must be able to find those programs.

There are two options:

1. Normally, the shell will look in an environment variable called `PATH` for a list of directories to search, separated by colons.

   ```
   $ echo $PATH
   /home/jokane/projects/add-music:/home/jokane/bin/mash:/home/jokane/bin/bang:/hom
   e/jokane/bin/off:/home/jokane/bin/install-packages:/home/jokane/bin/photo-tools:
   /home/jokane/bin/local-sync:/home/jokane/bin/audit-git:/home/jokane/bin/audit-fi
   les:/home/jokane/bin/bib-scripts:/home/jokane/bin/latex-scripts:/home/jokane/bin
   :/usr/local/texlive/2021/bin/i386-linux:/usr/local/texlive/2021/bin/x86_64-linux
   :/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/lo
   cal/games:/snap/bin
   ```

   Most commands refer to programs found this way.

2. For programs that are not in your `PATH`, you can tell the shell explicitly where to look by mentioning a directory.

   For example, mention the current directory using a period '`.`':

   ```
   $ ./execute-me
   There are currently 347 processes and 2 users.
   ```

   Or mention a directory using an absolute path starting with '/':

   ```
   $ /class/215/recbash
   ```

(8.8) *Where is this program?*

To see the path of the program that the shell would run, use the `which` command.

`which`

Look for an executable with the given name.

Example:

```
$ which ls
/usr/bin/ls
```

```
$ which python3
/usr/bin/python3
```

If no program with the given name is found, `which` produces no output:

```
$ which xyzzy
```

Note: A very small number of commands are **built-in** to the shell, so `which` cannot find them, even though they run just fine.

```
$ which cd
```

(8.9) *Shell variables*

A **shell variable** is a name associated with a value.

To define a shell variable, use an **assignment** statement:

```
$ name=Ernie
```

Note: No spaces around the equals sign.

To access the value of a shell variable, use a `$` with the name:

```
$ echo "$name"
Ernie
```

```
$ ls "$name"
ls: cannot access 'Ernie': No such file or directory
```

(8.10) *Variables and quoting*

Remember that we must use quotation marks to tell the shell to treat something with spaces as a single item.

```
$ name=Bert and Ernie
/bin/bash: line 4: and: command not found
$ name="Bert and Ernie"
```

```
$ ls $name
ls: cannot access 'Bert': No such file or directory
ls: cannot access 'and': No such file or directory
ls: cannot access 'Ernie': No such file or directory
$ ls "$name"
ls: cannot access 'Bert and Ernie': No such file or directory
```

(8.11) *Environment variables*

Shell variables are usually only available to the shell in which they are defined.

To make variables available to subprocesses, we must declare them to be **environment variables** using the `export` command.

`export`

Declare that a variable should be treated as an environment variable, and available to subprocesses launched by this shell.

print-name

```
#!/bin/bash

echo "$name"
```

Here's an example of an ordinary shell variable compared to an environment variable:

```
$ echo $name
Bert and Ernie
$ ./print-name

$ export name
$ ./print-name
Bert and Ernie
```

(8.12) *Command line arguments*

Arguments given on the command line are available to our scripts using special variables `$1`, `$2`, ...

Example:

find-by-name

```
#!/bin/bash
# Find files with names that contain the
# first argument given on the command line.
find -type f -name "*$1*"
```

To refer to **all** of the command line arguments, use `"$@"`.

Here's an example we used back in Chapter 4. (It also shows integer variables and a simple loop):

showargs

```
#!/bin/bash

i=0
for arg in "$@"
do
    let i++
    echo Arg $i is [$arg]
done
```

(8.13) *An automatic script*

Pulling together all of the ideas from today: When you start a new shell, it automatically executes a file called

`~/.bashrc`

(Reminder: '~' means 'home directory'. Starting with a '`.`' means it's treated as a hidden file.)

This file is a great place to set environment variables, such as PATH.

You can also change your prompt, set up aliases, etc.

Here's an example:

.bashrc

```
# Use colors for ls and grep, if they're
# available.
alias ls='ls --color=auto'
alias grep='grep --color=auto'

# Set PATH to find my scripts.
export PATH="~/bin:$PATH"
export PATH="~/bin/audit-git:$PATH"
export PATH="~/bin/photo-tools:$PATH"
export PATH="~/bin/install-packages:$PATH"
export PATH="~/bin/mash:$PATH"
export PATH="/usr/local/texlive/2021/bin/x86_64-linux:$PATH"

# Tab-completion and custom prompt for git.
if [ -e /etc/bash_completion.d/git-prompt ]
then
  source /etc/bash_completion
  source /etc/bash_completion.d/git-prompt
  export PS1='\h:\W$(__git_ps1 " (%s)")\$ '
  export GIT_PS1_SHOWSTASHSTATE=1
  export GIT_PS1_SHOWUPSTREAM="auto"
  export GIT_PS1_SHOWUNTRACKEDFILES=1
  export GIT_PS1_SHOWDIRTYSTATE=1
fi
```

(8.14) If you make a mistake

If you edit `~/.bashrc` and make a mistake setting `$PATH`, you might have trouble using `vim` to re-open the file to fix the mistake.

```
$ vim ~/.bashrc
$ bash
$ echo $PATH
/home/jokane/bin
$ ls
/bin/bash: line 4: ls: command not found
$ vim
/bin/bash: line 5: vim: command not found
```

If this happens, use the full path to `vim` instead of relying on `$PATH`, so you can edit the file and fix it:

```
$ /usr/bin/vim ~/.bashrc
```

(8.15) *Sample final exam questions*

1. A bash shell script should begin with which two characters?

   A. `/!`
   B. `#!`
   C. `%%`
   D. `#/`

2. Which of the following is a correct interpreter directive for bash?

   A. `#!/etc/bash`
   B. `#!/bin/bash`
   C. `#!/bin/sh`
   D. `#!/home/jokane/bin/bash`

3. What is the purpose of the special variables `$1`, `$2`, ...?

   A. To refer to the previous commands executed in the shell
   B. To refer to arguments given on the command line to a script
   C. To refer to the parent directories of the current directory
   D. To refer to the other users logged in to the system

4. Which of these commands prints the value of the variable `QUUX`?

   A. `show QUUX`
   B. `echo $QUUX`
   C. `echo QUUX`
   D. `cat $QUUX`

5. Which of the following steps is *not* necessary to make a file containing commands usable as a bash shell script?

   A. Compiling the file into an executable using a bash compiler.
   B. Putting the file in a place where the shell can find it.
   C. Making the file executable.
   D. Specifying the interpreter using an interpreter directive.

6. Which of these commands assigns a value to the shell variable `FOO`?

   A. `FOO=bar`
   B. `bar -> foo`
   C. `bar->foo`
   D. `foo =bar`

7. Which environment variable contains a list of directories that should be searched for programs to execute?

   A. `SHELL`
   B. `SEARCH`
   C. `PWD`
   D. `PATH`

8. Which of the following commands declares that the variable `QUUZ` should be available to subprocesses launched by this shell?

   A. `subproc QUUZ`
   B. `global QUUZ`
   C. `export QUUZ`
   D. `shipout QUUZ`

9. Which of the following shell features can be used in a shell script?

A. pipes
B. output redirection
C. input redirection
D. all of the above

10. The PATH variable is a list of directories, separated by ______.

A. periods (.)
B. commas (,)
C. colons (:)
D. semicolons (;)

11. Which of these commands will look for an executable with the given name?

A. `where`
B. `which`
C. `what`
D. `why`