UNIX 202

Ryan DeRue, Senior Computational Scientist

PURDUE UNIVERSITY
Rosen Center for Advanced Computing

Unix 202

Outline

What to expect from Unix 202

Objectives

- Learn about the different Shell start-up files, the order in which they are sourced, and how they affect your environment
- Discuss common uses for Bash scripts, the way they are organized, and common constructs within them
- Discuss how to schedule Bash scripts to run even when we aren't logged into the system

Unix 202

Shell Start-Up Files

BASH Start-Up Files

What do we mean by Start-Up files?

- Files that are sourced every time we login to our shell
  - To source a file is to execute within your current shell every line of code one at a time
  - A built-in for modifying a shell: `source`
    - Usage: `source filename [arguments]`
    - Can also use: `. filename [arguments]`
- By creating files that are sourced every time we login to our shell we can perform the work of configuring our shell a single time
- System-wide and User-level start-up files
  - `/etc/profile`: System-wide profile for all users
  - `~/.bash_profile`: User-level profile sourced by BASH
  - `~/.bash_login`: Legacy file sourced to conform to `/bin/csh`
  - `~/.profile`: Legacy file sourced to conform to `/bin/sh`

BASH Start-Up Files

What do we use Start-Up files for?

- Exporting important environment variables to our shell
  - Example: `export PATH=$PATH:/home/rderue/bin/`
    - Ensures that the `bin/` directory in my home directory is checked for executable files
- Creating shortcuts for our most frequently used commands
  - A built-in for creating shortcuts: `alias`
  - Usage: `alias [-p] [name[=value]]`
  - Causes `name` to perform the command with arguments given by `value`
  - Example `alias ll="ls –l"`
    - Allows me to use `ls` in long mode just by typing "`ll`"

BASH Start-Up Files

What if we don't want to run our start-up files?

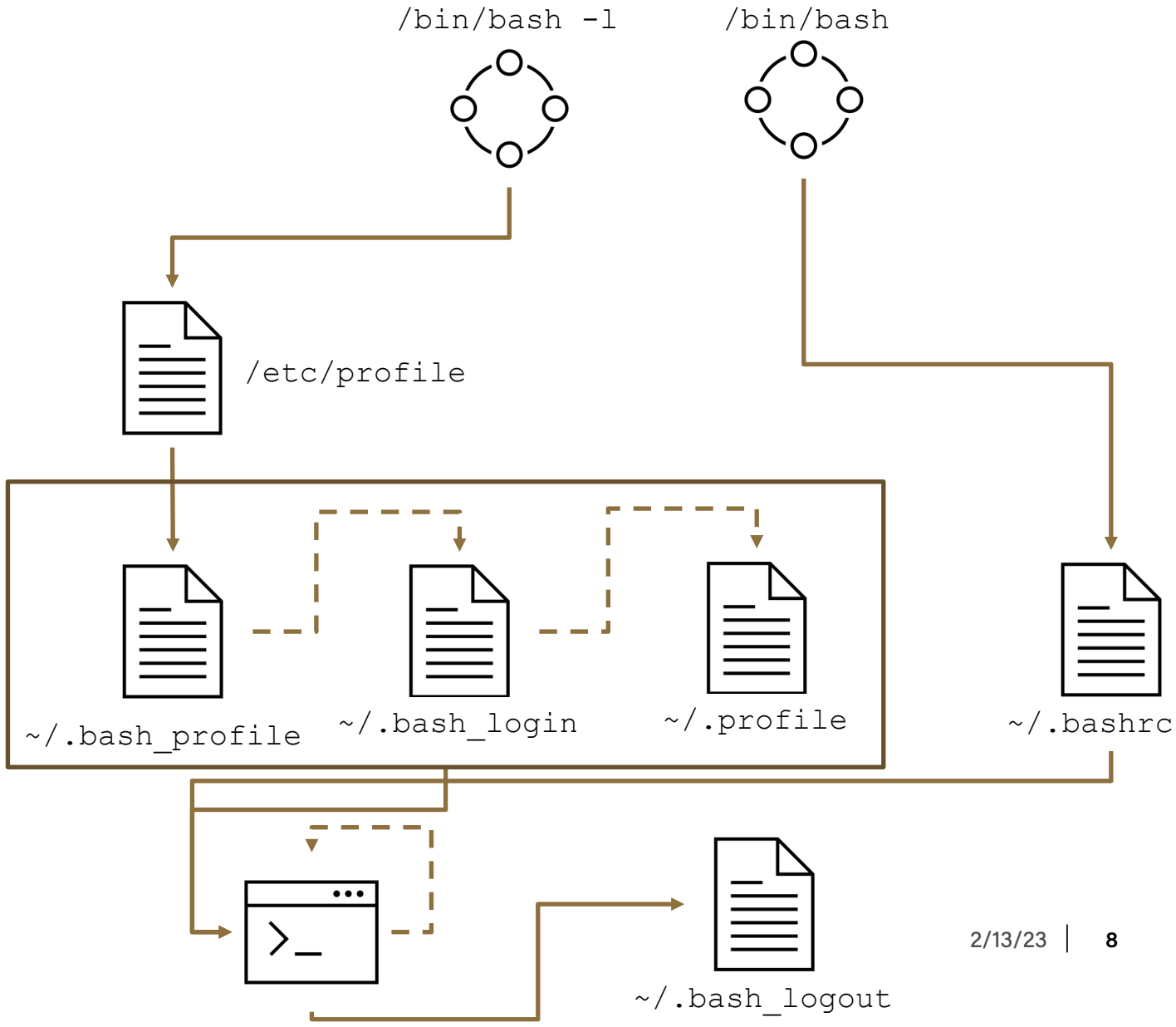
- When might we not want to use a login shell?
  - Many times, certain applications which connect to a server running a *nix OS do not want our modifications to our shell to get in their way
  - When we are trying to fix our start-up files

- Login Shells vs. Non-login Shells
  - `bash -l vs. bash`
  - Non-login shells will not source the files we have talked about
  - Non-login shells source a file named `~/.bashrc` instead
    - It's common that as part of your `.bash_profile`, you `source ~/.bashrc`

BASH Start-Up Files

In Summary

- We use BASH start-up files to ensure we have a consistent environment every time we log into our shell
- A shell sets up its environment differently depending on whether its in login or non-login mode
  - In non-login mode, only `~/.bashrc` is sourced. In login mode `/etc/profile/` and *one* of the user profiles is sourced
- We can also have a `~/.bash_logout` file that will be sourced when we exit our shell



Unix 202

Shell Scripts

Shell Scripts

What is a Shell Script and what do we use them for?

- Shell scripts are files containing instructions we want the shell to perform
- Shell scripts are useful for re-using work that you might need to do repetitively
- They provide a quick and easy method for sharing your useful tools with your colleagues
- We can also program a script to do a task that we want performed at regular intervals using “cronjobs”

The Anatomy of a Shell Script

- The “Shebang”
  - Very first line of the script and is used to denote the program that should interpret the file
  - Looks like: `#!/bin/bash` or `#!/bin/python`
- Lines starting with a ‘#’ character are not interpreted and are called comment lines
  - We use comment lines to explain in non-programmatic language what is going on
- Other lines will be interpreted as code for the interpreter given in the “shebang”
- A script must also have executable permissions set
  - A command for changing a file’s mode: `chmod`
  - Usage: `chmod [OPTION] MODE[,MODE] FILE`
  - Example: `chmod u+x,g+x myscript.sh`

Shell Scripts

How do we run a shell script and what happens when we do?

- There are two ways to execute a shell script

1. `./example.sh`
   - Searches my current working directory for the script `example.sh` and execute it
2. `example.sh`
   - Searches for a file called `example.sh` in each of the directories in my `$PATH` and executes the first one it finds

- When we execute a script, a child shell is created in order to run that script
  - The child shell is replaced by the program specified in the "shebang" and the name of script is given as an argument to that program
  - This implies the program does not need to be a shell

Passing Arguments to Scripts

```
$ ./example.sh this is an argument
```

- When possible, it can be more efficient to read input that comes with the script than waiting for input during execution
- There are special variables which are part of the built-in variables for dealing with arguments
  - `$#`: The number of arguments passed
  - `$@`: The arguments that were passed
  - `$1`: The first argument
  - `${n}`: The n^{th} argument. When n is multiple digits the curly braces are mandatory!
- This is how programs like `ls` know to handle the options you give them!

Built-In Variable	Value
`$#`	`4`
`$@`	`this is an argument`
`$0`	`./example.sh`
`$1`	`this`
`$2`	`is`
`$3`	`an`
`$4`	`argument`

Utilizing Subshells

- We can break up multiple commands that are meant to run together into their own subshells
  - Special built-in BASH variable: `$BASH_SUBSHELL`
    - Keeps track of the number of subshells we are “deep”
- Subshells inherit a copy of the parent’s variables, but modifying their copies does not affect the parents.
- We usually use subshells for one of two things
  - Command substitution
  - Creating subtasks within a script

Command Substitution

- A lot of times we may want to save the output of a command into a variable
  - We can do this by spawning a subshell to perform that command, and substituting that output somewhere
- Syntax: `$(command)`
  - Means execute `command` and replace `$(command)` with its output
  - An alternative syntax: `` `command` ``

```
$ date
Fri Feb 10 01:24:23 EST 2023
$ echo "The current date and time is: $(date)”
The current date and time is: Fri Feb 10 01:30:38 EST 2023
```

Parallelization Within Scripts using Subshells

- We can also run code in a subshell without substituting its output as in command substitution!
  - Syntax: `(command)`
- Just like with background processes, we can run multiple lines of code at the same time
  - We can use the exact same ampersand (&) syntax!
- When we parallelize our code, there is no guarantee that the individual tasks will complete in order
  - This can lead to some strange behavior
- A built-in for synchronization: `wait`
  - Pauses execution until the previous task finishes

Unix 202

Scheduling Scripts

Scheduling Scripts

Dealing with the `crond` daemon

- The `crond` daemon checks every minute for scheduled scripts
  - A daemon is a system process that is always running
  - We can interact with the daemon by leaving it instructions within a file
- A command for scheduling scripts: `crontab`
  - `Usage: crontab [-u user] [-l | -r | -e] [-i] [-s]`
    - If you don't specify a user, you will by default open your own
    - Typically, you will use: `crontab -e`
- What happens when you use this command?
  - Opens your cron job table to edit/add scheduled tasks

Interacting with your crontab

- If you've never interacted with crontab before, it will create a new one for you
- It expects each line to be formatted as:
  - `m h dom mon dow command`
    - `m=minute; 1-60`
    - `h=hour; 0-23`
    - `dom=day of the month; 1-31`
    - `mon=month; 1-12`
    - `dow=day of the week; 0-6 (Sunday-Saturday)`
    - `command=script or command to run`
- You can use `*` to wildcard each column
- Example
  - `0 0 1 * * command # Run at midnight on the first of every month`

Unix 202

Conclusions

THANK YOU

Feel free to reach out to rderue@purdue.edu with questions.

Slides are posted at:
https://www.rcac.purdue.edu/training/unix201