Week 3

Announcements

- Basic 1 is "due" today
  - Actually due on the 24th
- Advanced 1, Basic 2, and Advanced 2 are out
- Lecture 2 survey is closing today

Unix and You

Lecture 3

Where I try not to turn this into an OS lecture

Overview

1. What is Unix?
2. How does Unix work?
3. Interacting with Unix via Shells (feat. Bash)

What is Unix? (review)

- Family of operating systems derived from the original AT&T Unix from the '70s
  - Fun fact: C was developed for use with the original Unix
- Emphasis on small programs and scripts composed into bigger and bigger systems
- Preference for plain-text files for configuration and data
- Spawned many derivatives and clones: BSD, Solaris, AIX, mac OS, Linux
- Became so prevalent in industry and academia that it's been immortalized as a set of standards: **POSIX** (IEEE 1003)
- From here on out, whenever I say or write "Unix" and "*nix" I'm referring to (mostly) POSIX-compliant systems
  - mac OS is POSIX-certified, while Linux is not

What does POSIX mean for us?

- We get a neat set of standards!
- As long as you follow the standards (and avoid any implementation-specific behavior), your scripts/code should work on other POSIX systems

Examples of POSIX standard things

- C POSIX API: headers like `unistd.h`, `fcntl.h`, `pthread.h`, `sys/types.h`
- Command line interface and utilities: `cd`, `ls`, `mkdir`, `grep`
  - Commands in the specification
  - Sort by "Status"; "Mandatory" ones are the useful ones to look at
- File paths/names
- Directory structure
- Environment variables: `USER`, `HOME`, `PATH`

Unix philosophy

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

- Peter Salus, *A Quarter-Century of Unix* (1994)

How does Unix work?

We're starting from the ground up >:)

Components

Kernel

- Software that serves as the intermediary between hardware resources and user applications
  - Manages hardwares access
- Handles things like multi-tasking, security enforcement, file systems, device drivers, launching programs, and more
- Present a stable application programming interface (API) for user programs to use in the form of *system calls*

Applications

- Software that users run and interact with
- Includes things like Bash, nano, VS Code, Gnome Desktop, `ls` etc.

Hand in hand, these form an overall operating system

Unix design

- Effectively boils down to processes interacting with files
    - Program: list of instructions to execute
    - Process: a running instance of a program
- **Files serve as a sort of universal interface**
    - Processes pass data to each other via a read/write interface

Unix processes

- Identified by a process ID (PID)
- Associated with a user
- Has a current working directory
- Has an associated program *image*: the actual CPU instructions to run
- Has memory containing the image and program data like variables

Unix processes

- File descriptor table
  - Handles to various resources that have a file interface (read/write/**seek**)
  - *File descriptors* are indexes into this table
  - 0: Standard input (`stdin`, `cin`)
  - 1: Standard output (`stdout`, `cout`)
  - 2: Standard error (`stderr`, `cerr`)
  - POSIX functions for handling these: `open()`, `close()`, `read()` etc.
  - Don't confuse them with C stdio functions: `fopen()`, `fclose()`, `fread()` etc. (these are often an abstraction for the POSIX functions)
- Environment variables
  - Provide information about the process's environment
  - `PATH`: directories to find executables in
  - `PWD`: current working directory
  - `USER`: user
  - `HOME`: user's home directory
  - ...and more

Signals

- A way to communicate with processes
  - `man 7 signal`
  - `kill` (ignore the name) can signal processes
  - ^C (Ctrl-C) at a terminal sends SIGINT (interrupt)
  - ^Z sends SIGTSTP (terminal stop)
- Programs can implement handlers for custom behavior
  - SIGKILL and SIGSTOP can't be handled

Process creation

- A process calls `fork()` to make a copy of itself
  - The process is called the "parent" and the copy is called the "child"
  - The child is a **perfect copy** of the parent, except for the `fork()` return value
  - This includes program variables, program arguments, environment variables*, etc.
- The child can call `exec()` functions to load a new program
  - `man 3 exec`
  - This wipes the process's memory for the new program's data
  - Environment variables and file descriptor table is left the same
  - Cool, I'll have it run `execvp("ls", args)` to list the current directory!
  - But what if we parameterized the executable to run?
  - We have the beginnings of a shell...

Unix files

- In Unix, everything is a file
  - Data living on a disk? That's a file
  - Directories? Those are special kinds of files
  - Your instance of `vim`? That can be represented by a bunch of files!
- Unix files represent a *stream of bytes* that you can read from or write to
  - Serves as a neat interface
  - `stdin` and `stdout` are seen as files by your program
  - **What if we tie the output of one process to the input of another?**
- Files have various properties
  - You can check them with `ls -l`
  - `r`: read
  - `w`: write
  - `x`: execute
  - These three are often grouped together to form an octal digit (gasp! octal!)
  - User owner, group owner
  - `chmod` and `chown` can modify these

(Generic) Unix directory structure

Some normal ones

- `/`: root, the beginning of all things
- `/bin`: binaries
- `/lib`: libraries
- `/etc`: configuration files
- `/var`: "variable" files, logs and other files that change over time
- `/home`: user home directories

Everything is a file

- `/dev`: device files
- `/proc`: files that represent runtime OS information

Putting them together

- It's just processes interacting with other processes and files
- Processes create more processes (yes there is a primordial process)
- What if we hooked up processes end to end, `stdin` to `stdout`?
  - We can form a "pipeline" of data processing
- What if we tied the output of a process to a file instead of a terminal?

This is the job of a shell

Interacting with Unix via Shells

feat. Bash

```
:(){ :|:& };:
```

Do **NOT** run this

Job control

- We're familiar with just launching a process
  - `$ echo "hello world"`
- There's other things we can do, like launch it in the background with `&`
  - `$ echo "hello world" &`
- ^C (SIGINT) can cause most process to stop
- ^Z (SIGTSTP) can cause most processes to suspend
- `jobs` can list out processes (jobs table) that the shell is managing
- `bg` can background a process, yielding the terminal back to the shell
- `fg` can foreground a process, giving it active control of the terminal
  - `bg` and `fg` can index off of the jobs table
- `disown` can have the shell give up ownership of a process
- The `?` variable holds the exit status of the last command
  - 0 means success/true
  - Not 0 means failure/false

Stringing together commands

- `cmd1 && cmd2` (and)
  - Run `cmd2` if `cmd1` succeeded
- `cmd1 || cmd2` (or)
  - Run `cmd2` if `cmd1` failed
- `cmd1 ; cmd2`
  - Run `cmd2` after `cmd1`
- `cmd1 | cmd2`
  - Connect output of `cmd1` to input of `cmd2`
  - `cmd1`'s fd 1 -> `cmd2`'s fd 0
  - `$ echo "hello" | rev`

File redirection

- `<`: set file as standard input (fd 0)
  - `$ cmd1 < read.txt`
- `>`: set file as standard output, overwrite (fd 1)
  - `$ cmd1 > somefile.txt`
- `>>`: set file as standard output, append (fd 1)
  - `$ cmd1 >> somelog.txt`

General form (brackets mean optional)

- `[n]<`: set file as an input for fd *n* (fd 0 if unspecified)
  - "input" means that the process can `read()` from this fd
- `[n]>`: set file as an output for fd *n* (fd 1 if unspecified)
  - "output" means that the process can `write()` to this fd
  - `2>`: capture `stderr` to a file
- `[n]>>`: set file as an output for fd *n*, append mode (fd 1 if unspecified)

Advanced Bash file redirection

- `&>`: set file as fd 1 and fd 2, overwrite (`stdout` and `stderr` go to same file)
- `&>>`: set file as fd 1 and fd 2, append (`stdout` and `stderr` go to same file)
- `[n]<>`: set file as input and output on fd *n* (fd 0 if unspecified)
- `[n]<&digit[-]`: copies fd *digit* to fd *n* (0 if unspecified) for input; `-` closes *digit*
- `[n]>&digit[-]`: copies fd *digit* to fd *n* (1 if unspecified) for output; `-` closes *digit*
- `<<`: "Here document"; given a delimiter, enter data as standard input

```
$ cat << SOME_DELIM
> here are some words
> some more words
> SOME_DELIM
```

- `<<<`: "Here string"; provide string directly as standard input

```
$ cat <<< "here's a string!"
```

- Both Here documents and strings will expand variables (coming up)

Diving into Bash

- Side note: `bash` != `sh`
- `bash` has a feature superset over `sh` (kinda like a `vim`/`vi` relationship)
    - Again, confounded by some systems linking/aliasing `sh` to `bash`
- While this is about Bash, many other shells have the same syntax
- The horse's mouth: [GNU Bash manual]
    - If you like the nitty gritty details it's a great read
    - These slides summarize major features of Bash

What Bash does

- Receive a command from a file or terminal input
- Splits it into tokens separated by **white-space**
    - Takes into account *"quoting"* rules
- Expands/substitutes special tokens
- Perform file redirections (and making sure they don't end up as command args)
- Execute command

Finding programs to execute

- If the command has a `/` in it, it's treated as a filepath and the file will tried to be executed
  - `$ somedir/somescript`
  - `$ ./somescript`
- If the command doesn't have a `/`, `PATH` will be searched for a corresponding binary
  - `$ vim` -> searches `PATH` and finds it at `/usr/bin/vim`
  - This is why you have to specify `./` to run something in your current directory

Shell built-ins

- Some commands are "built-in"/implemented by the shell
  - These will take precedent over ones in the `PATH`
- Some other commands don't make sense outside of a shell
  - Think about why `cd` is a built-in and not a separate utility
  - (hint: `fork()` and `exec()`)

Shell and environment variables

- Shell variables stored inside the shell *process*
  - They're handled by the Bash program itself, stored as program data in the process's memory
  - Launched commands don't inherit them (what does `exec()` do?)
- Set them with `varname=varvalue`
  - **Meaningful whitespace!**
  - `varname = varvalue` is interpreted as "run `varname` with arguments `=` and `varvalue`"
- You can set *environment* variables with `export`
  - `export varname=varvalue`

Command grouping

- We discussed before that we can string commands together with `;`, `&&`, `||`
- We can also group commands together as a unit, with redirects staying local to them:
- `(commands)`: performs *commands* in a "subshell" (another shell *process*/*instance*; what does this mean for *shell* variables?)
- `{ commands; }`: performs *commands* in the calling shell instance
    - **Note**: There has to be spaces around the brackets and a semicolon (or newline or `&`) terminating the *commands*

Expansion and substitution

Bash has special characters that will indicate that it should *expand* or *substitute* to something in a command

Variable expansion

- `$varname` will expand to the value of `varname`
- `${varname}`: you can use curly brackets to explicitly draw the boundaries on the variable name
  - `$ echo ${varname}somestring` vs `$ echo $varnamesomestring`
- **Note**: expansions/substitutions will be further split into individual tokens by their white-space

Command substitution (via subshell)

- `$(command)` will substitute the output of a *command* in the brackets
  - `$(echo hello | rev)` will be substituted with "olleh"

Process substitution

- `<(command)` will substitute the *command* output as a filepath, with the output of *command* being **readable**
- `>(command)` will substitute the *command* input as a filepath, with the input of *command* being **writeable**
- `$ diff <(echo hello) <(echo olleh | rev)`
  - `diff` takes in two file names, but we're replacing them with command outputs

Arithmetic expansion

- `$((expr))` will expand to an evaluated arithmetic expression *expr*

But wait...

- What if I actually wanted to **not** expand a variable and keep the `$`?
- What if I didn't want a variable to be split by white-space?
- What if I'm lazy and don't want to escape spaces?

Quoting

- Allows you to retain certain characters without Bash expanding them and keep them one string
  - Common use case is to preserve spaces e.g. for filepaths that have spaces in them (spaces delimit tokens in a command)
- Single quotes (`'`) preserves **all** of the characters between them
  - `$ echo '$HOME'` will output `$HOME`
- Double quotes (`"`) preserve all characters except: `$`, `\`, and backtick
  - `$ ls "$HOME/Evil Directory With Spaces"` will list the contents of a directory `/home/jdoe/Evil Directory With Spaces`
  - **Variables expanded inside of double quotes retain their white-space**
  - (without this, that path would've had to have been `$HOME/Evil\ Directory\ With\ Spaces`, using `\` to escape the space characters)
- Note that when quoting, the quotes don't appear in the program's argument
  - `$ someutil 'imastring'`: `someutil`'s argv[1] will be `imastring`

Control flow

`if-elif-else`

```
# '#' comments out the rest of the line
# brackets indicate optional parts
if test-commands; then
  commands
[elif more-test-commands; then
  more-commands]
[else
  alt-commands]
fi
```

- *test-commands* is executed and its **exit status** is used as the condition
  - ***0*** = success = "true", everything else is "false"
- You can put the `if-elif-else` structure on one line!
  - This applies to the upcoming control flow structures as well

Commands for conditionals

You can use any commands for conditions, but these constructs should be familiar:

- `test expr`: `test` command
  - Shorthand: `[ expr ]` (remember your spaces! `[` is technically a utility name)
  - `test $a -eq $b`
  - `[ $a -eq $b ]`
- `[[ expr ]]`: Bash conditional
  - Richer set of operators: `==`, `=`, `!=`, `<`, `>`, among others
  - **Note**: The symbol operators above operate on strings, thus `<` and `>` operators do lexicographic (i.e. dictionary) comparison; "100" is lexicographically less than "2" since for the first characters "1" comes before "2"
  - Use specific arithmetic binary operators (*a la* `test`) if you intend on comparing numeric values
  - `[[ $a == $b ]]`
  - `[[ $a < $b ]]`: this would evaluate to "true" if a=100, b=2
- `(( expr ))`: Bash arithmetic conditional
  - Evaluates as an arithmetic expression
  - `(( $a < $b ))`: this would evaluate to "false" if a=100, b=2

`while`

```
while test-commands; do
  commands
done
```

- Similarly to `if`, the exit status of *test-commands* is used as the conditional
- Repeats *commands* until the condition **fails**

`until`

```
until test-commands; do
  commands
done
```

- Repeats *commands* until the condition **succeeds**

`for`

```
for var in list; do
  commands
done
```

- *list* will be **expanded** and on each iteration *var* will be set to each member of the list
- **Note**: if there is no `in list`, it will implicitly iterate over the argument list (i.e. `$@`)

Functions

```
func-name () compound-command
# or
function func-name [()] compound-command # [] for optional parens
```

- A **compound command** is a **command group** (`( )`, `{}`) or a control flow element (`if-elif-else`, `for`)
- Called by invoking them like any other utility, including passing arguments
  - Arguments can be accessed via `$n`, where *n* is the argument number
  - `$@`: list of arguments
  - `$#`: number of arguments

Examples

```
hello-world ()
{
  if echo "Hello world!"; then
    echo "This should print"
  fi
}
# calling
hello-world
```

```
function touch-dir for x in $(ls); do touch $x; done
# calling
touch-dir
```

```
echo-args ()
{
  for x in $@; do
    echo $x
  done
}
# calling
echo-args a b c d e f g
```

```
divide ()
{
  if (( $2 == 0 )); then
    echo "Error: divide by zero" 1>&2
    # the redirection copies stderr to stdout
    # so when echo outputs to its stdout, it's
    # really going to stderr
  else
    echo $(($1 / $2))
  fi
}
# calling
divide 10 2
divide 10 0
```

Scripts

- As was mentioned a few weeks ago, it's annoying to have to type things/go to the history to repeatedly run some commands
- Scripts are just plain-text files with commands in them
- **There's no special syntax for scripts: if you can enter the commands in them line by line at the terminal it would work**
- You can treat it as a simple programming language
- You can specify the interpreter program with a "shebang" on the first line
  - `#!/bin/bash`
  - `#!/bin/zsh`
  - `#!/usr/bin/env python`
- Arguments work like that of functions:
  - `$n` **Note**: $0 will refer to the script's name, as per *nix program argument convention
  - `$@`
  - `$#`

Running vs sourcing

- *Running* (executing) a script puts it into its own shell instance; shell variables set *won't* be visible to the parent shell
  - `./script.sh`
  - `bash script.sh`
- *Sourcing* a script makes your *current* shell instance run each command in it; shell variables set *will* be visible
  - `source script.sh`
  - `. script.sh`
- Think about the nuance here
  - Behavior of `cd` when running a script vs sourcing a script?

Running vs sourcing

- Say your shell is currently at `/home/bob`
- There's a script called `go-places` with the following contents:

```
cd /var/log
```

- Q1: Where would your current shell be if you ran `$ bash go-places`?
- Q2: Where would your current shell be if you ran `$ source go-places`?

Running vs sourcing

- Say your shell is currently at `/home/bob`
- There's a script called `go-places` with the following contents:

```
cd /var/log
```

- Q1: Where would your current shell be if you ran `$ bash go-places`?
  - A: `/home/bob`
  - This will create a new Bash instance, which will then perform the `cd`.
  - This will result in the current shell staying in the current directory, as it never ran `cd` in the first place
- Q2: Where would your current shell be if you ran `$ source go-places`?
  - A: `/var/log`
  - This will cause the current shell to read in and execute the `cd`
  - This will result in the current shell changing directories

Any other questions?