Unix and You

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?

- 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 pretty 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 hardware resources and access to them
- 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*

Components

Libraries

- Reusable pre-written software you can call upon
- Provide functionality that would be a pain to write every time (e.g. graphics)

Applications

- Software that users run and interact with or assist in the background
- 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
- 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

Unix processes

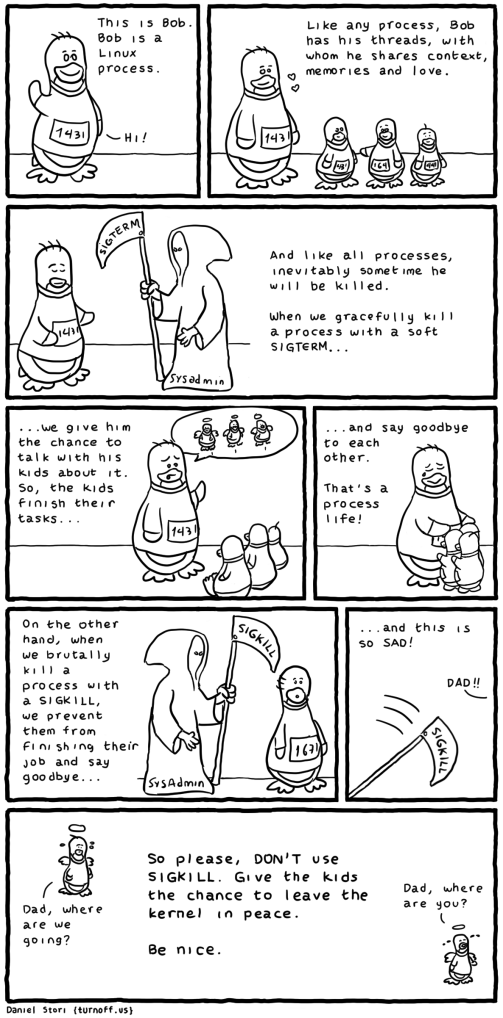
- File descriptor table
  - Handles to various resources that have a file interface (read/write/*seek*)
  - *File descriptors* are indexes into this table
  - 0: Standard input (`stdin`, `cin`)
  - 1: Standard output (`stdout`, `cout`)
  - 2: Standard error (`stderr`, `cerr`)
  - POSIX functions for 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)

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

signals - turnoff.us

the real reason not to use sigkill - turnoff.us



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, which is 0 for the child
  - 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
  - Effectively, "loads" a new program
  - 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: writing to a terminal is no different than writing to a file
  - `stdin` and `stdout` are seen as files by your program
  - **What if we tie the output of one process to the input of another?**
- As a side note: file name extensions have no intrinsic bearing on the data
  - A file can contain JPEG data but be named anything it wants
  - Most file formats have "magic numbers" in the first few bytes to help identify the file format
  - File name extensions are mostly meant to help you organize and recognize files

Unix files

- Files have various properties
  - You can check them with `ls -l`
  - `stat` can give more detailed information
  - `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

File mode bits

```
user  group  other
rwx   rwx    rwx
110   100    100
 6     4      4
```

```
chmod 644 some-file
chmod +x  execute-for-all
chmod u+x execute-for-only-me
chmod 777 chaos
chmod 600 secure-me
```

Links

- A special kind of file is a "symbolic" or "soft" link
  - This file simply contains a file path to another file, kind of like a shortcut
  - This can be useful if you want to have one file for public use to hide away underlying details
  - e.g. `python3` can be symbolically linked to a concrete file named `python3.10`
  - The utility `ln` is able to create symbolic links with the `-s` flag
- There are "hard" links as well
  - A hard link is really a file path to some underlying data
  - A file starts off with one hard link: the name/path it was given
  - You can create additional hard links that refer to the same data
- A hard link is directly linked to the underlying file data
- A soft link says "go check this other file path for the data"
  - A soft link can be given a bogus (broken) path

(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 data file instead of a terminal?
- If only we had a program we could interact with that could do these things for us

This is the job of a shell

What is a shell?

- Simply a program like any other that receives input and produces output
- This one just happens to interpret its input as "create more processes"
- They're not magical: you can write one too!

History of Unix shells

- The history of Unix shells is a bit messy
- Bunch of various shells around in the beginning that built off each other
  - There was no real "standard" set of syntax yet: these served as the foundations
  - Confusingly, they all used the name `sh`, so we call them by their creators
  - e.g. Thompson shell, Mashey shell, Bourne shell
- The Bourne shell (`sh`) ended up becoming a defacto gold standard
- POSIX standard shell (`sh`) takes a lot from the Bourne shell

There's lots of shells around

- Nowadays, when we say `sh` or shell syntax we generally refer to the POSIX standard or something minimalist along the lines of the Bourne shell
- The `sh` shell on your system probably isn't the actual Bourne shell: it's probably symbolically linked with another shell that is backwards compatible
  - Check it out: `ls -l /bin/sh`
  - A common one on Linux is `dash` (Debian Almquist shell)

There's lots of shells around

- There are many shells around that build off of the Bourne shell (and POSIX shell) and add more features
  - Not each shell supports each other shells' special non-standard features
  - If you write a script meant for `sh`, don't throw in a non-standard feature: check the shebang!
- Bash (`bash`) is the "Bourne again shell": adds a bunch of features
- Z shell (`zsh`) is another popular feature that adds some features that originated in other shells (like Bash)
- Picking a shell to use is a personal choice: play around with them!

Basic shell command structure

```
<command> <argument 1> <argument 2> <argument 3>
    ^    ^     ^
    |    |     |-- programs are provided these to
    |    |         interpret (remember argc and argv[]?)
    |    |
    |    |-- words separated by whitespace
    |
    |-- certain things are actual programs, certain things
        are handled by the shell ("built-ins")
```

General shell operation

1. Receive a command from a file or terminal input
   - `ls -l $HOME > some_file`
2. Splits it into tokens separated by **white-space**
   - Takes into account *"quoting"* rules
   - `ls`, `-l`, **`$HOME`**, `>`, `some_file`
3. Expands/substitutes special tokens
   - `ls`, `-l`, **`/home/brandon`**, `>`, `some_file`
4. Perform file redirections (and making sure they don't end up as command args)
   - `ls`, `-l`, `/home/brandon`; (set standard output to **`some_file`**)
5. Execute command (remember our friend `exec()`?)
   - `argc` = 3
   - `argv` = [`"ls"`, `"-l"`, `"/home/brandon"`]
   - Standard output redirected to `some_file`
   - First "normal" token is the command/utility to run

Stringing together commands

- `cmd1 && cmd2`
  - Run `cmd2` if `cmd1` succeeded
  - Like a short-circuiting AND in other languages
- `cmd1 || cmd2`
  - Run `cmd2` if `cmd1` failed
  - Like a short-circuiting OR in other languages
- `cmd1 ; cmd2`
  - Run `cmd2` after `cmd1`
- `cmd1 | cmd2`
  - Connect standard 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`
  - Creates file if it doesn't exist already
- `>>`: set file as standard output, append (fd 1)
  - `$ cmd1 >> somelog.txt`
  - Creates file if it doesn't exist already

File redirection

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)

Exercises

1. Write a command that saves the output of `ls` to a file `listing`
2. The command `rev` reverses a line of text and `sort` takes lines of input and outputs them in a sorted manner
   - Write a command that takes the output of `ls`, reverses the name of each file, sorts these reversed names and saves it to a file called `gnitsil`
3. Write a command that runs `git status` and saves the standard output to `out.txt` and standard error to `err.txt`
4. The command `date` outputs a timestamp
   - Write a command that appends the current timestamp to a file called `timestamps.log`

Shell and environment variables

- Shell variables are stored inside the shell *process*
  - They're handled by the shell 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`
  - `export existing_variable`
  - Sets a variable to be **exported** to new processes
- You can also set environment variables just for one command
  - `$ hello=world some-command-here`

Using variables

- You can use a variable with `$varname` or `${varname}`
  - e.g. `echo $PATH`, `echo $HOME`
- What the shell does is **"expand"** this to its value
  - Look back at the step-by-step of what a shell does
  - Before `echo` is run, `echo $HOME` becomes `echo /home/brandon`
  - Simple text substitution
  - Play with concatenation: `somevar=$HOME/hello`
- This isn't the only **"expansion"**
  - This is a very important topic for next week

Finding programs to execute

- If the command has a `/` in it, it's treated as a filepath and the file will be executed
  - `$ somedir/somescript`
  - `$ ./somescript`
  - Only works if the file has its execute bit set
- 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()`)

What even is an executable, anyway?

There are two classes of executable program

- Binaries
  - These are files that contain instructions that the computer understands natively at a hardware level (machine code)
  - You get these when you tell GCC or Clang to compile your C or C++ program
  - Various kinds of formats: ELF, Mach-O, PE, etc.
  - The first few bytes of these files usually have some special byte sequence to identify the file type
- Interpreted programs/scripts
  - These are plain-text files that contain human readable text that map to some programming language
  - These files are run through another program called an "interpreter" to do tasks specified in the program
  - Python scripts are typically run through a Python interpreter
  - Shell scripts are run through a shell

What even is an executable, anyway?

- The first line of a script *should* contain a **shebang**
  - This tells the OS what program to use as an interpreter
  - Starts with `#!` with the path to the interpreting program right after
  - `#!/bin/sh`: "Run this script with `sh`"
  - `#!/bin/bash`: "Run this script with Bash"
  - `#!/usr/bin/env python3`: "Run this script with whatever `env` finds as `python3`"
  - If there is no shebang specified, the OS usually assumes `sh`

Shell scripts

- 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 enter the commands in them line by line at the terminal it would work**
- Generally good practice to specify a shebang
  - It's usually a good idea to go with `sh` for universal compatibility
  - `bash` can also be a good choice due to ubiquity; just be aware it's not a standard
  - Don't mix up special Bash features in a script marked for `sh`!
- Arguments are presented as special variables
- `$n`: Argument *n*, where *n* is the number (e.g. `$1` is the 1st argument)
  - **Note**: `$0` will refer to the script's name, as per *nix program argument convention
- `$@`: List of all arguments
- `$#`: Number of arguments

Running scripts

- There's a nuance between `$ ./my-script` and `$ bash my-script`
- `$ ./my-script` tells the OS to execute the `my-script` file
  - The OS will try to identify the file and will look for a shebang for the interpreter
  - The OS will run the interpreter, feeding it `my-script`
- `$ bash my-script` tells the OS to execute `bash` with `my-script` as an argument
  - It's up to Bash to figure out what to do with `my-script`
  - In this case, Bash just reads the file and executes each line in it

Exercise

- Write a shell script that appends an ISO 8601 format timestamp, then appends the first argument to a file named `log`
  - `date -Isec` can get this timestamp for you
  - Make sure to give it a shebang
  - Make sure to `chmod` it so it's executable
  - Run it with an argument e.g. `$ ./myscript this-is-an-argument`

This was only a taste

- So far we only discussed shells and how they function in the context of the *nix systems
- Next week we'll talk about control flow, more advanced redirection, fancy shell features and more about the nitty gritty of shell syntax itself

Any other questions?