Unix and You

Week 2

Now in remark!

Overview

1. Introductions 2: Electric Boogaloo
2. Announcements
3. What is Unix?
4. How do I Unix?

GSI: Jiwon Joung

- CSE PhD student under Prof. Morley Mao
- BS, MS in CS at Harvard University
- Research interests are in systems, security, and autonomous vehicles
- Owner of the most amazing uniqname: computer@umich.edu
- Office hours:
    - Tuesday 12 pm - 4 pm
    - Friday 4 pm - 8 pm

Announcements

New website

- https://www.eecs.umich.edu/courses/eecs201
  - Will be updated regularly, will always have the latest information
  - Canvas files will be updated occasionally

New website

- https://www.eecs.umich.edu/courses/eecs201
  - Will be updated regularly, will always have the latest information
  - Canvas files will be updated occasionally

Grading policy update

- Attendance is now optional!
- Advanced exercise grading scheme changed
  - Each week will have a "set" of advanced exercises that you can complete
  - You don't have to complete all of the exercises in a set to get credit
  - You are expected to get at least 40 advanced exercise points
  - See website's Grading page for details

Assignments

- Homework 1 posted: due next Friday (Jan 24)
- Advanced 1 posted: due two Fridays from now (Jan 31)
    - I'm going to update this one later today to reflect the new grading policy
    - Also going to update it to have bit more hand-holding: didn't realize that Arch's Installation Guide has gotten a bit more confusing
    - If you already checked out, don't worry

What is Unix?

Where I try not to turn this into an OS lecture

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

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)

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

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` and the library that implements functions wrapping system calls
- Command line interface and utilities: `cd`, `ls`, `mkdir`, `grep`
- File paths/names
- Directory structure
- Environment variables: $USER, $HOME, $PATH

Unix and You

Probably what you actually want to get out of this lecture

Let's review some commands

- `ls`
- `pwd`
- `echo`
- `cat`
- `mkdir`
- `mv`
- `touch`
- `rm`
- `less`
- `man`

Q: How does a program start?

Q: How does a program start?

- On a POSIX system, your program calls `fork()` to clone itself to a new process, then `exec*(<program name>, ...)` on the new process to load a new program
  - These are functions that call operating system services
  - Cool! I'll write a program that calls `execvp("ls", ...)` to list my current directory!

Q: How does a program start?

- On a POSIX system, your program calls `fork()` to clone itself to a new process, then `exec*(<program name>, ...)` on the new process to load a new program
  - These are functions that call operating system services
  - Cool! I'll write a program that calls `execvp("ls", ...)` to list my current directory!
- What if we parameterized by taking in user input? Now we can `exec()` other things besides `ls`!

Q: How does a program start?

- On a POSIX system, your program calls `fork()` to clone itself to a new process, then `exec*(<program name>, ...)` on the new process to load a new program
    - These are functions that call operating system services
    - Cool! I'll write a program that calls `execvp("ls", ...)` to list my current directory!
- What if we parameterized by taking in user input? Now we can `exec()` other things besides `ls`!
- We now have the beginnings of a *shell*

Shells

- Provides a user interface to an operating system
- Launch new programs
- Handle input and output of the new programs
- Clean up after exited programs
    - ...or kill running programs

Shells

- Provides a user interface to an operating system
- Launch new programs
- Handle input and output of the new programs
- Clean up after exited programs
    - ...or kill running programs

Q: What shells have you used before?

Shells

- Provides a user interface to an operating system
- Launch new programs
- Handle input and output of the new programs
- Clean up after exited programs
    - ...or kill running programs

Q: What shells have you used before?

- `sh`
- `bash`
- `csh`
- `zsh`
- `fish`

Lets play around a bit with Bash

- Job control
- Signals
  - Ctrl-Z: SIGTSTP
  - Ctrl-C: SIGINT

Lets play around a bit with Bash

- Job control
- Signals
  - Ctrl-Z: SIGTSTP
  - Ctrl-C: SIGINT

Bash Cheatsheet

- `$?` exit status of previous command
- `cmd1 && cmd2`
  - run `cmd2` if `cmd1` succeeded
- `cmd1 || cmd2`
  - run `cmd2` if `cmd1` failed
- `cmd1; cmd2`
  - run `cmd2` after `cmd1`

What goes into a Unix system?

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's a bunch of files!

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's 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

- 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's 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
  - `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

Playing with files

- Standard files
  - `stdin` (file descriptor 0)
  - `stdout` (file descriptor 1)
  - `stderr` (file descriptor 2)
- (Basic) file redirection
  - `<` direct file to input
  - `>` direct output to file (overwrite)
  - `>>` append output to file
  - `|` pipe: tie output and input of two processes together
  - We'll look at more complex kinds later...

(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

Environment Variables

- These hold information about the environment that a process is executing in
    - Who is the user running this?: `$USER`
    - Where is the home directory?: `$HOME`
    - Where should `exec()` look for programs or scripts to run?: `$PATH`
- Processes launched by a shell will inherit the shell's environment variables

Environment Variables

- These hold information about the environment that a process is executing in
    - Who is the user running this?: `$USER`
    - Where is the home directory?: `$HOME`
    - Where should `exec()` look for programs or scripts to run?: `$PATH`
- Processes launched by a shell will inherit the shell's environment variables

How do I manipulate these?

- You can check them with `echo`
- Setting them depends on shell; for `bash` and some others:
    - `VARIABLE=value`
    - `export VARIABLE=value`

Environment Variables

- These hold information about the environment that a process is executing in
  - Who is the user running this?: `$USER`
  - Where is the home directory?: `$HOME`
  - Where should `exec()` look for programs or scripts to run?: `$PATH`
- Processes launched by a shell will inherit the shell's environment variables

How do I manipulate these?

- You can check them with `echo`
- Setting them depends on shell; for `bash` and some others:
  - `VARIABLE=value`
  - `export VARIABLE=value`
- It'd be nice if these variables stuck around...

Environment Variables

- These hold information about the environment that a process is executing in
  - Who is the user running this?: `$USER`
  - Where is the home directory?: `$HOME`
  - Where should `exec()` look for programs or scripts to run?: `$PATH`
- Processes launched by a shell will inherit the shell's environment variables

How do I manipulate these?

- You can check them with `echo`
- Setting them depends on shell; for `bash` and some others:
  - `VARIABLE=value`
  - `export VARIABLE=value`
- It'd be nice if these variables stuck around...
  - Set them in the shell's configuration file! e.g. `.bashrc`

Your programs can use them too!

```c
#include <stdio.h>
int main(int argc, char *argv[], char *envp[]) {
  for (int i = 0; envp[i] != NULL; ++i) {
    puts(envp[i]);
  }
  return 0;
}
```

Your programs can use them too!

```c
#include <stdio.h>
int main(int argc, char *argv[], char *envp[]) {
  for (int i = 0; envp[i] != NULL; ++i) {
    puts(envp[i]);
  }
  return 0;
}
```

Q: What sort of data structure is `envp`?

Writing scripts

- Scripts are really just collections of commands
- A *shebang* indicates what binary to feed the script to
  - `#!/usr/bin/bash`
  - `#!/usr/bin/python3`

Writing scripts

- Scripts are really just collections of commands
- A *shebang* indicates what binary to feed the script to
  - `#!/usr/bin/bash`
  - `#!/usr/bin/python3`
- Shell script syntax can be rather finicky and arcane
  - Meaningful whitespace? `VARIABLE=x` vs `VARIABLE = x`

Writing scripts

- Scripts are really just collections of commands
- A *shebang* indicates what binary to feed the script to
    - `#!/usr/bin/bash`
    - `#!/usr/bin/python3`
- Shell script syntax can be rather finicky and arcane
    - Meaningful whitespace? `VARIABLE=x` vs `VARIABLE = x`
- Shell scripts are better for smaller things
- Rule of thumb: More than 50-100 lines, more than a shell script
    - Unix philosophy: invoke other tools to help out!
    - Perhaps use another language e.g. Python

Any other questions?

If not, I'm just going to talk about computers, Linux, and whatever suits my fancy.