07 – Your shell, jobs, and proc

CS 2043: Unix Tools and Scripting, Spring 2019 [2]

Matthew Milano
February 6, 2019

Cornell University
Table of Contents

1. Processes Overview
2. Modifying Processes
3. Jobs
4. Customizing your Terminal
As always: Everybody! ssh to wash.cs.cornell.edu

- Quiz time! Everybody! run `quiz-02-06-19`
- You can just explain a concept from last class, doesn’t have to be a command this time.
Processes Overview
What is a Process?

• A process is just an instance of a running program.
• Not just a “program” - it is being executed.
• Not just a “running program”, as you can execute the same program multiple times.
  • These would be multiple processes running an instance of the same program.
• Example: if you open more than one terminal (windows or tabs), you are running multiple processes of your shell.
• You can execute `echo $$` to see the process of the current running shell.
Identification

- Processes have a unique “Process ID” (PID) when created.
- The PID allows you to distinguish between multiple instances of the same program.
- There are countless ways to discover the PID, as well as what processes are running.
- These methods often depend on how much information you want, as well as what your user privileges are.
Process Snapshot

ps [options]
- Reports a snapshot of the current running processes, including PIDs.
- By default, only the processes started by the user.
- Use -e to list every process currently running on the system.
- Use -ely to get more information than you can handle.
- Use -u <username> to list all processes for user username.
- Use -C <processname> to list all processes matching a name
- Use ps aux for “BSD” style ps, works on macOS/*nix
Resource Usage

Display and Update `top` CPU Processes

`top [flags]`

- Displays the amount of resources in percentages each process is using.
- Use `-d <seconds>` to control the update frequency.
  - The act of monitoring resources usage uses resources!
- Use `-u <user>` to show only the processes owned by `user`.
- Use `-p <PID>` to show only the statistics on process with id number `PID`.

- Can be a very powerful analysis tool.
Better Resource Usage

Display and Update **htop** CPU Processes

**htop [flags]**

- Displays the amount of resources in percentages each process is using.
- Use `-d <seconds>` to control the update frequency.
  - The act of monitoring resources usage uses resources!
- Use `-u <user>` to show only the processes owned by **user**.
- Use `-p <PID>` to show only the statistics on process with id number **PID**.

- Just a lot better than **top**, but not on all systems
- use F6 (the function key) to change sort order
Example: Resource Monitoring

- First, use `ps` to find the PID for `firefox`:
  
  ```
  $ ps -C firefox
  12975 ? 00:01:45 firefox
  ```

- Now that we have the PID of `firefox`, monitor using `htop`:
  
  ```
  $ htop -p 12795
  ```

- See `man htop` to understand what all is being reported.
- Some great top examples in [3].
Modifying Processes
Priority

- Suppose you want to run some long calculation that might take days, but would consume 100% of your CPU.
- Can we tell the server to give your process less priority in terms of CPU time?
- Recall that although Unix seems to run tens or hundreds of processes at once, one CPU can only run “one process” at a time.
- Quick switching back and forth between processes makes it seem as though they are all running simultaneously.
- In Unix, each process is given a priority when it starts.
  - This priority determines how frequently the process gets CPU time.
Initial Priority

Execute Process with Non-default Priority

\texttt{nice [options] command}

- Runs \texttt{command} with specified "niceness" value (default: \texttt{10}).
- \textit{Niceness} values range from \texttt{-20} (highest priority) to \texttt{19} (lowest priority).
- Only \texttt{root} can give a process a \textit{negative niceness} value.
- Commands run without \texttt{nice} have priority \texttt{0}.
- Example: \texttt{nice -n 10 deluge}
  - Prevent torrents from hogging the CPU.
  - … don’t pirate stuff folks
Change the Priority of a Running Process

renice <priority> -p <PID>

- Change niceness of process with id PID to <priority>.
- Remember: only root can assign negative values.
- You can only renice a process you started.
  - Of course, root can renice anything.

- renice 5 -p 10275
  - Set the niceness of the process with PID 10275 to 5.
  - Slightly lower than normal niceness (default: 0).

- renice 19 -u username
  - Set niceness of all processes owned by username to 19.
Kill or Signal a Process

kill [-signal] <PID>

- Sends the specified signal to the process with id PID.
- By default (no signal given), it terminates execution.

  - kill <PID> same as kill -15 <PID>
  - Signal 15 is SIGTERM (signal terminate).

Kill all Processes by Name

killall [-signal] <name>

- Kills processes by name.
- By default (no signal given), it terminates execution.

  - killall firefox same as kill -15 firefox
  - Signal 15 is SIGTERM (signal terminate).
## Useful Kill Signals

- Kill signals can be used by number or name.
- **TERM or 15**: terminates execution (default signal sent with `kill` and `killall`).
- **HUP or 1**: hang-up (restarts the program).
- **KILL or 9**: like bleach, can kill anything.
- Some examples:

  ```bash
  # Terminates process with PID 9009.
  $ kill 9009
  
  # REALLY kills the process with PID 3223.
  $ kill -9 3223
  
  # Restarts the process with PID 12221.
  # Particularly useful for servers / daemon processes.
  $ kill -HUP 12221
  
  # Remember `top` and `htop`? They can both `renice` and `kill`
  ```
Jobs
What are Jobs?

- A job is a process running *under the influence* of a job control facility.
- Job control is a built-in feature of most shells, allowing the user to pause and resume tasks.
- The user can also run them in the background.
- Not covered here: **crontab**. For future sys admins, read the article in [1].
Let’s use **ping** as an example.

**Send Request Packets to Network Host**

```plaintext
ping <server>
```

- Measure network response time (latency) to `<server>` and back.
- Sends short bursts to `<server>`, measures time until return.
- Example: `ping google.com`
  - Use `ctrl+c` to kill the process (**ping** runs until killed).

The **ping** command will keep running indefinitely until stopped.
Why we Need Job Control

- As long as `ping` runs, we lose control of our shell.
- This happens with many other applications:
  - Moving / copying large quantities of files.
  - Compiling source code.
  - Playing multimedia.
  - Scientific computing.
  - `cat` with no arguments
- We need ways to control this while still being able to continue to use our terminal!
Starting a Job in the Background

**Operator &**

`<command> [arguments] &`

- Runs the specified `command` as a background job.
- Unless told otherwise, will send output to the terminal!
- Example: `mplayer best_song_ever.flac &`

- If you already started the job, use `ctrl+z` to pause it.

**tee: split command output**

`tee <filename>`

- Redirects output to `<filename>` and still prints it
- good for logging within a pipestream!
## Sending a Job to the Background

### Discovering your jobs

**jobs**
- Prints the running, paused, or recently stopped jobs.
- Prints jobs with their **JOB IDs**.

### Background

**bg <JOB ID>**
- Resumes the job with id **JOB ID** in the **background**.
- Without **JOB ID**, resumes last job placed in background.

### Foreground

**fg <JOB ID>**
- Resumes the job with id **JOB ID** in the **foreground**.
- Without **JOB ID**, resumes last job placed in background.
Detaching Jobs

**No Hangup**

`nohup <command> [args]`

- *Background jobs* (started with `&`) end when terminal closed.
- `nohup` launches `command` so it will ignore `SIGHUP` signals.
- `nohup mplayer best_song_ever.flac >/dev/null 2>&1 &`

**Disown a job**

`disown [flags] jobspec`

- The `-h` flag prevents `jobspec` from `SIGHUP` killing it.
  - Use if you forgot to launch with `nohup`, for example.
- `j jobspec` is the job number (e.g., execute `jobs` to find it).
- E.g., if `mplayer` has `jobID 1`, then `disown -h %1`
The `/proc` filesystem

- Everything in Linux is represented by a file
  - this includes your processes

```
$ ls /proc | head -3
1
10
10377
```

- These are all running processes!
what’s in a process?

<table>
<thead>
<tr>
<th>attr</th>
<th>coredump_filter</th>
<th>gid_map</th>
<th>mountinfo</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>autogroup</td>
<td>cpuset</td>
<td>io</td>
<td>mounts</td>
<td>...</td>
</tr>
<tr>
<td>auxv</td>
<td>cwd</td>
<td>limits</td>
<td>mountstats</td>
<td>...</td>
</tr>
<tr>
<td>cgroup</td>
<td>environ</td>
<td>loginuid</td>
<td>net</td>
<td>...</td>
</tr>
<tr>
<td>clear_refs</td>
<td>exe</td>
<td>map_files</td>
<td>ns</td>
<td>...</td>
</tr>
<tr>
<td>cmdline</td>
<td>fd</td>
<td>maps</td>
<td>numa_maps</td>
<td>...</td>
</tr>
<tr>
<td>comm</td>
<td>fdinfo</td>
<td>mem</td>
<td>oom_adj</td>
<td>...</td>
</tr>
</tbody>
</table>

```bash
$ ls /proc/1
```
• `/proc/N/cwd` is the process’s working directory
  • you can CD into it!
• `/proc/N/exe` is the program
• `/proc/N/fd` contains open files
  • Fun trick: open a file with `less`, then remove it, then look in `/proc/N/fd`
• `/proc/mem` is the live process memory!
• `man proc` for a lot more information!
Customizing your Terminal
What is it and Why?

• You will spend a lot of time in your terminal.
• It’s worth spending a little time to configure it how you want.
• Customizations allow you to be
  1. More effective.
  2. Perform common operations more quickly.
  3. Make your terminal appear more comfortable for you.
  4. A super all-star-hacker-pro with l33t skillz.

• Think of it this way: it’s like buying a new house. Paint the walls, build a tool shed, meet your neighbors, throw some parties. Why buy it if you weren’t going to make it yours?
  • Why use the default terminal just because it came that way?
What are Dotfiles?

• “Dotfiles” change, add, or enhance existing functionality.
  • The files reside in your home (~) directory.
  • They are hidden files: their names start with a .

• Some common dotfiles you’ll hear about:

<table>
<thead>
<tr>
<th>File</th>
<th>Controls</th>
</tr>
</thead>
<tbody>
<tr>
<td>~/.bashrc</td>
<td>bash terminal behavior*</td>
</tr>
<tr>
<td>~/.bash_profile</td>
<td>bash environment variables*</td>
</tr>
<tr>
<td>~/.profile</td>
<td>shell environment variables*</td>
</tr>
<tr>
<td>~/.vimrc</td>
<td>Controls the behavior of vim</td>
</tr>
<tr>
<td>~/.emacs</td>
<td>Controls the behavior of emacs</td>
</tr>
<tr>
<td>~/.gitconfig</td>
<td>Controls the behavior of git</td>
</tr>
<tr>
<td>~/.tmux.conf</td>
<td>Controls the behavior of tmux (covered later)</td>
</tr>
</tbody>
</table>

• There are many possible dotfiles to customize.
• We will focus on configuring our shell (bash).

* What these do depends on what you write in them! See lecture demo.
A Reminder: common environment variables

- **$PATH**: where your shell looks to find programs
- **$EDITOR**: your preferred editor (defaults to nano)
- **$LANG**: your language and file encoding
- **$LD_LIBRARY_PATH**: where your dynamic libraries are (not always set)
- **$USER**: who you are
- **$HOME**: your home directory
- **$TERM**: how fancy your terminal can be
- **$MANPATH**: places to find man pages
So we now know a little bit about how a script is structured.
It just executes from the top to the bottom.
The shebang says how to run it. But...

**Execute source in Current Shell**

```bash
source <filename> [arguments]
```

- *Executing* script B from script A runs B in a *subshell*.
- *Sourcing* script B from script A executes in *current shell*.
  - If script B *exits*, then script A *exits*!
- Think of it like copy-pasting B into A at the line where `source B` is written in A.
- Just like `#include <header.h>` in C if you know it.
- Fundamental to the initial shell setup process:
  - All dotfiles related to your *shell* are *sourced*. 
What Happens When

- There is a **lot** going on with dotfiles; no “standard” protocol.
- What happens when depends on:
  1. Your operating system.
  2. The shell you are using.
  3. For graphical logins, what your desktop / window manager is.
- There is an important difference between types of shells:
  - There is a “login” shell, and a “interactive” shell.
  - “Login” shell: takes place *once*, when you login.
    - `.profile`, `.bash_profile`, `.zprofile`, depending on what your shell is.
  - “Interactive” shell: takes *every time* you spawn a new shell.
    - E.g. **ctrl+shift+n** on Linux, **cmd+n** on Mac.
    - Inherits all actions that took place at *login*.
    - `.bashrc`, `.zshrc` depending on what your shell is.
There is even still an important distinction:

- A graphical login (logging in through the GUI).
- A login shell (disabled GUI, or used `ssh` or something).

Graphical logins:

- I will not cover this. There is way too much going on.
- Depends on what your GUI (Gnome, KDE, etc) is.
- A fantastic explanation in [4].
  - Hey! Look around the rest of the site!
  - Lots of other great information available!!!

Login shells:

- For simplicity, assume that when you login through your GUI, it triggers a login shell to be called.
- This is mostly true, but not exactly.
- Discussion to come: Bourne shells (`bash`, `ksh`, ...) vs `zsh`
  - Only because Bourne shells and `zsh` are “incompatible”.
• Where do the environment variables like $PATH come from?
• For Bourne Shells:
  1. System level configuration files are sourced. Same for all users.
     • The file /etc/profile is sourced.
     • Do NOT edit this file directly. It sources anything found in /etc/profile.d/*.sh. Put additional resources there.
     • This is where PATH among many other variables is getting set!
  2. User-level configuration files are sourced (if found).
     • bash looks for ~/.bash_profile first. If it sees it, it sources it.
     • Only if bash does not find ~/.bash_profile, it looks for ~/.bash_login next and then ~/.profile last.
     • ksh, on the other hand, only looks for ~/.profile.

• For zsh, the same pattern occurs:
  1. System level configuration: /etc/zprofile.
     • Typically, it emulates ksh and sources /etc/profile!
  2. Look for ~/.zprofile.
Know Your Shell

- `$SHELL` reports your default shell (`echo $SHELL`).
- How do I know what my shell looks for and in what order?
  - `man <shell>` and search for `INVOCATION` as well as `FILES`.
  - Or cruise the Arch Wiki – they’re great! E.g. Arch on `zsh`.

**Change your Login Shell**

```
chsh -s /absolute/path/to/new/shell username
```

- GNU and BSD `chsh` are slightly different, read the `man` page!
- Example usage to change `$SHELL` for `username`:
  ```
  $ sudo chsh -s /bin/zsh username
  ```
- Warning: do not change the `$SHELL` of the root user!
- Typically, `chsh` will modify `/etc/passwd`
  ```
  - grep your `username` and read last field.
  ```
