Lecture 11: Q&A

Recommendations on learning Operating Systems related topics

Some good resources to learn about this topic:

- MIT's 6.828 class - Graduate level class on Operating System Engineering. Class materials are publicly available.
- Modern Operating Systems (4th ed) - by Andrew S. Tanenbaum is a good overview of many of the mentioned concepts.
- The Design and Implementation of the FreeBSD Operating System - A good resource about the FreeBSD OS (note that this is not Linux).
- Other guides like Writing an OS in Rust where people implement a kernel step by step in various languages, mostly for teaching purposes.

Python versus a Bash scripts versus some other language

Bash scripts are useful for short and simple one-off scripts when you just want to run a specific series of commands.

- bash is easy to get right for a simple use case but it can be really hard to get right for all possible inputs.
- bash is not amenable to code reuse so it can be hard to reuse components of previous programs you have written.
- bash relies on many magic strings like `$?` or `$@` to refer to specific values.

Therefore, for larger and/or more complex scripts we recommend using more mature scripting languages like Python or Ruby.

Difference between `source run.sh` and `./run.sh`

For `source` the commands are executed in your current bash session and thus any changes made to the current environment, like changing directories or defining functions will **persist** in the current session once the `source` command finishes executing.

When running the script standalone like `./script.sh`, your current bash session starts a new instance of bash that will run the commands in `script.sh` Thus, if `script.sh` changes directories, the new bash instance will change directories but once it exits and returns control to the parent bash session, the parent session will remain in the same place.

Similarly, if `script.sh` defines a function that you want to access in your terminal, you need to `source` it for it to be defined in your current bash session.

Places where various packages and tools are stored

Regarding programs that you execute in your terminal, they are all found in the directories listed in your `PATH` environment variable.

- `/bin` - Essential command binaries

- `/sbin` - Essential system binaries, usually to be run by root
- `/dev` - Device files, special files that often are interfaces to hardware devices
- `/etc` - Host-specific system-wide configuration files
- `/home` - Home directories for users in the system
- `/lib` - Common libraries for system programs
- `/opt` - Optional application software
- `/sys` - Contains information and configuration for the system (covered in the first lecture)
- `/tmp` - Temporary files (also `/var/tmp`). Usually deleted between reboots.
- `/usr/` - Read only user data
  - `/usr/bin` - Non-essential command binaries
  - `/usr/sbin` - Non-essential system binaries, usually to be run by root
  - `/usr/local/bin` - Binaries for user compiled programs
- `/var` - Variable files like logs or caches

Difference between Docker and a Virtual Machine

Docker is based on a more general concept called containers. The main difference between containers and virtual machines is that virtual machines will execute an **entire OS stack**, including the kernel, even if the kernel is the same as the host machine. Unlike VMs, containers avoid running another instance of the kernel and instead share the kernel with the host.

Thus, containers have a **lower** overhead than a full VM. On the flip side, containers have a weaker isolation and only work if the host runs the **same** kernel.

Lastly, Docker is a specific implementation of containers and it is tailored for software deployment. Because of this, it has some quirks: for example, Docker containers will not persist any form of storage between reboots by default.

More Vim tips

- Plugins - Take your time and explore the plugin landscape. There are a lot of great plugins that address some of vim's shortcomings or add new functionality that composes well with existing vim workflows. For this, good resources are VimAwesome and other programmers' dotfiles.
- Marks - In vim, you can set a mark doing `m<X>` for some letter X. You can then go back to that mark doing `'<X>`. This let's you quickly navigate to specific locations within a file or even across files.
- Navigation - `Ctrl+O` and `Ctrl+I` move you backward and forward respectively through your recently visited locations.
- Undo Tree - Vim has a quite fancy mechanism for keeping track of changes. Unlike other editors, vim stores a tree of changes so even if you undo and then make a different change you can still go back to the original state by navigating the undo tree. Some plugins like gundo.vim and undotree expose this tree in a graphical way.
- Persistent undo is an amazing built-in feature of vim that is disabled by default. It persists undo history between vim invocations. By setting `undofile` and `undodir` in your `.vimrc`, vim will storage a per-file history of changes.