Q&A

For the last lecture, we answered questions that the students submitted:

Any recommendations on learning Operating Systems related topics like processes, virtual memory, interrupts, memory management, etc

First, it is unclear whether you actually need to be very familiar with all of these topics since they are very low level topics. They will matter as you start writing more low level code like implementing or modifying a kernel. Otherwise, most topics will not be relevant, with the exception of processes and signals that were briefly covered in other lectures.

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.

What are some of the tools you'd prioritize learning first?

Some topics worth prioritizing:

- Learning how to use your keyboard more and your mouse less. This can be through keyboard shortcuts, changing interfaces, &c.
- Learning your editor well. As a programmer most of your time is spent editing files so it really pays off to learn this skill well.
- Learning how to automate and/or simplify repetitive tasks in your workflow because the time savings will be enormous...
- Learning about version control tools like Git and how to use it in conjunction with GitHub to collaborate in modern software projects.

When do I use Python versus a Bash scripts versus some other language?

In general, bash scripts are useful for short and simple one-off scripts when you just want to run a specific series of commands. bash has a set of oddities that make it hard to work with for larger programs or scripts:

- bash is easy to get right for a simple use case but it can be really hard to get right for all possible inputs. For example, spaces in script arguments have led to countless bugs in bash scripts.
- bash is not amenable to code reuse so it can be hard to reuse components of previous programs you have written. More generally, there is no concept of software libraries in bash.
- bash relies on many magic strings like `$?` or `$@` to refer to specific values, whereas other languages refer to them explicitly, like `exitCode` or `sys.args` respectively.

Therefore, for larger and/or more complex scripts we recommend using more mature scripting languages like Python or Ruby. You can find online countless libraries that people have already written to solve common problems in these languages. If you find a library that implements the specific functionality you care about in some language, usually the best thing to do is to just use that language.

What is the difference between `source script.sh` and `./script.sh`

In both cases the `script.sh` will be read and executed in a bash session, the difference lies in which session is running the commands. 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. Otherwise, if you run it, the new bash process will be the one to process the function definition instead of your current shell.

What are the places where various packages and tools are stored and how does referencing them work? What even is `/bin` or `/lib`?

Regarding programs that you execute in your terminal, they are all found in the directories listed in your `PATH` environment variable and you can use the `which` command (or the `type` command) to check where your shell is finding a specific program. In general, there are some conventions about where specific types of files live. Here are some of the ones we talked about, check the Filesystem, Hierarchy Standard for a more comprehensive list.

- `/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

Should I `apt-get install` a python-whatever, or `pip install` whatever package?

There's no universal answer to this question. It's related to the more general question of whether you should use your system's package manager or a language-specific package manager to install software. A few things to take into account:

- Common packages will be available through both, but less popular ones or more recent ones might not be available in your system package manager. In this case, using the language-specific tool is the better choice.
- Similarly, language-specific package managers usually have more up to date versions of packages than system package managers.

- When using your system package manager, libraries will be installed system wide. This means that if you need different versions of a library for development purposes, the system package manager might not suffice. For this scenario, most programming languages provide some sort of isolated or virtual environment so you can install different versions of libraries without running into conflicts. For Python, there's virtualenv, and for Ruby, there's RVM.
- Depending on the operating system and the hardware architecture, some of these packages might come with binaries or might need to be compiled. For instance, in ARM computers like the Raspberry Pi, using the system package manager can be better than the language specific one if the former comes in form of binaries and the latter needs to be compiled. This is highly dependent on your specific setup.

You should try to use one solution or the other and not both since that can lead to conflicts that are hard to debug. Our recommendation is to use the language-specific package manager whenever possible, and to use isolated environments (like Python's virtualenv) to avoid polluting the global environment.

What's the easiest and best profiling tools to use to improve performance of my code?

The easiest tool that is quite useful for profiling purposes is print timing. You just manually compute the time taken between different parts of your code. By repeatedly doing this, you can effectively do a binary search over your code and find the segment of code that took the longest.

For more advanced tools, Valgrind's Callgrind lets you run your program and measure how long everything takes and all the call stacks, namely which function called which other function. It then produces an annotated version of your program's source code with the time taken per line. However, it slows down your program by an order of magnitude and does not support threads. For other cases, the `perf` tool and other language specific sampling profilers can output useful data pretty quickly. Flamegraphs are a good visualization tool for the output of said sampling profilers. You should also try to use specific tools for the programming language or task you are working with. For example, for web development, the dev tools built into Chrome and Firefox have fantastic profilers.

Sometimes the slow part of your code will be because your system is waiting for an event like a disk read or a network packet. In those cases, it is worth checking that back-of-the-envelope calculations about the theoretical speed in terms of hardware capabilities do not deviate from the actual readings. There are also specialized tools to analyze the wait times in system calls. These include tools like eBPF that perform kernel tracing of user programs. In particular `bpftrace` is worth checking out if you need to perform this sort of low level profiling.

What browser plugins do you use?

Some of our favorites, mostly related to security and usability:

- uBlock Origin - It is a wide-spectrum blocker that doesn't just stop ads, but all sorts of third-party communication a page may try to do. This also covers inline scripts and other types of resource loading. If you're willing to spend some time on configuration to make things work, go to medium mode or even hard mode. Those will make some sites not work until you've fiddled with the settings enough, but will also significantly improve your online security. Otherwise, the easy mode is already a good default that blocks most ads and tracking. You can also define your own rules about what website objects to block.
- Stylus - a fork of Stylish (don't use Stylish, it was shown to steal users' browsing history), allows you to sideload custom CSS stylesheets to websites. With Stylus you can easily customize and modify the appearance of websites. This can be removing a sidebar, changing the background color or even the text size or font choice. This is fantastic for making websites that you visit frequently more readable. Moreover, Stylus can find styles written by other users and published in userstyles.org. Most common websites have one or several dark theme stylesheets for instance.
- Full Page Screen Capture - Built into Firefox and Chrome extension. Lets you take a screenshot of a full website, often much better than printing for reference purposes.
- Multi Account Containers - lets you separate cookies into "containers", allowing you to browse the web with different identities and/or ensuring that websites are unable to share information between them.
- Password Manager Integration - Most password managers have browser extensions that make inputting your credentials into websites not only more convenient but also more secure. Compared to simply copy-pasting your user and password, these tools will first check that the website domain matches the one listed for the entry, preventing phishing attacks that impersonate popular websites to steal credentials.
- Vimium - A browser extension that provides keyboard-based navigation and control of the web in the spirit of the Vim editor.

What are other useful data wrangling tools?

Some of the data wrangling tools we did not have time to cover during the data wrangling lecture include `jq` or `pup` which are specialized parsers for JSON and HTML data respectively. The Perl programming language is another good tool for more advanced data wrangling pipelines. Another trick is the `column -t` command that can be used to convert whitespace text (not necessarily aligned) into properly column aligned text.

More generally a couple of more unconventional data wrangling tools are vim and Python. For some complex and multi-line transformations, vim macros can be a quite invaluable tool to use. You can just record a series of actions and repeat them as many times as you want, for instance in the editors lecture notes (and last year's video) there is an example of converting an XML-formatted file into JSON just using vim macros.

For tabular data, often presented in CSVs, the pandas Python library is a great tool. Not only because it makes it quite easy to define complex operations like group by, join or filters; but also makes it quite easy to plot different properties of your data. It also supports exporting to many table formats including XLS, HTML or LaTeX. Alternatively the R programming language (an arguably bad programming language) has lots of functionality for computing statistics over data and can be quite useful as the last step of your pipeline. ggplot2 is a great plotting library in R.

What is the 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. In Linux, this is achieved through a mechanism called LXC, and it makes use of a series of isolation mechanisms to spin up a program that thinks it' s running on its own hardware but it' s actually sharing the hardware and 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. For instance if you run Docker on macOS, Docker needs to spin up a Linux virtual machine to get an initial Linux kernel and thus the overhead is still significant. 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.

What are the advantages and disadvantages of each OS and how can we choose between them (e.g. choosing the best Linux distribution for our purposes)?

Regarding Linux distros, even though there are many, many distros, most of them will behave fairly identically for most use cases. Most of Linux and UNIX features and inner workings can be learned in any distro. A fundamental difference between distros is how they deal with package updates. Some distros, like Arch Linux, use a rolling update policy where things are bleeding-edge but things might break every so often. On the other hand, some distros like Debian, CentOS or Ubuntu LTS releases are much more conservative with releasing updates in their repositories so things are usually more stable at the expense of sacrificing newer features. Our recommendation for an easy and stable experience with both desktops and servers is to use Debian or Ubuntu.

Mac OS is a good middle point between Windows and Linux that has a nicely polished interface. However, Mac OS is based on BSD rather than Linux, so some parts of the system and commands are different. An alternative worth checking is FreeBSD. Even though some programs will not run on FreeBSD, the BSD ecosystem is much less fragmented and better documented than Linux. We discourage Windows for anything

but for developing Windows applications or if there is some deal breaker feature that you need, like good driver support for gaming.

For dual boot systems, we think that the most working implementation is macOS' bootcamp and that any other combination can be problematic on the long run, specially if you combine it with other features like disk encryption.

Vim vs Emacs?

The three of us use vim as our primary editor but Emacs is also a good alternative and it' s worth trying both to see which works better for you. Emacs does not follow vim' s modal editing, but this can be enabled through Emacs plugins like Evil or Doom Emacs. An advantage of using Emacs is that extensions can be implemented in Lisp, a better scripting language than vimscript, Vim' s default scripting language.

Any tips or tricks for Machine Learning applications?

Some of the lessons and takeaways from this class can directly be applied to ML applications. As it is the case with many science disciplines, in ML you often perform a series of experiments and want to check what things worked and what didn' t. You can use shell tools to easily and quickly search through these experiments and aggregate the results in a sensible way. This could mean subselecting all experiments in a given time frame or that use a specific dataset. By using a simple JSON file to log all relevant parameters of the experiments, this can be incredibly simple with the tools we covered in this class. Lastly, if you do not work with some sort of cluster where you submit your GPU jobs, you should look into how to automate this process since it can be a quite time consuming task that also eats away your mental energy.

Any more Vim tips?

A few more 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 lets 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.

- Undo with time - The `:earlier` and `:later` commands will let you navigate the files using time references instead of one change at a time.
- 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 store a per-file history of changes.
- Leader Key - The leader key is a special key that is often left to the user to be configured for custom commands. The pattern is usually to press and release this key (often the space key) and then some other key to execute a certain command. Often, plugins will use this key to add their own functionality, for instance the UndoTree plugin uses `<Leader> U` to open the undo tree.
- Advanced Text Objects - Text objects like searches can also be composed with vim commands. E.g. `d/<pattern>` will delete to the next match of said pattern or `cgn` will change the next occurrence of the last searched string.

What is 2FA and why should I use it?

Two Factor Authentication (2FA) adds an extra layer of protection to your accounts on top of passwords. In order to login, you not only have to know some password, but you also have to "prove" in some way you have access to some hardware device. In the most simple case, this can be achieved by receiving an SMS on your phone, although there are known issues with SMS 2FA. A better alternative we endorse is to use a U2F solution like YubiKey.

Any comments on differences between web browsers?

The current landscape of browsers as of 2020 is that most of them are like Chrome because they use the same engine (Blink). This means that Microsoft Edge which is also based on Blink, and Safari, which is based on WebKit, a similar engine to Blink, are just worse versions of Chrome. Chrome is a reasonably good browser both in terms of performance and usability. Should you want an alternative, Firefox is our recommendation. It is comparable to Chrome in pretty much every regard and it excels for privacy reasons. Another browser called Flow is not user ready yet, but it is implementing a new rendering engine that promises to be faster than the current ones.