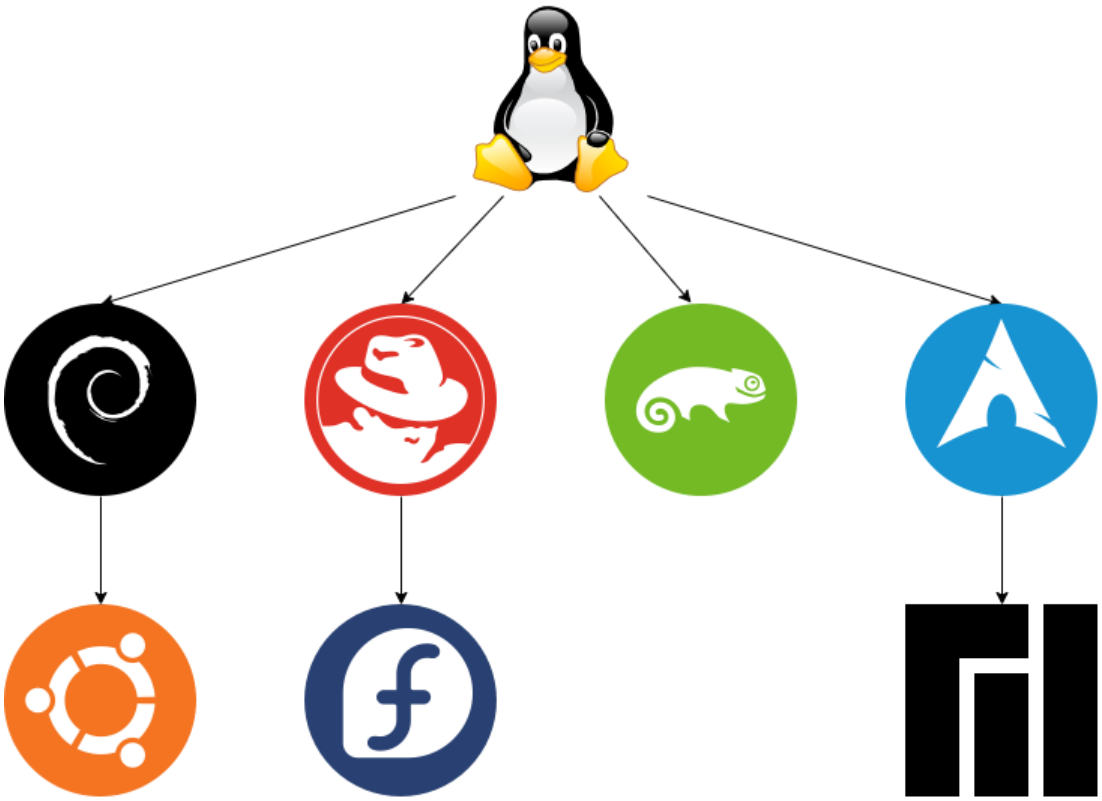
Lab 1 - Linux Basics

Introduction

The purpose of this lab is to get familiar with the environment that will be used in the following labs. Linux is an open-source Unix-like operating system that offers great flexibility for teaching and making experiments.

Variations

Linux consists of a kernel and other software that facilitates the capabilities of a full-fledged operating system. Linux is being used for very diverse purposes; PCs, servers, embedded systems, supercomputers, etc. Therefore, it comes in many flavors called *distributions*, or *distros* for short. Following is a sample of some of the more famous ones! Here is a (very) long list!



GUIs and CLIs

Users interact with the operating system through some kind of an *interface*. An interface can be categorized as a *graphical user interface (GUI)* or a *command-line interface (CLI)*. While GUIs are more familiar with users coming from other operating systems like Windows and MacOS, CLIs can be *much* more efficient in administrative and complicated tasks, especially when needed to be carried out repeatedly and/or on multiple machines.

Most modern PC-targeted distros include a GUI by default called a *desktop environment (DE)*. Examples of famous DEs include *GNOME*, *KDE*, *XFCE*, *LXDE*, *Unity*, and *Pantheon*.

There are a handful of CLIs (aka *shells*) in Linux, and they behave identically across all distributions. The most popular of them is *bash* which will be used extensively in this course.

Commands

Commands are executables that perform a specific task. They can accept one or more *options* and *arguments*. Following are samples of basic commands!

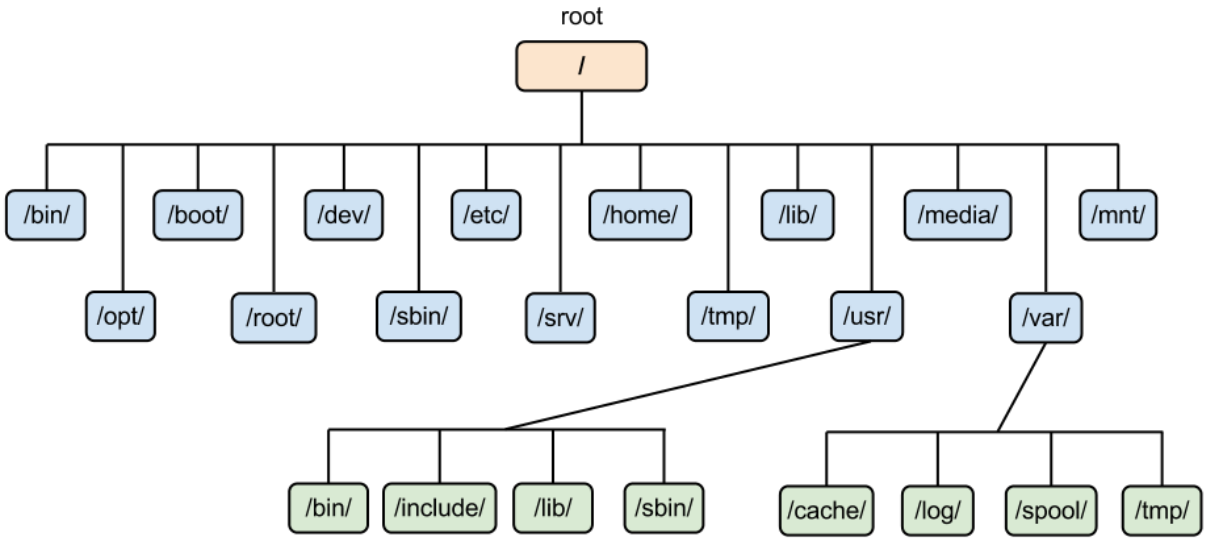
`whoami`	Prints the name of the current user.
`who`	Lists all users currently logged in the system.
`uptime`	Prints the time the system has been active.
`date`	Prints the date of the day.
`clear`	Clears the screen.
`echo`	Prints whatever comes after it.
`uname -a`	Prints the kernel version (`-a` is an *option*)
`man <command>`	Shows the manual of a command (`<command>` is an *argument*)

Note that commands are VERY sensitive to spaces!

Files and Paths

Everything in Linux is a file! Files are organized into directories (folders) which can include both files and other directories. A path is a way of locating a file (or directory) within the *file system hierarchy*.

Filesystem Hierarchy



In Linux, there are no drive letters. A single device is designated as the *root*. This device has the special responsibility of containing directory *mount points*. Directory mount points are just regular directories in which drives and partitions are mounted. All drives and partitions (floppy disks, CD-ROMs, hard disk, removable USB drives, etc.) are configured to be mounted as one of the directories on the file system. Most mount-points are put into a directory called */mnt*. This is merely a convention. Thus, the CD-ROM is mounted at the location */mnt/cdrom*, the floppy disk can usually be found at the location */mnt/floppy*. Most Linux distributions will automatically detect when a CD-ROM, a floppy disk, or other removable media are inserted, and automatically mount them to an appropriate location. Before the media can be ejected, unmount must be performed (often the GUI will do the unmount automatically).

Each user has a *home* directory */home/<username>* where they keep their personal data.

Moving Around

An *absolute* path is a path that starts at the root (always starts with a /) and includes all the parent directories up to the designated file (or directory).

A *relative* path is a path that starts from the current *working directory* (never starts with a /) and includes how to reach the designated file (or directory) from it.

`pwd`	Prints the current working directory.
`cd` *<directory>*	Changes the working directory.
`mkdir` *<directory>*	Creates a new empty directory.
`rmdir` *<directory>*	Deletes an empty directory.
`touch` *<file>*	Creates a new empty file.
`rm` *<file>*	Deletes a file.
`mv` *<file/directory>*	Moves/Renames a file or a directory.
`cp` *<file/directory>*	Copies a file or a directory.
`file` *<file>*	Prints a file type depending on its contents (not its extension).
`cat` *<file>*	Displays the contents of a file.
`rm -r` *<directory>*	Deletes a directory and its contents.

Editors

Editors are programs that help *edit* files (it is in the name!), and considering that *everything in Linux is a file*, they are extremely important to carry out any task in Linux. There are great GUI editors including Visual Studio Code, which we will be using extensively in future labs (note that it is very different from the Visual Studio IDE). However, the ability to use *at least one* of the default CLI editors can be very helpful; these include `nano` (the simplest one), `emacs` (a very powerful editor), and `vi`/`vim` (an equally powerful editor that managed to trap > 1 million users so far!).

Commands are files as well!

Again, *everything in Linux is a file!* This includes (most) commands as well. Most commands are in fact just executables (programs) that run when their names are typed (with the exception of the internal shell commands like `cd`). The `which` command can be used to find out where each command is on the system (including `which` and the shell themselves!).

But how does the shell know *where* to look? Does it search all drives for an executable matching the name of each command? Well… it searches, but only a handful of paths; namely the paths included in the environment variable *PATH*. (Environment variables are very important to the shell and they will be covered, along with variables in general, in a future lab. For now, the command `echo $PATH` can be used to print these paths. *Knowing this, can you guess what actually happens when you "install" a new command or software?)*

IO Redirection

Again and again, *everything in Linux is a file!* This includes inputs and outputs storage. Each command has 3 streams (aka *pipes*): *input (stdin)*, *output (stdout)*, and *error (stderr)*. By default, they are attached to the shell. That is, read input from it and write outputs and errors to it. Alternatively, they can be *redirected* to a file or to other commands by the following operators!

`cmd < file`	Redirects `cmd` input stream to read from `file`
`cmd > file`	Redirects `cmd` output stream to (over)write to `file`
`cmd 2> file`	Redirects `cmd` error stream to write to `file`
`cmd >> file`	Redirects `cmd` output stream to append to `file`
`cmd > /dev/null`	Discards `cmd` output stream (yes, even `null` is a file!)
`cmd1 | cmd2`	Redirects `cmd1` output stream to `cmd2` input stream (aka *piping*)