Lab 1: Linux

Cloudbase Solutions AC Labs 1

Cloudbase Solutions SRL

Liga AC Cloudbase Labs 1: Linux

Cloudbase Solutions

April 17, 2023

Abstract

This document, which is based on the contents of the GNU/Linux Liga AC LABS 2023 session held by Cloudbase Solutions, is meant to act as a "begginer's guide" to the history and inner workings of GNU/Linux, whose openness and reliability have made it the dominant server-side Operating System of our time.

The main takeaway for the Reader should be how Linux's Open Source nature and and ease of inspection of its internal workings make it the best place for learning how Operating Systems work.

We will walk you through some of the hardware considerations that went into the design of the Linux kernel and its boot process, the numerous abstractions offered to the User by Linux, and the various pieces of software that go into forming a fully-fledged Linux distribution.

After some initial theoretical considerations, this document will focus on applying all the theory through numerous Bash command listing which we encourage the Reader to follow along with on their own Linux system.

An additional section containing tasks geared towards the Reader applying the knowledge on their own will be included at the end as well.

Contents

Listings:

Chapter 1

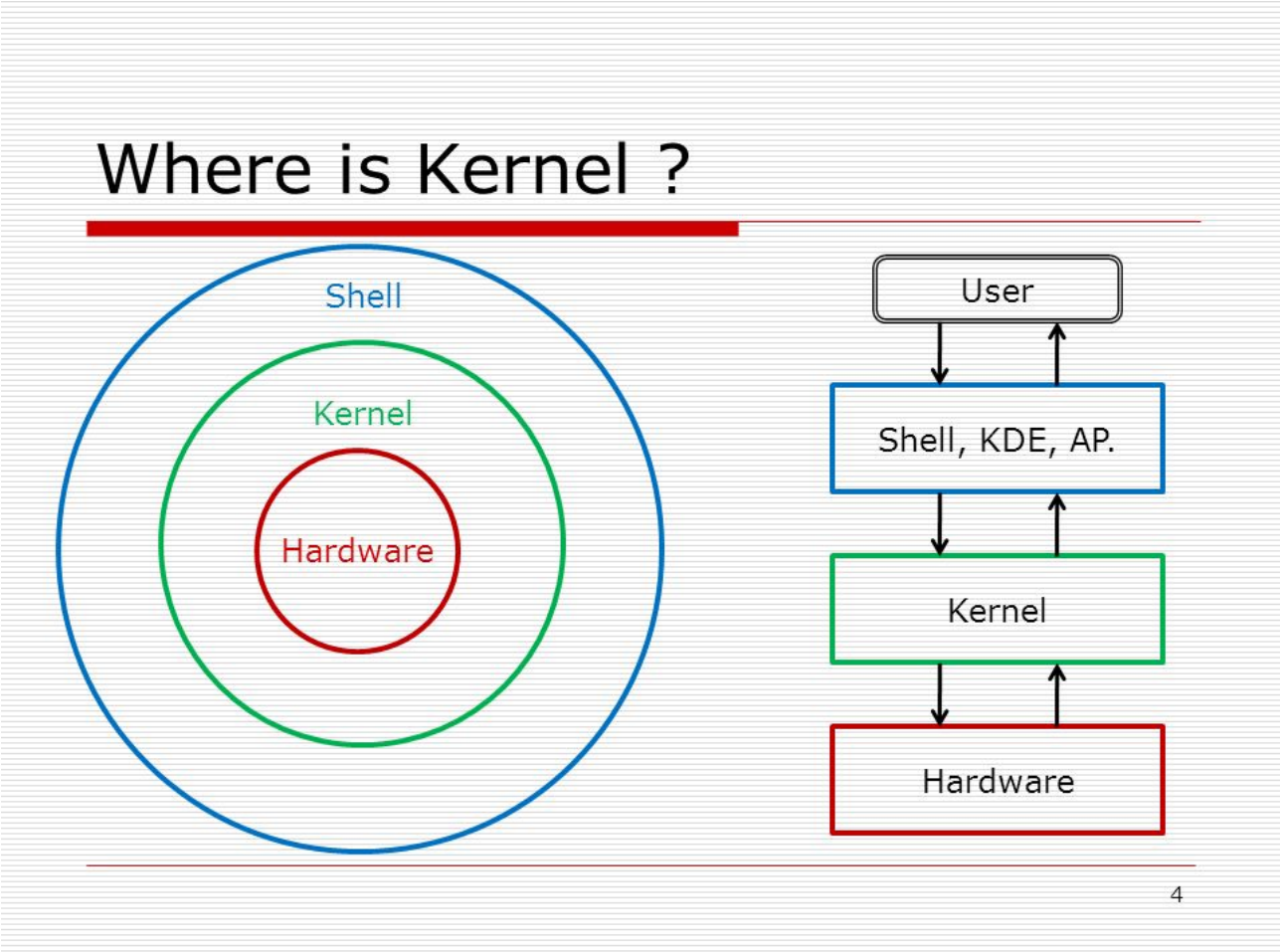
Operating Sytems of Abstraction

Abstracting (verb) is the process of considering something theoretically or separately from something else.

Oxford Dictionary

That might sound vague, but it is exactly what Operating Systems (**OS**es) allow programmers and users to do: disconsider (i.e. **abstract away**) hardware details and put their trust in the *higher-level* functionality offered by the OS.

Figure 1.1: Operating System layers diagram.



1.1 Modern OS features.

Most modern OSes offer the same set of high-level features:

- **hardware abstraction**: allow users/programmers to only care about interacting with the OS in question, and not with whatever underlying hardware the machine is using.
- **multi-process**: manage the *lifecycle* of user applications by:
  - allocating memory (RAM) for each application.
  - scheduling applications to be run on the CPU.
  - offering functionality to applications through syscalls:
    - file I/O.
    - network access.
    - user interactions through keyboard or mouse events.
- **human-usable**: be as streamlined for users to interact with as possible by offering:
  - a **view into the system** for the user. (aka the **console** we see on our monitors)
  - **system control** for the user through a **shell**, aka **User Interface (UI)**:
    - **Graphical User Interface (GUI)**: clicky mouse-y environments we all know and love. Fan favorites include Windows, macOS, `GNU`/Linux, iOS, Android, and many more...
    - **Line/Character User Interface (LUI/CUI)**: strictly text-driven UIs favored by the hyper-efficient. Common favorites include `powershell.exe` on Windows or `bash` on Linux.
- **multi-user**: allow more than one user to be `logged into the system` at the same time.

1.2 Imagining zero abstraction

The list of things OSes do for us is so extensive, it might be worth taking a step back and trying to to imagine a world without them just to put things into perspective.

1.2.1 One machine per program

Without an OS to manage virtual memory for multiple applications running in parallel, each individual computer will be cursed to only load and execute one application at a time, because every application ever written would "feel entitled" to allocating memory starting from address `0x00`.

Virtual memory allows apps to not have to consider their memory range, as the OS transparently manages a little *memory sandbox* for all its cute little child processes. In this memory sandbox, user applications can all assume they're running at `0x00` and the OS will map an actual piece of physical memory for them.

If any of the child processes is greedy enough to try to allocate infinity Mb of RAM, or access the memory of other processes, they immediately get `SEGFAULT`'ed.

1.2.2 Begging programs to stop

Without an OS to schedule CPU time in a completely fair manner (wink wink), users would need to wait for each and every single application which manages to get loaded on the CPU to exit (successfully or not), before machine control can be handed to a new process.

While it is theoretically possible to have *all* applications behave nicely and give up CPU time to other applications periodically (referred to as cooperative multitasking), in practice the only way to reliably (and fairly!) schedule CPU time to applications is to have the OS manage which apps get to execute when, and allow the OS to interrupt any process which is being too greedy, or to play favorites schedule some other process in its place.

1.2.3 3000-line Hello Worlds

Behold, the 3k+ lines of code for for driving a TTY (the console device your terminal application is showing you), in all its glory.

Whenever you do a `printf()`, your output is processed through this code from the Linux kernel before showing up on your screen, and without it, every `hello_world.c` ever written would have to copy-paste this code (and God knows how much more code) just to be able to say hello to you.

Chapter 2

What makes Linux. . . a Linux?

Despite all of us claiming we're "Linux users", the reality is that **nobody** uses Linux by itself, because Linux is only the **kernel** of the operating system.

What we've all been actually using are **Linux distributions**, which are pre-compiled and pre-curated selections of applications which happened to need to also come with the Linux kernel for them to run. (see 1.2.3)

2.1 UNIX: the original Linux.

You may have noticed that some of the features we previously listed (1.1) such as "multi-process" and "multi-user" seem. . . pretty trivial? That's because we've all been priviledged enough to always have them and take them for granted, but back in the late 60's there was no such luck.

Wanted to use the computer? Feel free to schedule a 15m session on Monday. Got a quick calculation you wanted to run? Something's already running until February. Had separate text formatter and text highlighter programs but wanted to both format and highlight your text? Guess it's time to write a text fomat-highlighter from scratch. . .

Out of this dystopia, UNIX would rise above the masses of outdated systems and obtuse designs with its (at the time) novel approach to multi-user and multi-processes, as well as introduce a set of core *design values* which would end up being reflected everywhere within the system:

- make each program do ONE thing, but do it really well.
- **combine the inputs/outputs** of smaller, more focused programs into **program pipelines**.
- **focus on delivering intermediate versions of programs** early and iterating on them (aka proto-Agile)
- always **prefer building tools for doing things reliably**, even if the tool seems like it won't be used after its job is done.

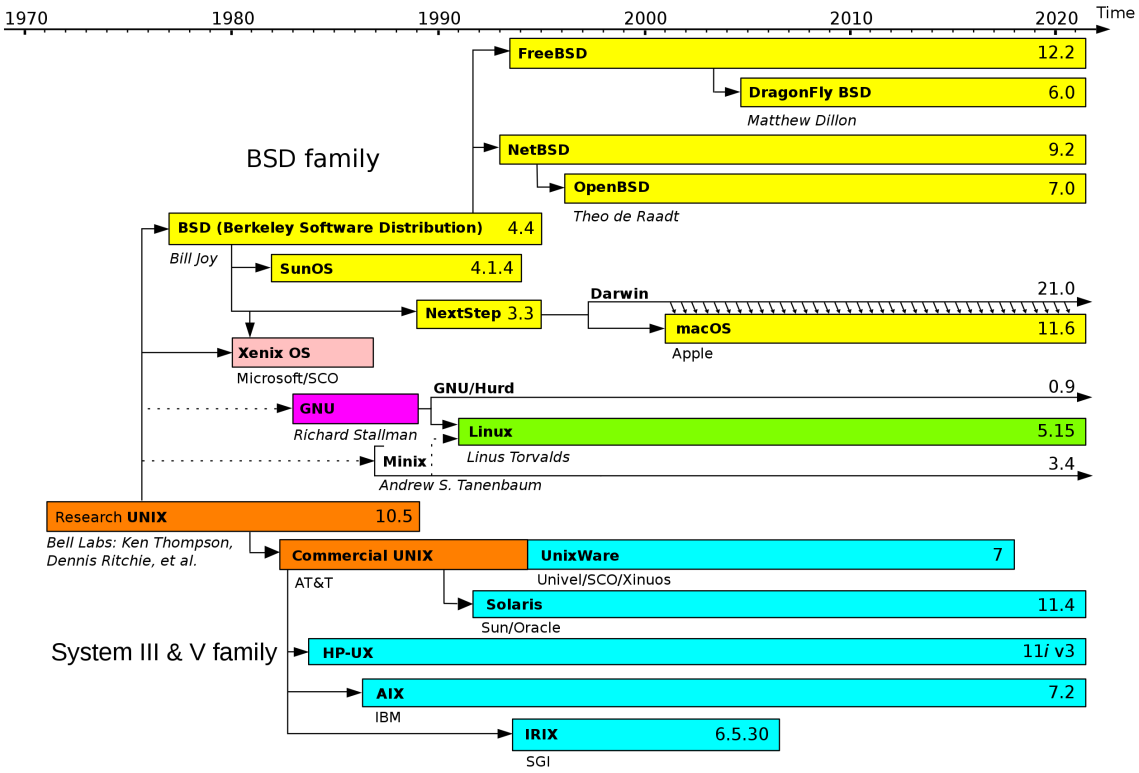
Naturally, the philosphy prevailed, and UNIX became the new standard for what an OS should be almost overnight. Every tech company and their uncle copied everything they could for their own decrepit systems, and after a period a relative peace and innovation, the period that would come to be known as the "UNIX wars" began.

2.2 POSIX progenitors.

The problems faced by Operating System developers are as constant as time itself, and as such, the solutions have usually been pretty *cough, very* similar.

Back in the 80ś, so many companies were pumping out so much hardware which was fundamentally not cross-compatible, which lead to each shipping their own proprietary version of UNIX. These UNIX derivatives (also called "*NIXes" because they even ripped off the naming), despite being all based on the same UNIX philosophies, were virtually incompatible between each-other (sometimes even between machines from the same company!)

Figure 2.1: UNIX Wars timeline..



In a vain attempt to standardize, the industry collectively settled on what became to be known as POSIX, or the **P**ortable **O**perating **S**ystem **I**nterface. (the 'X' was, fittingly, stolen from UNIX for no reason. . .)

Alas, all that did was make the corporations even more competitive, since strong-arming their users on compatibility limitations was no longer an option, and the UNIX wars degenrated into the UNIX meltdown.

2.3 The GNU in GNU/Linux is important.

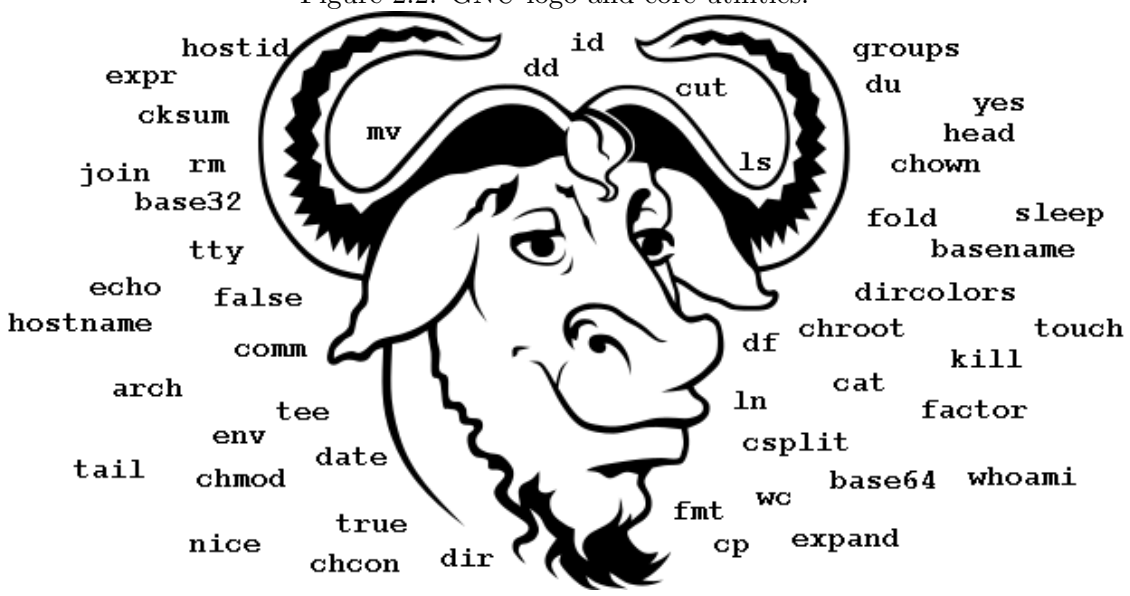
The GNU project (**G**NU's **N**ot **U**nix) was a concerted effort by a group of American software engineers (most notably Richard Stallman, an actual Holy Ghost of software ethics) who was fed up with the lack of inter-operability of 1980s UNIX-based operating systems and decided to write a new and free (as in speech!) alternative named GNU.

It is impossible to understate the cultural importance the GNU project has had on the software industry, forcing it to a move from an interest-driven, almost anti-operability attitude to embracing Open Source and interoperable media formats.

Even now, half a century later, the legacy of the GNU project touches us all through software like:

- glibc: a collection of standard C libraries you are 100% guaranteed to have `include<>`d at least once in your life.
  - the GNU core utilities: the `ls`es and `mkdir`s and `rm -rf /`es we all know and love.
- the GRUB bootloader: that little menu thing that lets you choose between Linux and Windows if you're dual-booting both OSes on your laptop.
- GIMP: a free and Open Source Photoshop alternative.
- obscenely many more.

Figure 2.2: GNU logo and core utilities.



2.3.1 The Linux kernel: right place, right time.

Back in the early 90's, while the GNU project (2.3) was going along generally swimmingly, the planned GNU Hurd kernel, a relatively avangarde attempt at a new kernel design called a microkernel, was not going as well...

Coincidentally, a Finnish CS student named Linus Torvalds was independently sticking it up to the man by attempting to implement a clone of a Unix kernel (just the kernel!), which he called Linux.

Like two sides of the same coin, the Linux kernel and GNU System utilities and user applications were merged into the most disruptive force the OS industry had seen.

By 2000, Linux had a 25% market share in the online HTTP server space. By 2010, the vast majority of every website was running a LAMP stack or some sort of popular Linux-based setup.

The Internet and the world as a whole would be a considerably more closed and more human-hostile place without the efforts of the **GNU** project, and Linux's integration into it.

2.4 Anatomy of a Linux distro

Although the goals of a Linux distribution greatly impacts how the finished distro will look, any "fully-fledged" modern Linux distribution will almost surely contain the following:

2.4.1 Kernel and Init System.

- a pre-compiled **Linux kernel** alongside:
  - a **bootloader**: a program meant to be executed by the PC's firmware after startup. The sole purpose of the bootloader is to load the Linux kernel and some supporting data into memory and start executing the kernel. Your Ubuntu install is almost surely using GRUB.
  - an **initrd** ("**init**ial **R**AM **D**isk"): is basically a special disk image format which contains device drivers (usually as kernel modules) and some initial cofiguration data used by the Linux kernel for initial startup.
- an **init system**: a program meant to be the first program (PID 1) started by the Linux kernel after it finishes its startup procedure. The purpose of the init system is to start all the non-kernel services needed by the user, such as:
  - SSHD for remote connections, or
  - CUPS for sharing an attached printer within an office network.
  - the Desktop Environment's core services.

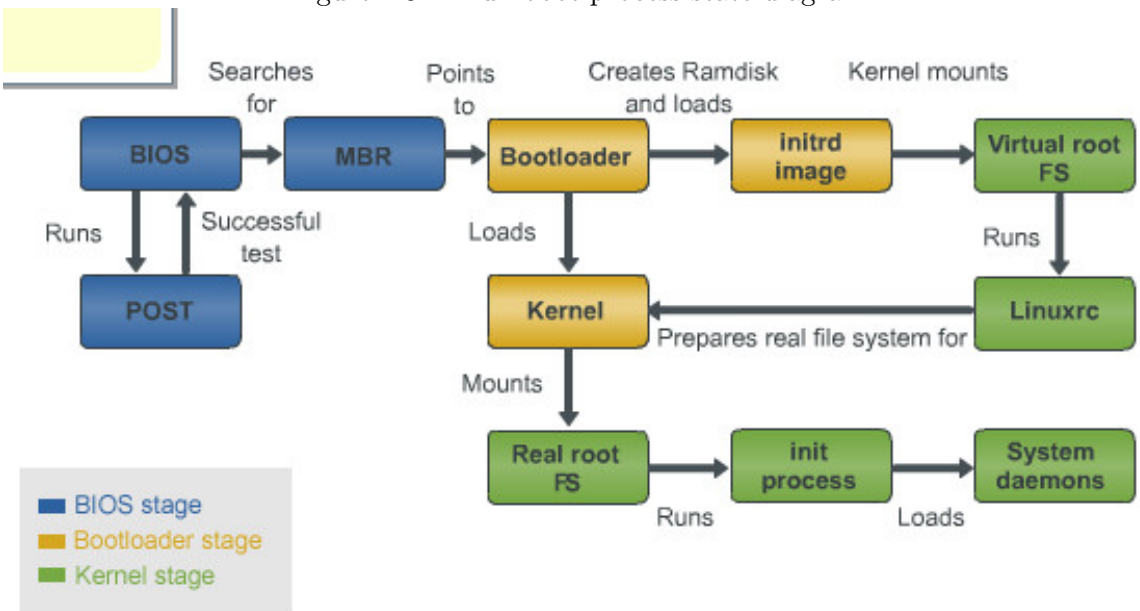
2.4.2 Package Manager.

- a **package manager**: given the nature of Open Source, the only "inconvenience" between you and your software is building/installing it. For this, all Linux distros come with a dedicated **package management system** which is used to:
  - configure and check remote **package repositories** (repos with older packages are also called **package archives**) for any software available to install and update.
  - download **pre-built packages**, **including their dependencies** at the user's request.
  - **resolve version mismatches** (a.k.a. "dependency Hell") when different packages depend on different versions of one root package.
  - **unpack and install** the selected packages.
  - **run pre/post-install scripts** for the packages at the correct time packages and all associated dependencies at user requests.
  - **uninstall** packages and all associated dependencies at user request.

2.4.3 Userspace programs.

- the **Desktop Environment** (DE): a collection of services which provide a Graphical User Interface (GUI) for humans to interact with the computer through. DEs can have varying amounts of moving parts depending on the number of features it aims to offer, but most have a:
  - **Display Server**: a service who manages applications in the "display area" of the graphical environment. Display Servers are responsible for forwarding user events (e.g. mouse clicks or keyboard button presses) to the right application on the desktop.
  - **Window Manager**: a service which controls how the applications are shown and drawn on screen. Most of us have only ever used a Compositing Window Manager, which allows users to move and arrange Windows freely across the desktop draw space. Other implementations exist of course, with a popular alternative being Tiling Window Managers, which strictly allow users to arrange Windows in a grid as "tiles".

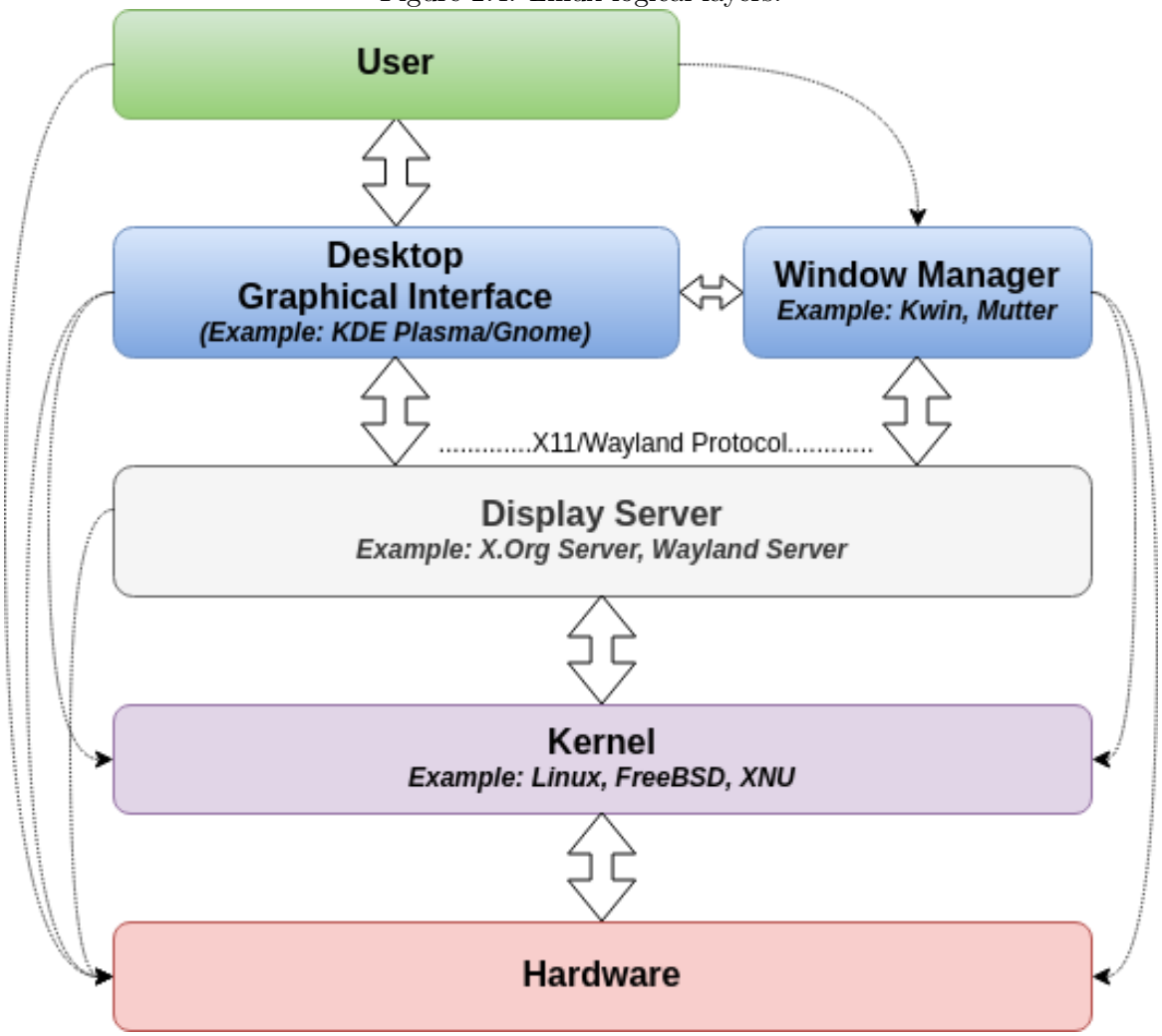
Figure 2.3: Linux boot process state diagram.



Last, but certainly not least, any Linux distro meant for human use should obviously include apps for human use, such as:

- a **Web browser**:
  - **Chromium**: an Open Source Chrome alternative.
  - **Firefox**: d'oh.
  - **Desktop Env-specific browsers**: made by and for commutites of Desktop Environments. Most notable example would be Konqueror on KDE-based distros.
- **media playback**:
  - **photos**: usually a Desktop Environment-specific Photo app.
  - **videos**: VLC, the only video player anyone ever needs.
- **productivity**:
  - **office docs**: Microsoft Office alternatives like LibreOffice or OpenOffice.
  - **programming IDEs**: Eclipse and the usual crowd.
  - **programming-focused editors**: Vim, Emacs, VSCode, and many more...

Figure 2.4: Linux logical layers.



2.5 Notable Linux distributions.

There are literally hundreds of actively-maintained Linux distributions, the topmost popular of which (at least in terms of user popularity) are tracked on DistroWatch.

Here's a short table with some of the most popular OSes out there and what their core components are named so you can look up the ones that pique your interest:

OS	Audience	Kernel	Desktop Env Name	App Format	Exe Format
Windows	Users	Windows NT	Windows	MSI	`.exe`
Android	Users	Linux	Dalvik?	`.apk`	`.jar`
Ubuntu	Users	Linux	GNOME/Wayland	`.deb`	ELF
Red Hat	Enterprise	Linux	GNOME/Xorg	`.rpm`	ELF
OpenBSD	Users	BSD	`cwm`/Xorg	Ports	ELF
macOS	Users	Darwin	Aqua!?	DMG	Mach-O exe
Red Star OS	N. Koreans	Linux	KDE	Deb?	ELF

Chapter 3

Shelling 101

3.1 Shell for what?

Why, the shell is for you, the user, to interact with the low-level features offered by the operating system.

Low-level features like:

- launching a process
- checking on the state of the system
- interacting with devices attached to the system

At the end of the day, the shell is a just program just like any other program you may have interacted with, it just has a couple of special characteristics which sets it apart:

- shells are (`REPL`s):
  - **R**eads user input.
  - **E**valuates what the user wants.
  - **P**rints the result.
  - **L**oops back to the beginning Read stage.
- shells are programming *tools* with:
  - limited types: all shell data is 100% a string, 100% of the time!
  - limited constructs: just some `if`s and loops and hoops...
  - *very* limited actual functionality, as most network/system/etc tasks are done by a program the shell executes.

3.2 Just a handful of shells.

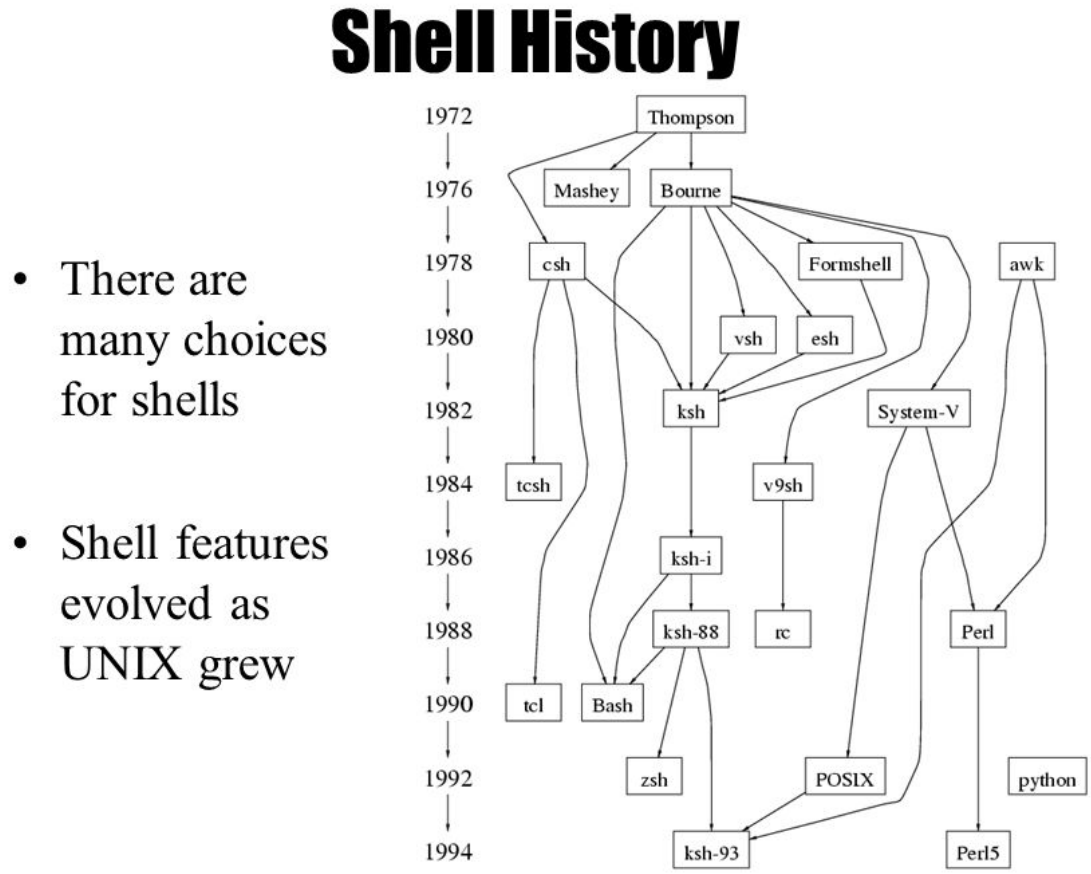
There have been numerous shells throughout history, but here's a list of the notable ones:

- `sh` (Bourne shell, '76): one of the most popular shells in the 80s, and the first to combine numerous syntactic niceties such as `for` loops and global variables.
- `csh` (C shell, '78): a shell whose syntax was meant to be closer to C that the Fortran-e shell languages at the time. Added numerous user-focused features like command history and searching.
- `ash/dash` (Almquist shell, '89): a shell designed to consume as little resources as possible, offering decent script compatibility but very few user features. Meant for embedded or smaller-scale systems like routers etc...
- `bash` (Bourne Again (LOL) shell, '89): a shell built from scratch by the GNU project (2.3) for their new free and Open Source software ecosystem. Combines syntax and features from both `sh` and `csh`.

- `zsh` (Z shell, '90): fully `bash` compatibile, but with considerably more user-focused features and extensibility.
- `fish` (Friendly Interactive Shell, 2005): ditches `bash` compatibility entirely in favor of a considerably higher-level scripting language and way more user-oriented features.

Consider trying out as many shells and you can until you find the right one for you!

Figure 3.1: UNIX Shell family tree.



3.3 It's just a text-based file manager.

The current shell of choice which comes pre-bundled with the vast majority of Linux installations is `bash`, so it the shell we shall be using moving forward.

Open up your favorite terminal emulator and try to follow along.

```
# You can type commands like "echo" and its arguments and press Enter to execute:
$ echo "Hello $USER, you look lovely today!"
Hello <You>, you look lovely today!!

# What a pleasant and courteous shell. You can thing of shells like File Explorer
# apps, in that they're always in a "current working directory" open, check with pwd:
$ pwd
/home/you

# You can see the files in your current working directory with "ls":
$ ls
<probably nothing>

# You can create a new directory with "mkdir", and change to it using "cd":
$ mkdir new
$ ls
new
$ cd new
$ pwd
/home/you/new

# You can create an empty file using "touch":
$ touch new_file
$ ls
new_file

# Remove the file using "rm":
$ rm new_file
$ ls
<nothing again>

# Two dots ".." is a "shortcut" for the parent directory:
$ pwd
/home/you/new
$ cd ../
$ pwd
/home/you

# Finally, you can remove directories (if empty) using "rmdir":
$ rmdir new; ls
<probably nothing again>
```

Listing 1: Basic bash navigation.

3.4 Stop! Shortcut time.

```
# NOTE: these examples show the cursor as "|". DO NOT actually write "|"!
# First, write out a random long sentence. The more senseless, the better!
$ Everybody knows Windows is objectively better than Linux.| # Press Enter
Everybody: command not found

# Ah dammit, I meant to print that out but I forgot to write "echo".
# DO NOT type out the whole sentence again, instead just write:
$ echo !!
# NOTE: Bash will both print the replaced command, and run it for you:
echo Everybody knows Windows is objectively better than Linux.
Everybody knows Windows is objectively better than Linux.

# Nice, but reading that back out loud, it hit me that I misspelled "worse".
# DO NOT type out everything again, instead press the UP arrow key:
<press up arrow key>
$ echo Everybody knows Windows is objectively better than Linux.|

# DO NOT click or use the side arrow keys to move one letter at a time!!!
# Hold down Control (or Alt in some shells) and arrow keys to scroll words:
<press Ctrl+Left arrow 2 times>
$ echo Everybody knows Windows is objectively better |than Linux.

# DO NOT hold the backspace key to delete the previous word. Instead use Ctrl+W:
<press Ctrl+W to delete previous word>
$ echo Everybody knows Windows is objectively |than Linux.
<now type out "worse " and press Enter>
$ echo Everybody knows Windows is objectively worse |than Linux.
Everybody knows Windows is objectively worse than Linux.

# Perfect, but now I decided that everybody's dumb and only I know the truth.
<press UP arrow key>
$ echo Everybody knows Windows is objectively worse than Linux.|
<press Ctrl+A to go to start>
<press Ctrl+Right arrow twice>
$ echo Everybody| knows Windows is objectively worse than Linux.
<press Ctrl+W to delete "Everybody" and replace with "$USER">
$ echo $USER| knows Windows is objectively worse than Linux.
<Me> knows Windows is objectively worse than Linux.

# Now I want to go all the way back to my first clearly false statement though.
# DO NOT hold the UP arrow key until the command comes up.
# Use Ctrl+R and type out a relevant piece of the command and Ctrl+E to select:
<press Ctrl+R and type out "better", and then Ctrl+E>
<press Ctrl+C to exit the search anytime>
$ echo Everybody knows Windows is objectively better than Linux.
Everybody knows Windows is objectively better than Linux.

# Like loving Windows, browsing the shell one character at a time is equally
# dellusional. Make sure to learn these shortcuts well to live a happy life!
```

Listing 2: Bash keyboard shortcuts.

3.5 What are these commands, anyway?

A command is simply a string containing:

- Word 1: the **name of the executable** of the command.
- Words 2+: the **command arguments** which are passed to the executable.

To give a pertinent example in C:

```c
#include <stdio.h>

// This is a trivial C program which prints each command line argument
// given to it back to the screen.
// bash -c "my_command --arg1 --arg2=5 --arg3 val3"
// main(5, {"my_command", "--arg1", "--arg2=5", "--arg3", "val3"});
int main(int argc, char *argv[])  {
    // argc = number (aka "count") of arguments (including command name)
    for(int i = 0; i < argc; i++) {
        // argv = array (aka "vector") of strings with arguments
        printf("Arg #%d: %s\n", i+1, argv[i]);
    }
    return 0;
}
```

Listing 3: Command line argument handling in C programs.

```
# What is "ls" actually?
$ ls
<something>

# "ls" is an executable like any other, you can even make it list itself:
$ ls /usr/bin/ls
/usr/bin/ls

# Don't think it's the actual "ls"? Try having Bash execute it:
$ /usr/bin/ls
<that something again>

# But what happens when we try to run "ls ls"?
$ ls ls
ls: cannot access 'ls': No such file or directory
```

Listing 4: Understanding how bash commands are called.

Ok, let's try to deconstruct that "ls ls" abomination we just had Bash do above:

- Bash recognized the first "ls" as "/usr/bin/ls" and executed it
- Bash treated the second "ls" as the simple string argument "ls", and passed it as the first argument to "/usr/bin/ls".
- "/usr/bin/ls" then thought we, the user, asked it to list all the files in the *directory* named "ls". (which failed miserably)

So now we know that Bash only substitues the first "word" in the command to the actual executable it calls, and passes everything else as arguments, but how does Bash know *where* the actual executables are located?

```
# Bash has a special environment variable called $PATH.
# It is a ':'-separated string of system paths where executables are searched for.
# Note "/usr/bin", where "ls" lives, is part of the list.
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin

# We can check where a command is located using the "which" command:
$ which ls
/usr/bin/ls

# Great times to be had indeed, but does this mean Bash can't ever run
# any program which is outside of those specific directories!?
# Let's set up quick experiment in a temporary testing directory:
$ mkdir testdir
# This will copy "/usr/bin/ls" into "testdir/myls"
$ cp /usr/bin/ls testdir/myls

# Now let's try to run our new "myls" binary.
$ myls
myls: command not found

# That can't work, since the "testdir" directory is outside of Bash's $PATH.
# In order to execute "myls", we need to tell Bash exactly where it is.
# Option 1: Prefix with "./" to tell Bash it's in a path relative to
# the current directory (which is a shortcut for ".")
$ ./testdir/myls
<your stuff>
# Option 2: give it the absolute path starting from "/", can call from any dir:
$ /home/you/testdir/myls
<your stuff>

# Option 3: Update the $PATH
# NOTE: do NOT put spaces around the '=', and ensure "/home/you/testdir"
# is the correct path where "myls" is located!
$ PATH="$PATH:/home/you/testdir"
$ which myls
/home/you/testdir/myls
$ myls
<your stuff again>

# Cool, now we can use "myls" in any directory without needing its path!
$ cd /
$ myls
<the stuff in your root folder>

# Hey, let's "which" a bunch of random commands for fun:
$ which rm
/usr/bin/rm
$ which which
/usr/bin/which
$ which cd
<nothing!? There is NOTHING on cd!?!?!?!>
```

Listing 5: The $PATH environment variable.

3.6 Builtin bash commands.

Although the vast majority of commands you'll be using are actually executable binaries located somewhere within the `$PATH`, there are a handful of commands which are *built into* Bash and other shells.

```
# As we've previously seen, there is no actual executable for "cd":
$ which cd
<nothing shows up>

# This makes sense, since the current working directory ("pwd") is something
# we'd expect to be remembered by Bash itself, so changing the current working
# directory would have to be done within Bash too, NOT using a separate binary.

# Alongside "cd", there's also "pushd" and "popd".
# (originally features introduced in the "csh" shell, and later aped by Bash)
# "pushd" is like "cd" but it remebers every step you take.
$ mkdir -p ./test1/test2/test3
$ pushd test1; pushd test2; pushd test3
$ pwd
/home/you/test1/test2/test3

# After some "pushd"-ing, you can always "popd" to go one level back.
$ popd; popd
/home/you/test1

# Note that "popd" does NOT simply "cd ../".
$ pushd test2/test3
/home/you/test1/test2/test3
$ popd; pwd
# NOTE: pops us back up to "test1". (pun very much intended)
/home/you/test1

# Another really cool and useful builtin is 'alias', which allows you to...well
# ...alias a command (single word!) to another command like so:
$ alias lsroot="ls /"
$ lsroot
<whatever you have in your / directory>

# You can review Bash's builtins any time by reading their manual ("man") page.
# One cool thing worth noting is that there's builtins named "if" and "while",
# which has nothing to do with anything, and totally isn't foreshadowing...
$ man builtins
<a scary wall of text will appear: arrow keys to navigate; "Q" to exit>
```

Listing 6: Showcase of builtin Bash commands.

3.7 Files, directories, and links.

Although, as we'll see later, UNIX systems subscribe to the "everything is a file" paradigm, we'd like to take some time to explain the 3 basic filetypes you'll be interacting with the most:

```
# First and foremost, directories, which are quite self-explanatory:
$ mkdir testdir
$ file testir
testdir/: directory
# Running "ls -l" will show us more info on files, including their type.
$ ls -l
...
drwxrwxr-x  3 <you> <you>   4096 Apr 14 19:49 testdir
...
# Note the starting "d": this indicates the item is a directory and we can "cd":
$ cd testdir

# Regular files are also self-explanatory:
$ echo "Hello, World!" > file.txt
$ file file.txt
$ file file.txt : ASCII text
$ ls -l file.txt
-rw-rw-r-- 1 <you> <you> 14 Apr 15 07:05 file.txt

# Note the leading "-", indicating it's a regular file. Do also note, however, that
# "regular files" refers to the Filesystem concept of a file, and nothing about
# the format of a file. In this sense, the following are also "regular files":
$ file $(which ls)
# "ELF" stands for "Executable Linkable Format", which is the EXE format on UNIXes.
/usr/bin/ls: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), ...
$ ls -l $(which ls)
# Binary executables are also "regular files", so they start with "-" as well.
-rwxr-xr-x 1 root root 138208 Feb  7  2022 /usr/bin/ls

# Anything that is a file in a directory (images, videos, libraries, games, etc...)
# will all count as "regular files".

# It's also worth mentioning "hidden" files and directories.
# There is literally nothing special them whatsoever, they just happen to have
# their name start with ".", which makes Bash and other executables ignore them.
# NOTE: "~" is a special alias for your home directory.
$ cd ~
$ ls
<note whatever stuff you have in your home dir>
$ ls -a
.  ..  .bash_history  .bash_logout  .bashrc ...
# See, even "ls" is in on it, and we need to ask it to list ALL files with "-a".

# "." is the current directory, and ".." its parent, so you can move up a dir by:
$ cd ../
$ ls .
<the contents of the parent directory>
```

Listing 7: Files and directories showcase.

```
# The final filetype we'll review now is the symbolic link. (aka "symlink")
# You can think of symlinks as a "pointer" to another file or directory which
# transparently acts like the thing it's pointing to.
$ ln -s file.txt file_txt.symlink
$ file file_txt.symlink
file_txt.symlink: symbolic link to file.txt

# NOTE: if you try to "cat" the link, it will actuall "cat" the file it's pointing to.
$ cat file_txt.symlink
Hello, World!
$ ls -l file_txt.symlink
# "ls" correctly identifies it as a symlink with the leading "l".
lrwxrwxrwx 1 <you> <you> 8 Apr 15 07:12 file_txt.symlink -> file.txt
# NOTE: both the symlink file, and the original can be deleted independently of
# each-other, which can potentially lead to "broken symlinks":
$ rm file.txt
$ file file_txt.symlink
file_txt.symlink: broken symbolic link to file.txt
$ cat file_txt.symlink
# This error message can seem counter-intuitive because the link file still
# exists, it's just that "cat" is failing to open the file it's linking to:
cat: file_txt.symlink: No such file or directory

# There is also a non-symbolic kind of link which is called a "hard link".
# You can think of hard links as simply "two names, exact same file":
$ echo "Hello again!" > file.txt   # Recreate file.txt if you deleted it.
$ ln file.txt hardlink.txt
$ ls -l
# NOTE: they both look like the same file because they are!
-rw-rw-r--  2 <you> <you>   13 Apr 15 07:18 file.txt
-rw-rw-r--  2 <you> <you>   13 Apr 15 07:18 hardlink.txt
# Reading/writing from either will actually be done on the same file.
$ cat file.txt hardlink.txt
Hello again!
Hello again!
# NOTE: there are TWO ">>" symbols, which append to the file instead of overwriting.
$ echo "Adding this to the hard link." >> hardlink.txt
$ cat file.txt
Hello again!
Adding this to the hard link.
# Removing any of the two entries (original or hard link) will still keep the other:
$ rm hardlink.txt
$ ls -l file.txt
```

Listing 8: Symlinks and hard links showcase.

3.8 Glob patterns.

Imagine having a directory with 100+ files that end in ".txt" you want to remove and having to run "rm" on each and every one of them. Well you don't, thanks to Glob patterns.

```
# First, let's create some directories and files to play with:
$ mkdir -p test11/test12
$ echo "file11" > test11/test12/file11.txt
$ echo "file21" > test11/test12/file21.txt
$ echo "nonfile11" > test11/test12/file11.nontxt

# Now let's say we want to see the files of any directory that is in test11.
# We can have Bash complete matching filenames for us using "*".
$ ls test11/*
file11.txt file11.nontxt

# You can see what actually happens behind the scenes if you "echo" it.
# Bash is transparently replacing '*' patterns with whatever it finds
# BEFORE it executes the command.
$ echo ls test11/*
ls test11/test12

# You can prevent this "glob pattern matching" from occuring by using ':
$ ls 'test11/*'
ls: cannot access 'test11/*': No such file or directory

# Also, if bash fails to match the pattern, it'll just pass the '*' as is:
$ ls asdqtedad*dasdcaefa
ls: cannot access 'asdqtedad*dasdcaefa': No such file or directory

# Now we can clearly deduce that Bash is the one doing the substitution,
# and actual executables such as "ls" have no clue what to do with "*".

# Now let's try to specifically get the "*.txt" ones:
$ ls test11/*/file*txt
test11/test12/file11.nontxt  test11/test12/file11.txt  test11/test12/file21.txt

# No good, we need the "." in ".txt" too:
$ ls test11/*/file*.txt
test11/test12/file11.txt  test11/test12/file21.txt

# How about the ones that start in 1X specifically though ? (pun indented)
# You can replace only one character with the '?' glob:
$ ls test11/*/file1?.txt
test11/test12/file11.txt

# We can also use a "glob expansion" with this syntax:
$ ls test11/test12/file{1,2,3}1.txt
# NOTE: Bash trusts the user and expands these even if the files don't exist:
ls: cannot access 'test11/test12/file31.txt': No such file or directory
test11/test12/file11.txt  test11/test12/file21.txt
```

Listing 9: Bash glob patterns.

3.9 Bash's configuration files.

```
# First and foremost, remeber that cool "alias" builtin?
$ alias lsroot='ls /'

# If you were to ever open another shell, or God forbid, reboot the system,
# you will notice that our nice alias is gone :(
# Fortunately, there is a special hidden file in our home directory named
# ".bashrc". The "rc" at the end stands for "Run Commands", which is a very
# dumb name for indicating this file is full of commands which should be run
# *whenever Bash starts up*. Let's check the file out with the "nano" editor:
$ nano ~/.bashrc

# Let's take a moment to notice some of the cooler defaults in Ubuntu's bashrc:
...
# This makes "ls" color directory names and special files differently:
alias ls='ls --color=auto'
...
# This horrendous "PS1" variable is actually the Bash command prompt.
# Backslashed things like "\u" are auto-replaced by Bash with your username etc.
# The awful "\[\033[01;32m\]" things are actually special codes which change
# the color of the terminal!
PS1='\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ '

# Let's have some fun and add our cool "lsroot" alias and change the prompt.
$ nano ~/.bashrc
<at end of file, write the following>
alias lsroot='ls /'
PS1="\u@\h is in \w :: "
<save the file and exit>

# NOTE: for the changes to take effect, we must "source" (aka re-run) .bashrc:
$ source ~/.bashrc
you@yourpc is in yourpwd :: lsroot
<lists your / directory contents>

# To revert the changes, re-edit "~/.bashrc" and remove the two lines we added
you@yourpc is in yourpwd :: nano ~/.bashrc
you@yourpc is in yourpwd :: source ~/.bashrc
$ # back to normal, amen...

# Other Bash config/static files you can check out:
# File which saves your command history for you to "Ctrl^R" later.
$ cat ~/.bash_history
# Directory with bash scripts which contain completion commands.
$ cat /etc/bash_completion.d
# Files which run every time a user logs in/out of a session.
$ cat ~/.bash_login ~/.bash_logout
# A "legacy config file" read by both Bash and other shells:
$ cat ~/.profile
# When in doubt:
$ man bash
```

Listing 10: Bash configuration files.

3.10 Process input and output

We've all used files streams in C before, but did you know all the `*printf` calls below are functionally identical?

```c
#include<stdio.h>
#include<string.h>

#define SIZE 100

int main() {
    // While you're probably used to using 'fprintf()' for writing to files,
    // the below is exactly identical to 'printf()'. In fact, 'printf()' is the
    // one which is ape-ing on 'fprintf()', as all 'printf()' does is forward
    // args to 'fprintf(1, ...)', just have a look at the code:
    // https://github.com/lattera/glibc/blob/master/stdio-common/printf.c#L33
    //
    // The '1' represents the File Descriptor of the standard output stream.
    // Every process' standard output stream is '1'.
    fprintf(stdout, "Hello fprintf'd World! Printed to %d\n", stdout);
    printf("Hello printf'd World!\n");

    // '2' is the standard error stream (a separate stream for printing errors)
    fprintf(stderr, "Hello fprintf'd World! Printed to %d\n", stderr);

    // Similarly, '0' is the standard input stream (aka your keyboard)
    // The below are similarly identical:
    char name1[SIZE], name2[SIZE];
    printf("Give us your name: ");
    scanf("%s", name1);
    printf("Give us your name again: ");
    fscanf(stdin, "%s", name2);
    printf("%s == %s: %d\n", name1, name2, strncmp(name1, name2, SIZE));

    return 0;
}
```

Listing 11: Output streams showcase in C.

Every process ever started on a Linux system has at least 3 I/O streams just for it:

- `stdin`: stands for Standard Input (File Descriptor 0). When running a program in bash, `stdin` is basically directly hooked up to your keyboard.
- `stdout`: stands for Standard Output (File Descriptor 1). Whenever your program `printf`'s anything, it goes to the OS-managed `stdout` buffer of your running program, and eventually is streamed to your screen.
- `stderr`: stands for Standard Error (File Descriptor 2). This is just meant as a separate output stream for you to print error messages to, and is separated to avoid having the outputs and errors of your programs intermixed. (They will still appear inter-mixed in Bash, but we will see how to separeate them ourselves shortly.)

```
# Firstly, let's list the contents of our root.
$ ls /
<your root stuff>

# Saving the stdout of a command to a file is a simple as writing "> file.name":
$ ls / > root.txt
# You can use "cat" to print the contents of a file:
$ cat root.txt
<your root stuff>

# You can also use "<" (other direction!) to have "cat" receive files as stdin:
$ cat < root.txt
<your root stuff, again>

# We can also cut the file middle-man and just link "ls"'s standard output to
# "cat"'s standard input using the "|" ("pipe") operator.
$ ls / | cat

# Notice something weird? Running "ls" puts the files on one line, but
# saving it to a file puts one filename per line; IS THIS A MAD HOUSE!?!?!
# Well no, it's actually the "ls" binary being able to tell that its output
# stream is stdout, which is presumably a terminal with a human looking at it,
# so it tries its best to arrange the items in nice columns for us.
# As soon as "ls" can tell that its output is being redirected to a file,
# it will just put everything on a separate line.
# What about the colors? Well you see, the colors are put by "ls" automatically
# based on some internal rules, as well as the $LS_COLORS variable:
$ echo $LS_COLORS
rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;33;01:

# See those weird ':'-separated "ln=01;36" entries? That, for example, tells
# "ls" that if a file is a link (created using the "ln" command), then "ls" should
# display the color code "1;36" around it. Let's try the color ourselves!
# Note the "-e" argument which stops "echo" from consuming the backslashes itself.
$ echo -e "\e[1;36mLight Cyan Text"
Light Cyan Text

# These color codes are interpreted by the Terminal app itself!!!
# "ls" is smart enough to know when it's printing to a terminal emulator which
# is able to handle color or not, but we can force it to color just to see:
$ ls --color=yes > file.txt # "cat file.txt" will be colored
$ ls --color=yes | cat      # will be colored
$ echo $(ls --color=auto)   # will be colored

# What happens if we don't wanna save the output of "ls" anywhere and just
# trash it? There's a special pseudo-file created and managed by the OS
# called "/dev/null", which "eats up" any data you feed to it WITHOUT saving it!
$ ls / > /dev/null  # Won't show anything, "ls" output went bye-byte.
```

Listing 12: Bash stdin/stdout showcase. (1)

Now let's try to play around with `stderr` for a bit.

```
# What about listing both our root and some random non-existent file name then?
$ ls / /file-does-not-exist > /dev/null
ls: cannot access '/file-does-not-exist': No such file or directory

# What's up with that!? Well, what's happening is that "ls" is printing errors
# to its stderr output stream, NOT to its stdout output stream.
# Our ">" only redirects stdout, to redirect stderr, we'll need to use "2>"
# notation, which redirects standard error (file descriptor 2) to the file.
$ ls / /file-does-not-exist 2> /dev/null
<contents of your / dir>

# Well damn, that now makes stdout not go to /dev/null anymore, and now we
# see the contents of out "/". No worries though, we can join both together:
# "&>" notation will tell Bash to join the stdout and stderr streams together.
$ ls / /file-does-not-exist &> /dev/null

# You can even specify which stream (1 or 2) should be redirected into which.
# This will redirect stdout into stderr, and so will NOT reach "cat" at all,
# and as a result, "file.txt" will be empty, because "cat" did not receive any
# data through stdin, and thus "cat" did not write anything to its stdout (file.txt)
$ ls / /file-does-not-exist 1>&2 | cat > file.txt
$ cat file.txt  # Empty!
# If we were to redirect stderr (2) into stdout (1), "cat" would then receive through
# its stdin both the combined stdout and stderr of "ls", which will then go in the file.
$ ls / /file-does-not-exist 1>&2 | cat > file.txt
$ cat file.txt  # both root contents, and error message.

# NOTE: this will mean the stream separation will NOT apply any more:
$ ls / /file-does-not-exist &> file.txt
# Will show both error and output messages now, and it's impossible to deduce
# which message came from which stream, so don't do that unless you have to!
$ cat file.txt
```

Listing 13: Bash stdout/stderr showcase. (2)

3.11 Pre-defined variables built into Bash.

Bash has quite a handful of **build-in variables** which provide information on the Bash proces for the user:

```
# First off, we've already seen the $PATH variable, which contains a
# ":"-separated list of system paths that Bash expects to find commands in.
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:...

# We've also seen the $PS1 variable, which defines the format of the prompt:
$ echo $PS1

# There's variables which indicate the User and Group Bash is running as:
$ echo $USER
$ echo $UID
$ echo $GROUPS

# Here's the ID of the Bash process. (they're both equivalent)
$ echo "$BASHPID"
9919
$ echo "$$"
9919

# Bash even knows where you live:
$ echo $HOME
# And also where you are whenever you go:
$ echo $PWD
# And of course, Bash stalks everything you do too:
$ echo $HISTFILE   # Path to the Bash history file where you're being recorded.
# You can always ask Bash to stop without needing a restraining order though:
$ HISTSIZE=0
<congratulations, your UP arrow key is now useless in Bash.>

# Weirder ones also include:
# Number of commands run so far. Run it multiple times and see!
$ echo $LINENO
# Different random number every time.
$ echo $RANDOM
# Will hold the stack of dirs when you pushd/popd.
$ echo $DIRSTACK
```

Listing 14: Builtin Bash variables.

3.12 Process IDs. (PIDs)

In this section, we'll be showing how PIDs work by nesting some Bash shells.

```
# As previously discussed, Bash is, and this is true: also a program.
# So what would happen if we were to try to search for it?
$ which bash
/usr/bin/bash

# *heavy breathing* we HAVE TO try this:
$ bash
$

# Nothing happened eh? Well, such is life, guess I'll just "exit" this
# shell and go about my day...
$ exit
exit
$

# What? I am still in Bash despite exiting?! It's A MAD HOUSE, I TELL YOU!!!
# No it isn't, what actually happened is that we launched a *second* Bash,
# and all we did was "exit" out of it, which returned us to the first Bash
# We can check what processes our *current* user session is running with "ps":
$ ps
    PID TTY          TIME CMD
   5721 pts/0    00:00:00 bash
   5738 pts/0    00:00:00 ps

# Or you can always check the $BASHPID or $$ variables:
$ echo "$BASHPID"; echo $$
5721
5721

# Now let's open a new shell and re-check.
$ bash
$ ps
    PID TTY          TIME CMD
   5721 pts/0    00:00:00 bash
   5739 pts/0    00:00:00 bash
   5745 pts/0    00:00:00 ps

# Bingo, we can clearly see a second "bash" process with ID '5739',
# alongside the O.G. '5721'. We are in a second Bash shell now!
# Sweet! Let's try to mess with the parent process now!
$ echo "Screw you, parent shell!!!"
Screw you, parent shell!!!
$ kill -SIGKILL 5721
<the Window completely closed and we are left staring at our own reflection>
<maybe picking on parent processes wasn\'t such a good idea after all...>
```

Listing 15: Bash multiprocessing basics.

3.13 Background jobs and signals.

```
# First off, if you ever find a command is taking too long (like this damned
# "sleep" command) Ctrl+C will have Bash stop it:
$ sleep 30
<press Ctrl+C to stop the sleep>

# If, instead, you'd like to pause the current command so you can resume it
# later, you can press Ctrl+Z, and Bash will ask Linux to pause the process:
$ sleep 30
<press Ctrl+Z to pause the process>
[1]+  Stopped                 sleep 30

# Bash has now registered "sleep 30" as job #[1], and paused it.
# You can bring any paused job "back into the foreground" with "fg":
$ fg 1
sleep 30  # "fg" will always tell you what command it brought up.
<seemed to stop instantly>

# Hey, how come the "sleep" finished so fast after we resumed it?
# This is actually a side-effect of how "sleep" is implemented: it doesn't
# "actively sleep" for 30s, but instead, it just waits for the OS to notify it
# when 30s have passed. (and the OS itself never sleeps, obviously)
# We can check this by running two separate sleeps:
$ bash -c "sleep 15; echo mid; sleep 15; echo done"
<wait 15+ seconds, then Ctrl+Z to pause the *separate* Bash shell with the sleeps.>

# As you can see, the first sleep exists instantly because it was always
# timer-based instead of wait-based, but the "bash" shell  is also paused.

# How about actually running something in the background? Just put "&" at the end:
$ bash -c "sleep 10; echo I was running this whole time" &
[1] 15488   # This is the job ID (1), and the new process' PID. (15488)
<after 10s, the message should be echoed in your terminal>

# You can also bring running background jobs to the foreground too with "fg" again:
$ bash -c "sleep 10; echo Please leave me alone" &
fg
bash -c "sleep 10; echo Please leave me alone"
Please leave me alone

# You can brutally murder any background job you don't like with "kill":
$ sleep 10000 &
# Note the process ID of the "sleep". You can also check it with the "ps" command.
[1] 15493
$ jobs
[1]+  Running                 sleep 1000 &

$ kill -SIGINT 15493
# NOTE: if you run "jobs" fast enough, you can see the process being interrupted:
$ jobs
[1]+  Interrupt               sleep 1000
```

Listing 16: Background jobs and signals.

So now we know that signals are something you can pass to running processes, but how do they work exactly?

Signals are POSIX OS features, which means that processes (like Bash) needs to ask the OS to send another process (e.g. "sleep") a certain signal code (e.g. `SIGINT`). The receiving process needs to define a **signal handler** (aka C function) for each signal type it wants to support, and the OS will then execute that piece of code whenever a signal is received. This means that we must rely on executables having proper signal handlers defined, or use the most powerful weapon we have against them: the `SIGKILL` signal.

```
# Note that signals like "SIGINT" ("interrupt") is kinda like a message we're
# sending to another process: we're asking it nicely to "interrupt its life"
# for us, but it is ultimately the process receiving the signal which decides
# what to do about it. This applies to all signals except one: SIGKILL.
$ sleep 10000 &
[1] 15498

# SIGKILL will have the OS (not the target process) instantly terminate it.
$ kill -SIGKILL 15498
$ jobs
# Running "jobs" fast might still show the dead body of the process you just killed:
[1]+  Killed                  sleep 10000

# There are a bunch of other notable signals to send too. You can have "kill" list them:
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
...

# Notable ones include:
# SIGTERM is a stronger kind of "SIGINT" which asks the process harder to terminate.
$ kill -SIGTERM 15498
# SIGSTOP and SIGCONT is to tell a process to stop/continue whatever it's doing.
$ kill -SIGSTOP 15498
# SIGQUIT/SIGCORE ask the process to quit but also produce a core dump (for debugging)
$ kill -SIHQUIT 15498
```

Listing 17: Signal types and SIGKILL.

3.14 Environment variables and parent/child processes.

Every process ever launched on a Linux system (except for the Linux kernel and init system 2.4.1), MUST have been launched at the request of another process.

This implies two things:

- the child process **inherits the environment** of the parent process. This includes things like the Current Working Directory ("pwd"), and Environment Variables.
- the parent process *knows* the PID of the child process thanks to how the fork syscall's return value works. This means that the parent cand ask the OS to send so-called **signals** (e.g. SIGKILL) to the child process.

Let's do some more nested shell shenanigans to showcase this:

```
# One of the most counterintuitive aspects of Bash is that, out of all things,
# "pwd" (which prints the current directory) is actually an executable:
$ which pwd
/usr/bin/pwd

# How come "pwd" knows what the working dir of the parent Bash shell is?
# Well...it doesn't, it's just that every executable Bash calls will
# automatically inherit Bash's current working directory, so "pwd" is actually
# showing us what the working directory of "pwd" is. (we just know it's the same)

# First, let's define a variable and start second shell.
$ VAR="test variable"
$ bash
$ ps
    PID TTY          TIME CMD
   5891 pts/0    00:00:00 bash
   5960 pts/0    00:00:00 bash
   5966 pts/0    00:00:00 ps

# Now let's print the variable out:
$ echo $VAR
<nothing>
# Whoa what happened to my variable?! Let's exit the second shell and recheck:
$ exit
exit
$ echo $VAR
test variable

# So it still existed in the first Bash, but the second Bash never got it.
# You can have Bash *export* variables to *any* child process it starts
# by using the "export" keyword when declaring it:
$ export VAR="test variable meant for the second shell"
$ bash  # open second Bash again.
$ echo $VAR
test variable meant for the second shell

# Alternatively if you only want to pass a variable to *one* process,
# you can prepend its declaration to the command name.
$ VAR="local var meant for the 3rd shell" bash
$ echo $VAR
local var meant for the 3rd shell
$ exit       # exit the 3rd Bash and recheck.
$ echo $VAR  # back to the second shell now.
test variable meant for the second shell

# You can always check what environment variables you have defined by using:
$ env
<list of env vars you were using this whole time without knowing it>
```

Listing 18: Bash parent/child process variable showcase.

Chapter 4

Regular Expressions

4.1 Why do we need them?

Regular expressions (a.k.a. RegExes) are a handy tool when manipulating strings. You can think of a RegEx as a pattern to be matched when searching or replacing parts of strings.

4.2 Cheatsheet

The patterns are nothing else but a combination of identifiers.

4.2.1 Anchors

∧ - marks the beginning of a line

"∧The" - will match strings starting with **"The"**

$ - marks the end of a line

"end$" will match strings that end with **"end"**

"∧The end$" - matches all strings that are exactly **"The end"**

4.2.2 Quantifiers

Quantifiers are a way of specifying the count of a specific token in a pattern.

* - matches **zero or more** of the preceding token

\+ - matches **one or more** of the preceding token

? - matches **zero or one** of the preceding token

{n} - matches **exactly n** of the preceding token

{m,n} - matches **between m and n** of the preceding token

{m,} - matches **m or more** of the preceding token

Examples

abc* matches a string that has ab followed by **zero or more c**
abc+ matches a string that has ab followed by **one or more c**
abc? matches a string that has ab followed by **zero or one c**
abc{2} matches a string that has ab followed by **2 c**
abc{2,} matches a string that has ab followed by **2 or more c**

abc{2,5} matches a string that has ab followed by **2 up to 5 c**
a(bc)* matches a string that has a followed by **zero or more copies of the sequence bc**
a(bc){2,5} matches a string that has a followed by **2 up to 5 copies of the sequence bc**

4.2.3 Capturing groups

a(bc) - parentheses create a **capturing group** with **value bc**

a(?:bc)* - using ?: we **disable the capturing group**

a(?< foo **>bc)** - using ?< foo > we put a name to the group

This operator is very useful when we need to extract information from strings or data using your preferred programming language. Any multiple occurrences captured by several groups will be exposed in the form of a classical array: we will access their values specifying using an index on the result of the match.

If we choose to put a name to the groups (using (?< foo >...)) we will be able to retrieve the group values using the match result like a dictionary where the keys will be the name of each group.

4.2.4 Character classes

. - matches **any character**

\d - matches a **single character** that is a **digit**

\w - matches a **word character** (alphanumeric character plus underscore)

\s - matches a **whitespace character** (includes tabs and line breaks)

\t - matches **tabs**

\n - matches **newline characters**

\r - matches **carriage returns**

User-defined character classes

You can define your own character classes using square brackets.

For example:

[abc] - matches either **a, b or c**

[a-z] - matches any lowercase letter in the alphabet

[a-d] - matches any lowercase letter between a and d

[A-Z] - matches any uppercase letter in the alphabet

[a-zA-Z] - matches any letter in the alphabet

[a-dr-z] - matches any letter between a and d or between r and z

[0-9] - matches any digit (identical to \d)

[2-5] - matches any digit between 2 and 5

Negated character classes

\D - matches a **single character** that is **not a digit**

\W - matches **anything but a word character**

\S - matches **anything but a whitespace character**

[∧abc] - matches **anything but a, b or c** (the negation is expressed using the ∧ character)

4.3 Escaping special characters

When building RegExes, some characters need to be escaped using a backslash in order to not interpret them as a special characters and just match them instead.

For example, use '\.' instead of '.' if you want to match a '.' character instead of interpreting '.' as any character.

E.g. **'∧[a-zA-Z]\w+\.com$'** can be used to match website names that are registered under the .com domain.

4.4 Greedy and lazy match

The quantifiers (*** + {}**) are greedy operators, so they expand the match as far as they can through the provided text.

For example, **"\[.+\]"** matches **"[start] content [end]"** in ***"Let's catch some tags: [start] content [end]"***. In order to catch only the tags we can use a ? to make it lazy. Notice that the square brackets have been escaped in order to capture the actual [] characters instead of interpreting the content as a character group.

Therefore, **"\[.+?\]"** will only capture **[start]** and **[end]** instead of including the content between the tags.

4.5 Look-ahead and Look-behind

d(?=r) - positive look-ahead - matches a **d** only if is **followed by r**, but **r will not be part of the overall regex match**

(?<=r)d - positive look-behind - matches a **d** only if is **preceded by an r**, but **r will not be part of the overall regex match**

d(?!r) - negative look-ahead - matches a **d** only if is **not followed by r**, but **r will not be part of the overall regex match**

(?<!r)d - negative look-behind - matches a **d** only if is **not preceded by an r**, but **r will not be part of the overall regex match**

4.6 RegEx examples

Let's see some practical examples:

4.6.1 Example 1

```
# First, let's create a new directory for our exercise
$ mkdir regex1
$ cd regex1

# Now, let's create a new file and add some data to it
$ echo "123456789
392948291
321582923
321904984
Not a number
hello" >> phone_no.txt

# Now, let's try changing the phone number format using sed
# we want to change a number like 123456789 to (123)456-789
$ sed -E 's/([0-9]{3})([0-9]{3})([0-9]{3})/(\1)-\2-\3/g' phone_no.txt > phone_no_formatted.txt
$ cat phone_no_formatted.txt
(123)-456-789
(392)-948-291
(321)-582-923
(321)-904-984
Not a number
hello
```

Listing 19: RegEx examples.

Here's a breakdown of how the command works:

- sed is the command to perform text transformations.
- -E enables extended regular expressions in sed.
- **'s/([0-9]{3})([0-9]{3})([0-9]{3})/(\1)-\2-\3/g'** is the substitution pattern. It uses regular expressions to match numbers in the format of 123456789, and captures the first three digits, second three digits, and third three digits separately using parentheses. It then rewrites the matched pattern as (123)-456-789 using backreferences to the captured groups.

4.6.2 Example 2

Explanation for listing 20:

- First, we pipe the output of ifconfig to grep
- **"grep -Eo '^[^[:space:]]+|inet ([0-9]{1,3}\.){3}[0-9]{1,3}"** searches for two patterns in the output of ifconfig.
  - The first pattern matches the interface name, which is the first word in each line.
  - The second pattern matches the IPv4 address associated with the interface.
  - The -o option tells grep to only output the matched strings, one per line.
  - **[:space:]** is a POSIX character class that matches any whitespace character, including space, tab, newline, and carriage return (equivalent to \s).
- **"sed 'N;s/\n/ /"** reads two lines at a time and replaces the newline character between them with a space. This effectively joins the interface name and IPv4 address on the same line, separated by a space.

```
# We have a bash script that for some reason wants to print
# the IPv4 address associated with every network interface.
# Let's see how the output of ifconfig looks like
$ ifconfig

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.28.54.81  netmask 255.255.240.0  broadcast 172.28.63.255
        inet6 fe80::215:5dff:fef9:959c  prefixlen 64  scopeid 0x20<link>
        ether 00:15:5d:f9:95:9c  txqueuelen 1000  (Ethernet)
        RX packets 114  bytes 10160 (10.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 13  bytes 1006 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

# Whoa... a lot of stuff...
# What if we wanted to only see the IPv4 address of every interface?
$ ifconfig | grep -oE '^[^[:space:]]+|inet ([0-9]{1,3}\.){3}[0-9]{1,3}' | sed 'N;s/\n/ /'
eth0: inet 172.28.54.81
lo: inet 127.0.0.1
```

Listing 20: Further RegEx examples.

4.6.3 Useful resources

sed manual: `https://www.gnu.org/software/sed/manual/sed.html`

grep manual: `https://www.gnu.org/software/grep/manual/grep.html`

You can see some commonly used RegEx samples here: `https://digitalfortress.tech/tips/top-15-commonly-used-regex/`

Play around with RegExes here: `https://regexr.com/`

Chapter 5

Shell scripting

While going through the numerous Bash listings in this document, you may have found yourself annoyed by the fact that you had to copy and run individual commands over and over, and telling yourself that there must be a better way. Shell Scripting allows you to define (and refine!) Bash commands in files and re-run them at your leisure. In this chapter, we'll be walking you through the basics of automation using Bash.

5.1 Error codes and control flow basics.

While we've already seen that all programs run by Bash are able to print error messages on a dedicated output stream called `stderr` (12), parsing the actual messages in order to programatically determine if a command has failed would be more trouble than it's worth. Fortunately, there was a reason you needed to declare `int main()` in C this whole time: **the return code of the program is expected to indicate whether (and which) error the program has encountered**. Let's see how it works more in-depth:

```
# First, let's get "ls" to fail miserably:
$ ls /non-existent-file
ls: cannot access '/non-existent-file': No such file or directory

# While it's always nice to see an error message as clear and concise as that
# if you're a human, a machine would have a terribly hard time reading it.
# Luckily, Bash always saves the 8-bit unsigned code of the last command as $?
$ echo $?
2

# As we can see, "ls" returned error code 2. Can we get it to return another one?
$  ls --color=objectively-not-a-color-flag; echo $?
1

# Having different error codes helps a lot when cross-referencing the "man" pages.

# So, presuming that we live in a Utopia and all the programs people write are
# infallible and respect all the standards, we can assume anything non-zero is bad.
# Bash is comfortable making that assumption because it provides the built-in
# operators "&&" and "||", which we can use to affect the control flow of commands.

# The "&&" operator will only execute the second command if the first one exit 0's.
$ echo "This echo will exit 0" && echo "So the second echo will execute too"
This echo will exit 0
So the second echo will execute too

$ ls /this-ls-will-exit-2 && so-i-can-type --literally --anything=here
ls: cannot access '/this-ls-will-exit-2': No such file or directory
# NOTE: the bogus second command doesn't even get looked up, let alone executed.

# The "||" operator will only execute the second if the first one exits non-zero.
$ ls /this-ls-will-exit-2 || echo "So this echo gets executed"
ls: cannot access '/this-ls-will-exit-2': No such file or directory
So this echo gets executed

$ echo "This echo will exit 0" || so-this-command --will-never=get-executed
This echo will exit 0
# NOTE: the bogus second command doesn't even get looked up, let alone executed.

# The operators can also be chained together left-associatively:
$ echo 1st || ls 2nd && ls 3rd || ls --color=this-causes-error-code-1
1st
# NOTE: "ls 2nd" was skipped.
ls: cannot access '3rd': No such file or directory
ls: invalid argument ‘this-causes-error-code-1’ for ‘--color’
...

# The final error code of the line will be last one which executed.
# In the above case, it was the last "ls" with code 1, but under the wrong
# circumstances, it can be less clear what the result's gonna be:
$ echo 1st && ls 2nd || echo 3rd || ls --color=this-causes-error-code-1
# You'll have to try it to find out.
```

Listing 21: Bash error codes.

5.2 Everything is a string, and you should `test` it.

```
# You know what would be really handy? If we could check things. Any things.
# For this, there is the dedicated "test" command:
$ which test
/usr/bin/test
$ test "anything"; echo $?
0  # NOTE: any non-zero string counts as a "successful check".
$ test ""; echo $?
1  # NOTE: testing an empty string is always an "unsuccessful check".

# Checking if a string *is* actually empty requires negating the check:
$ test ! ""; echo $?
0

# You can compare strings with "==" and "!=".
$ test "yes" == "yes"; echo $?
0
$ test "yes" != "no"; echo $?
0

# NOTE: ALL Bash values are just strings. Bash has ZERO concept of ANY other type.
# DO NOT EVER assume anything is ever anything more than a string!

# Feast your eyes on these abominations:
$ test false && echo "Counterintuitive af."
# NOTE: the *STRING* "false" is non-empty, so it "counts as true" (exit 0).
Counterintuitive af.
$ test 0 && echo "A bit less counterintuitive but still..."
A bit less counterintuitive but still...  # Same goes for the non-zero *string* "0".

# This also applies to numbers which look like strings.
# "==" and "!=" only act like a dumb "strcmp"!
$ test 05 == 5 && echo "Nothing happens because the STRING 05 != the STRING 5."

# To actually compare numbers, you need to use the dedicated -eq/ne/-lt/-gt.
$ test 05 -eq 5 && echo "Always compare numbers with -eq..."
Always compare numbers with -eq...
$ test 1000 -gt 100 && echo "Ofc 1000 > 100!"
Ofc 1000 is greater than 100!
$ test 1000 -lt 100 || echo "But 1000 < 100 is wrong!"
But 1000 < 100 is wrong!

# More than just strings and number-like strings, "test" can also test files/dirs:
$ test -e /         && echo "-e checks if a path exists (file, directory, link, anything)"
$ test -f ~/.bashrc && echo "-f specifically checks if something is a file"
$ test -d /         && echo "-d specifically checks if something is a directory"
$ test -L /bin      && echo "-L specifically checks if something is a *symbolic* link"

# Check out the man page for "test"s full functionality!
$ man test
```

Listing 22: Showcasing all of Bash's datatypes: the string.

5.3 Defining and using variables.

As any good pretend-programming-language, Bash has the concept of **variables** (which are **ALL strings**). The variable system is obviously great for saving the **string** output of commands.

```
# First and foremost, variables are declared as follows:
$ VARIABLE_NAME="variable value"
$ echo $VARIABLE_NAME
variable value

# DO NOT put spaces around the "=". This will make Bash think that the
# VARIABLE_NAME is a command, and the "=" should be its first argument.
$ VARIABLE_NAME = variable value
VARIABLE_NAME: command not found

# DO NOT put "$" when declaring a variable! Bash will replace it with NOTHING!
$ $VARIABLE_NAME="variable value"
# NOTE: "$VARIABLE_NAME" got replaced with nothing, and bash tried to run "=var...".
=variable value: command not found

# DO NOT forget the quotes for the value if it has spaces! Bash will think you
# only meant to declare the variable "locally" for this one command.
$ VARIABLE_NAME=variable value
value: command not found
# You can verify this is the correct behavior by running the following
$ VARIABLE_NAME=local_variable bash -c 'echo $VARIABLE_NAME'  # NOTE: single quotes!
local_variable

# Now that we know how to declare variables, let's use them!
$ echo "$VARIABLE_NAME"
variable value
# Works fine, but what if we wanted to literally print "$VARIABLE_NAME"? Single quotes:
$ echo '$VARIABLE_NAME'
$VARIABLE_NAME

# Careful though, it's extremely easy to misuse variables too:
$ PREFIX="Hello, "
$ echo "$PREFIXWorld!"
!
# Whenever inlining variables, always use "${VAR}" notation so Bash knows where names end:
$ echo "${PREFIX}World!"
Hello, World!

# NOTE: IMPORTANT!!!
# One last extremely useful variable tip is variable expansion syntax:
$ echo "${VAR1:-default_str}"  # Prints $VAR1 if it's non-empty-string, else "default_str".
default_str
$ echo "$VAR1"  # NOTE: still empty
$ echo "${VAR1:=default_str}"  # Same as above, but also sets VAR1=default_str
```

Listing 23: Variables in Bash.

5.4 Using variables with commands.

Of course, Bash's variable system wouldn't be very useful without the ability to use the variables in commands, or maybe even use variables...*as* commands.

```
# Before anything, please take care when inlining variables with spaces in commands:
$ TOLS="/bin /usr"
$ ls $TOLS
<works as expected, as the command expands to "ls /bin /usr">
$ ls "$TOLS"
# NOTE: failed miserably, since there is no directory named "/bin " (note the space)
ls: cannot access '/bin /usr': No such file or directory

# As mentioned, Bash replaces ALL variables with strings BEFORE running it.
# This means that there's nothing stopping us from...you know...running a varibale!
$ LSROOT="ls /"
$ $LSROOT
# Nice, but if we want to list /bin?
$ $LSROOT bin
ls: cannot access 'bin': No such file or directory
# Oh right, the above expands to "ls / bin", we need it to be joined.
$ ${LSROOT}bin
<lists /bin>

# Running all these "ls" commands is really tiring though, it would be nice if
# we could just save the it for later, you know?
$ LISTEDROOT=ls /
-bash: /: Is a directory   # NOTE: bash tried to run "/" with "LISTEDROOT=ls".

# Well we can, using the magic of command substitution:
$ LISTEDROOT="$(ls /)"  # Clearer to anyone.
$ LISTEDROOT=`ls /`     # 100% equivalent, but less easy to notice the backticks.
$ echo "$LISTEDROOT"
bin boot dev ...

# Nice, now we never have to list the root directory ever again ;)
```

Listing 24: Using variables with and as commands.

5.5 The start of every script.

There are a handful things every Bash script should probably include all of the time.

```
#!/bin/bash
# ^ the above line starting with "#!" (no spaces!) and then a path to an
# executable is called a "Shebang!". If (and only if!) placed on the FIRST line
# of a text file which is executable (chmod +x file.txt), then Bash will
# automatically execute that program and pass the file as an argument.
# This works with any scripting language, not just Bash. For example, if this
# were a Python 3 file, we would have written "#!/usr/bin/python3".

# NOTE: IMPORTANT!!!
# -e will make Bash stop the script the instant a command returns a non-zero
# exit code, as the default Bash behavior is to always run the whole script.
set -e

# NOTE: IMPORTANT!!!
# -u makes Bash immediately exit the second you reference a $VAR that's undefined.
# There is never any reason to not pre-define variables to "", so always do it!!!
set -u

# pipefail will make command pipelines ("cmd1 | cmd2 | cmd3") return the exit
# code of the first (leftmost) command that fails, as by default Bash will only
# return the exit code of the last command ("cmd3"), regardless of "cmd{1,2}".
set -o pipefail

# -x will make Bash print every command *before* it runs it, but *after*
# Bash does the variable substitution. That can prove extremely useful when
# attempting to debug scripts with very nested functions and variables.
set -x

# Because we set '-e' above, as we always should, it means that if any
# command errors out before our script gets the chance to clean up any
# temporary files, those files will pollute our system forever more.
# The "trap" builtin allows you to "register" a thing for Bash to do
# at a given point, but you have to declare it at the start of the file!
trap "echo Script exiting unexpectedly, performing cleanup now; rm /tmp/some-junk" EXIT

# The above will execute whenever any action leads to the exitting of the shell.
# You can also define different handlers for different signals the script might receive.
$ trap "echo This will get triggered if you Ctrl^C the script" SIGINT
$ trap "echo And this one when any one of these signals hits" SIGHUP SIGQUIT SIGABRT SIGTERM

# You can safely write the worst possible script you can now, your mind at ease
# that any and all abd events will be "trap"'d and the necessary cleanup done.
```

Listing 25: Bash script start.

5.6 Pretending Bash is a programming language.

```
# NOTE: "if/elif/else", "while", "for", and "case" are all Bash builtins.
# Please run "man builtins" for full details.

# First and foremost, the humble "if":
if test -e /doesnt-exist
then
    echo "We were not ready at all for that file to exist ngl."
elif test -e /bin/getting-warmer
    echo "A bit more ready for this one, but still surprised."
else
    echo "We're always ready for the base case!"
fi

# NOTE: if/elif/while all TREAT THEIR CONDITION AS A COMMAND, so:
if "this should check if the string is non-empty"; then
    echo "it was non-empty"
fi
# This will have Bash tell us that that string is *NOT* a command with this:
this should check if the string is non-empty: command not found

# If you ever find yourself needing an infinite loop, you could:
while test "this string check will always succeed"
do
    echo "ran once"
    # NOTE: you can "break" out of "while" and "for" loops:
    break
done

# Other alternatives is to use the *EXECUTABLE PROGRAMS* called "true" and "false".
$ which true false
/usr/bin/true
/usr/bin/false
$ true && echo "This will always work cause 'true' ALWAYS returns exit code 0."
$ false || echo "Same for this, as 'false' will ALWAYS retun exit code 1."

# The "for" loop will *NOT* execute its condition. Instead, all a "for" loop
# does is "split" its condition. (based on spaces/tabs/newlines by default)
for item in this will NOT get executed, but split into words based on spaces
do
    echo "${item}"
done

# Side-NOTE: The way "for" splits strings is based on the "$IFS" variable,
# which is a string of chars "for" should split by. For example:
IFS="5"  # If you run this, you'll notice the next "for" will split by '5'.

# To run a command and capture its output, use "$(cmd)" or "`cmd`":
for number in `seq 0 10`; do
    echo $number
done
```

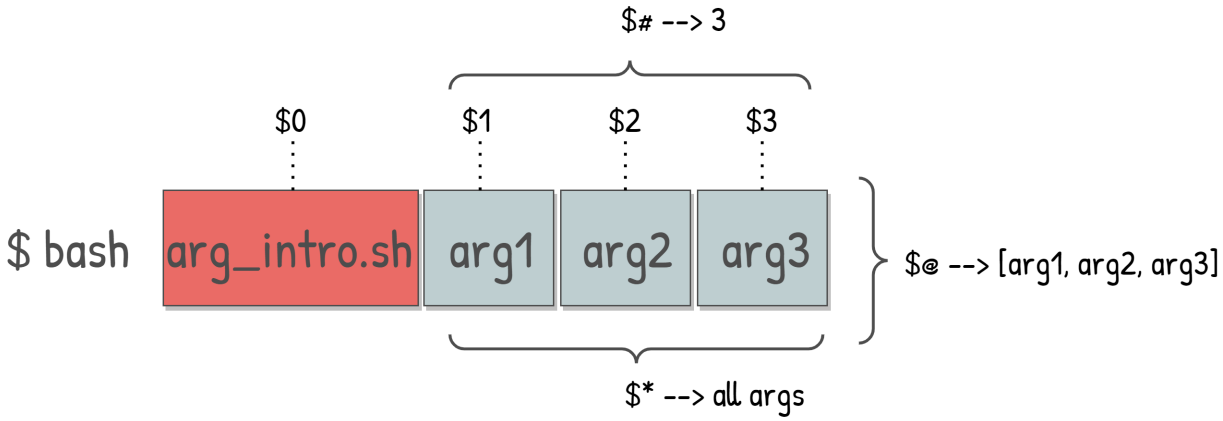
Listing 26: Bash programming constructs.

5.7 Calling and passing arguments to scripts.

```
#!/bin/bash

# NOTE: never forget these:
set -eux
set -o pipefail

# Let's assume this is a script which we want to make callable like:
# bash /path/to/script --bool-flag --str-flag some-str arg1 arg2
# We'd like to have the position of "--bool-flag" and "--str-flag val" not matter.

# NOTE: pre-define vars because we used 'set -u'.
BOOLF=""; STRF=""; ARG1=""; ARG2=""

# First off, here's how to access the arguments:
$ echo $0           # $0 will always be the name of the script.
$ echo $@           # $@ contains ALL arguments (excluding script name)
$ echo $#           # $# contains the number of arguments (excluding script name)
$ echo $1 $2 $3     # $N will be the positional argument on the Nth place.

# That's cool and all, but what are we supposed to do if we can only access the
# arguments positionally? For this, we need to process the args out of order
# using the the "shift" command.

# Instead of "while", you could also use 'for $arg in "$@"', but then "shift" won't
# work, so if you have args like "--str-flag some-str", you NEED to use a while:
while [ "$#" -gt 0 ]; do
    case "$1" in
    # --bool-flag is a simple "boolean" flag:
    --bool-flag) BOOLF="YES" ;;  # NOTE save whatever value you'd like.
    # For "--str-flag some-str", we must also "shift" to get "some-str".
    --str-flag) shift; STRF="$1" ;;
    # Any "non-flag" args should be treated as ordered positional args:
    *) if [ ! "$ARG1" ]; then
           ARG1="$1"
       elif [ ! "$ARG2" ]; then
           ARG2="$1"
       else
           echo "Too many positional arguments."
           exit 1
       fi ;;
    esac
    # Shift will remove the first arg from $@, and also decrement $#.
    shift
done
if [ ! "$ARG1" ] || [ ! "$ARG2" ]; then
    echo "not enough positional arguments"
    exit 2    # NOTE: different error code here for debugging later.
fi

echo "BOOLF='${BOOLF}'; STRF='${STRF}'; ARG1='${ARG1}'; ARG2='${ARG2}'"
```

Listing 27: Passing and parsing arguments in scripts.

Figure 5.1: Bash script argument referencing.



5.8 Defining functions in Bash.

Bash also lets us define functions within scripts, which can very simply be thought of as "sub-scripts": they can be individually called by name, will have their own set of arguments (`$`), but unlike calling an external script, function will have access to the non-`exported` variables of the "enclosing script".

```
#!/bin/bash

# NOTE: never forget these:
set -eux
set -o pipefail

# Declare a var to play with later:
SCRIPT_VAR="some global variable"

# You can declare a function with the "function" keyword, and do NOT need
# to specify its number of arguments, since all functions can take any #args.
function my_first_function {
    # NOTE: all of the below arguments are the function's, NOT the script's!
    echo $@        # $@ contains ALL arguments (excluding function name)
    echo $#        # $# contains the number of arguments (excluding function name)
    echo $1 $2 $3  # $N will be the positional argument on the Nth place.

    # NOTE: $0 will always be the name of the script tho, NOT the function.
    # This is to allow function to "re-call" the parent script if needed.
    echo $0

    # NOTE: the function has access to variables defined outside its scope.
    echo "External var is: ${SCRIPT_VAR}"

    # You can avoid polluting the script variable namespace with "local":
    local LOCAL_VAR="some local variable"

    # WARN: DO NOT use "exit" in functions, as it will exit the whole script!
    # Instead, you can "return <code>", which also returns a code.
    return 0
}

# You can simply call the function like any other command now:
my_first_function arg1 arg2 arg3

# Apart from the obvious code reuse benefits, the most common usecases for declaring
# functions are the need for "recursive code" (you'll see later), and "trap" behavior.
function cleanup_on_exit {
    echo "script $0 is exitting. Cleaning up all temporary files."
    rm /tmp/some-temp-files-for-our-script || true
    echo "script $0 cleanup was successful."
}

# You can "register" this function to be executed the second the script exits with "trap":
trap cleanup_on_exit EXIT
```

Listing 28: Bash funtions showcase.

Chapter 6

Users and permissions.

In this chapter, we'll be discussing all about the security model for files and programs on POSIX-based systems.

6.1 Users, groups, fin.

The old UNIX security model is extremely simple, yet extremely effective. There are exactly two entities at play:

users : entities meant to actually use the computer. They can log in, create and delete files and directories, and some users can interact with the system.

groups : entities whose only purpose is "banding up" users. Groups are *NOT* accounts which can log in or affect the computer in any way, only Users can. Groups allow users to share selected files and execute selected programs *from* each-other (but not *as* each-other!) if the Users share the same Group.

6.2 File ownership and permission.

The are three basic "dimensions" which need to be considered when judging the legality of an action in UNIX:

- **ownership**: which specific User and Group (both singular!) own the item.
- **filetype**: what type of filesystem object is the action being performed on:
  - **regular files**: anything from an ASCII text file, executable binary files like "ls", and even the Linux kernel itself.
  - **directories**: self-explanatory.
  - **hard links**: considering they're basically like giving two names to the same file, it's important to remember that changing the permissions of on will also reflect on the permissions of the other.
  - **symbolic links**: the actual permissions check on links is done against the permissions of the regular files/directories that they point to.
- **permissions**: whether or not a User can perform a given **action**:
  - **read**: whether someone can "cat" a file or "ls" a directory.
  - **write**: whether or not someone can modify an existing file, or "act on" a given file entry *in a given directory*. File creation, deletion, and renaming all require **write** access to the directory the file is in, *not* the file itself!
  - **execute**: whether or not someone can run a given executable (or script, since at the end of the day scripts are also run by executables), or whether a directory can be used as the Current Working Directory. (aka "cd" into it)

Let's see what this looks like in the shell:

```
# First off, never forget who you are:
$ whoami
<your username>

# Good, and also never forget what defines you:
$ cat /etc/passwd | grep "$(whoami)"
<you>:x:1000:1000:<you>:/home/<you>:/bin/bash

# Yep, your whole identity: User and Group IDs, your favorite shell,
# even the place you call home...just one single line in a file.
# Well at least the computer doesn't know your password...

# NOTE: the Ubuntu installer usually defaults the ID of the "main user" created
# by the system to 1000, and it also creates a group for it with the same ID/name:
$ cat /etc/group | grep "$(whoami)"
<you, but as a group>:x:1000:

# Bash also knows on behalf of which User and Group it is running as:
$ echo "$USER ID: $UID; Groups: $GROUPS"
<you> ID: 1000; Groups: 1000

# As such, any file you create from Bash will inevitably inherit your UID/GID:
$ echo "test" > file.txt
$ ls -l file.txt
-rw-rw-r-- 1 <you> <you, but as a group> 5 Apr 14 19:27 file.txt

# Also, every command Bash starts will also inherit the UID/GID. Try with a new bash:
$ bash -c 'echo "test" > file2.txt'; ls -l file2.txt  # Same as above.

# This is not something Bash decides, it's the rules of the system: any process
# or file created by a Bash shell will share the UID/GIDs of the parent.
# This is how "whoami" and "groups" work: they actually tell you *their* UID/GID,
# but because we know they're the same as Bash's, we can use that info.
```

Listing 29: Showcasing file types and permissions.

Figure 6.1: `/etc/passwd` user definition format.

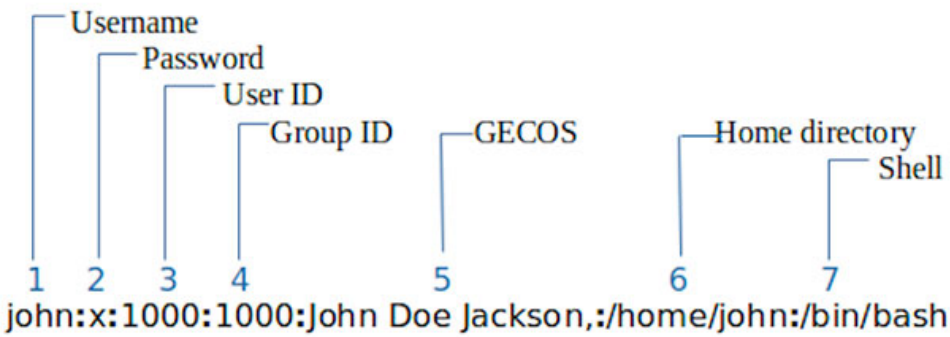
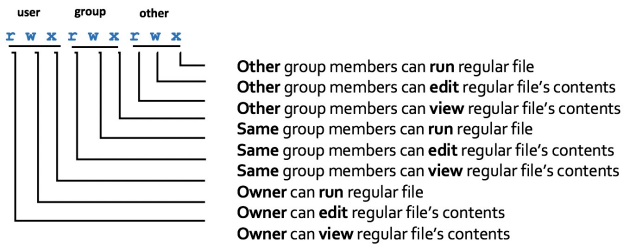
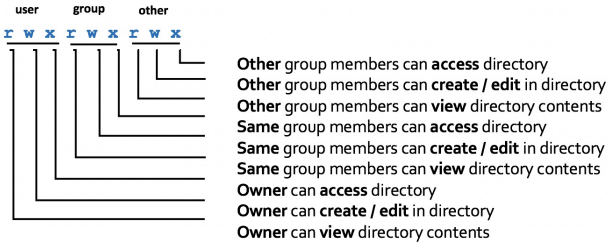


Figure 6.2: File and Directory permissions mask explanations.






```
# As shown in the above diagrams, those weird "rwx" bits on files in the output of
# "ls -l" actually represent the permissions certain entities have with said file.
$ echo 'echo Hello from the testscript!' > testscript.sh
$ ls -l
-rw-rw-r--  1 <you> <you, but as a group>   32 Apr 15 15:46 testscript.sh
# The first 'rw-' represents the permissions the owner of the file (you)
# have over the file. (r = read, w = write, '-' should be x = execute)

# Well damn, looks like this script isn't executable...which makes it worthless!
# We can fix that with "chmod", which changes the file's "mode bits".
# Asking chmod to "+x" will tell it to add the executable permission on the file.
$ chmod +x testscript.sh
$ ls -l testscript.sh
-rwxrwxr-x  1 <you> <you, but as a group>   32 Apr 15 15:46 testscript.sh

# Nice, now we can just run the script using "./testscript.sh". But what if we
# want to prevent anyone (including ourselves) from ever seeing this script again?
$ chmod -r testscript.sh
$ ls -l
--wx-wx--x  1 <you> <you, but as a group>   32 Apr 15 15:46 testscript.sh
$ cat testscript.sh
cat: testscript.sh: Permission denied

# It just hit me, the second '-wx' represents the permissions for users who are part
# of the Group which is set on the file. (the groupname is the same as your username)
# Don't trust anyone is a group named after yourself, we should revoke their access:
$ chmod g-wx testscript.sh
$ ls -l testscript.sh
--wx-----x  1 <you> <you, but as a group>   32 Apr 15 15:46 testscript.sh

# Damn, security's a lot of work, it would be nice if we could just run one command
# with like...a single number representing all the mode bits we want to set or somm.
$ chmod 764 testscript.sh
$ ls -l testscript.sh
-rwxrw-r--  1 <you> <you, but as a group>   32 Apr 15 15:46 testscript.sh
# 7 = 111 = rwx = permissions for the User who owns the file. (you)
# 6 = 110 = rw- = permissions for users in the Group set on the file.
# 4 = 100 = r-- = permissions for anyone who isn't even in the Group.
```

Listing 30: Changing regular file permissions with `chmod`.

6.3 Directory and link permissions.

It's worth noting that permissions on directories and links act slightly differently tha regular files.

```
# Let's first create a directory we can use as a guinea pig:
$ mkdir testdir; ls -l
drwxrwxr-x 2 <you> <you> 4096 Apr 15 16:24 testdir

# All good, but now let's make a random file and start removing permissions:
$ echo "I am an innocent bystander." > testdir/testfile
$ chmod -w testdir
$ echo "I am a guilty bystander." > testdir/testfile2
-bash: testdir/testfile2: Permission denied
# Looks like we can't create new files any more, how about renaming existing ones:
$ mv testdir/testfile testdir/testfile2
mv: cannot move 'testdir/testfile' to 'testdir/testfile2': Permission denied
# Damn, now I'm quite frustrated, and feel like rebelling out of spite:
$ rm testdir/testfile
rm: cannot remove 'testdir/testfile': Permission denied

# Ok, so now we know that 'w' permission on directories affects all aspects of
# file/directory creation/deletion within that directory, but what about 'x'?
$ chmod -x testdir; ls -l
dr--r--r-- 2 <you> <you> 4096 Apr 15 16:29 testdir
$ cd testdir/
-bash: cd: testdir/: Permission denied

# Huh, looks like the 'x' permissions on directories just means using it as the
# Current Working Directory for a process. (in this case, our Bash shell)
# The 'r' permissions is less exciting, it just lets you "ls" or "testdir/*.txt".

# For hard links (which is basically just giving two names to the same file),
# the appropriate file/directory permissions apply as we've seen so far:
$ touch file.txt; ln file.txt hardlink.txt; ls -l
-rw-rw-r-- 2 <you> <you, but as a group> 0 Apr 15 16:36 file.txt
-rw-rw-r-- 2 <you> <you, but as a group> 0 Apr 15 16:36 hardlink.txt
# Modifying permissions for one filename will also affect the other, try it!
$ chmod 600 file.txt hardlink.txt

# For symlinks, the permissions work the same way, but only the permissions on
# the link's target file are actually checked:
$ ln -s file.txt symlink.txt; ls -l
lrwxrwxrwx 1 <you> <you, but as a group> 8 Apr 15 16:36 symlink.txt -> file.txt
# Note that symlink permissions will always say "lrwxrwxrwx", but they're irrelevant:
$ ./symlink.txt
-bash: ./symlink.txt: Permission denied
```

Listing 31: Directory and link permissions.

6.4 sudo and SetUID binaries.

You may have noticed that the vast manjority of things on the system belong to some bastard named `root`. That, according to the POSIX standard, is the name of the "admin account" on Linux systems, and so it owns basically everything important on the system. In order to install/update anything though, we'd need to do it as the root user one way or the other:

```
# First off, you can try to "ls -l" around to see what files belong to what users:
$ ls -l /bin
...
-rwxr-xr-x 1 root root      138208 Feb  7  2022 ls

# He's got our boy "ls"! We have to save him somehow!!!
$ mv $(which ls) ~/safety/
mv: cannot move '/usr/bin/ls': Permission denied

# Oh right, we can't move things out of /bin without permissions on it.
# But what's this, one of the files from above had some weird permissions:
$ ls -l $(which sudo)
-rwsr-xr-x 1 root root      232416 Apr  3 18:00 sudo

# See that 's' in the 'rws' permissions group.
# That stands for "setuid": if set, it means that anyone can run that
# executable (sudo in this case) as the owner of the file. This needs to be
# set using "chmod +sx", so that "root" guy must have been a good sport and
# set it for us. Let's try to save our boy "ls" now, but first, a copy:
$ sudo cp /usr/bin/ls /usr/bin/ls-copy
<input your personal password>
$ ls -l /usr/bin/ls-copy
# Well would you look at that, "sudo" (which got executed as root) also executed
# "cp" as root, and so the resulting copied file also belongs to that bastard:
-rwxr-xr-x 1 root root 138208 Apr 15 17:06 /usr/bin/ls-copy

# Screw it, let's just make our getaway.
$ cp /usr/bin/ls-copy ~/ls; ls -l ~/ls
-rwxr-xr-x 1 root root 138208 Apr 15 17:06 /home/<you>/ls

# Damn, it's still got his name on it even after we moved it, "chown" it now:
$ chown $USER:$USER ~/ls
chown: changing ownership of '/home/<you>/ls': Operation not permitted

# Double damn, only the owner of the file can "transfer" its ownership, so we
# need to run the "chown" command as root himself:
$ sudo chown $USER:$USER ~/ls; ls -l ~/ls
-rwxr-xr-x 1 <you> <you, but as a group> 138208 Apr 15 17:06 /home/<you>/ls

# Which users can and cannot use "sudo" is determined by the /etc/sudoers file.
# In most cases, your username won't be explicitly there, but instead your user
# is made part of the "sudo" Group, which make "sudo" allow you to user it.
$ sudo cat /etc/sudoers
...
%sudo   ALL=(ALL:ALL) ALL  # Allow members of group sudo to execute any command.
```

Listing 32: sudo and SetUID.

6.5 Managing Users and Groups.

Creating Users and Groups in UNIX systems is a relatively straigntforward process.

```
# In order to create a group, all you really need is a good name.
$ groupadd coolgs

# Groups are defined in the /etc/group file, one group per line:
$ cat /etc/group | grep coolgs
coolgs:x:1001:

# Creating a user is bit more involved, we need to decide a range of parameters
# beyond just the username, such as the home directory, login shell, and groups.
$ useradd -d /home/epic -G coolgs,sudo -s /bin/bash epic-username
$ cat /etc/passwd | grep epic
epic-username:x:1001:1002::/home/epic:/bin/bash

# Before we can take our shiny new account for a spin, we need to set a password.
# Passwords are hashed and stored in /etc/shadow, let's see how it looks like:
$ sudo cat /etc/shadow | grep epic
epic-username:!:19462:0:99999:7:::

# That "!" indicates the user has no password set, we can update it by doing:
$ passwd epic-username
<input password of your choice, but make it epic>

# Now we can indeed see the ugly hash of the password in /etc/shadow:
$ sudo cat /etc/shadow | grep epic
epic-username:<an truly epic hashed password>:19462:0:99999:7:::

# "su" stands for "switch user", you'll need to input "epic-username"s password:
$ su epic-username
$ whoami
epic-username

# Nice, now that we switched to the most epic user on the system, let's go home:
$ cd ~
bash: cd: /home/epic: No such file or directory

# Of yeah, we didn't actually create a home directory for this account.
# Screw it, an account this epic deserves to call / its home:
$ usermod -d / epic-username
usermod: user epic-username is currently used by process 14587
# Damn, type "exit" to log out of this epic Bash session and re-run the "usermod".

# Once we've had our fun, we can then go ahead and clean everything up:
$ userdel epic-username
$ groupdel coolgs

# You can verify all traces of epicness have left your system by:
$ cat /etc/passwd | grep epic
$ cat /etc/group | grep coolgs
```

Listing 33: User and Group management.

Chapter 7

Getting around the filesystem

This section aims to prove to you the power of Open Source OSes like GNU/Linux: the fact they openly bear their internal guts for anyone to inspect. (and poke!)

7.1 Everything is a file.

Before moving forward, we must internalize one of Linux's core values which it inherited from its Unix ancestry: literally 100% of everything is a file:

```
# The "file" command is unspeakably useful for telling us what files are:
$ file /etc/passwd  # this file defines ALL users on the system (including you!)
/etc/passwd: ASCII text

# There's more just files and directories through, here's that magical
# device which we can redirect output we want to discard to:
$ ls / > /dev/null  # Will print nothing.
$ file /dev/null
/dev/null: character special (1/3)

# Character special devices are "virtual" in that they're managed entirely by
# the kernel, and thus only exist while the system is running. Couple more:
$ file /dev/zero     # character special you can read infinite zeros from
$ file /dev/urandom  # can read infinite pseudo-random numbers from instantly
$ file /dev/random   # can read infinite truly random numbers which require time
# Try all the above with the "od" command, "man od" for more info:
$ od -vAn -N4 -tu4 < /dev/urandom
3834605808

# All your disks and other storage devices are there too as "block devices":
$ file /dev/sda
/dev/sda: block special (8/0)
# And here's some files you can echo random data to control the disk's driver:
$ file /sys/class/block/sda/device/driver/unbind
/sys/class/block/sda/device/driver/unbind:    regular file, no read permission
# Note your VM's root disk might be "/dev/vda" or something else, check with "lsblk".

# Here's the initial RAM disk which contains the kernel drivers on boot:
$ file -L /boot/initrd.img
/boot/initrd.img: ASCII cpio archive (SVR4 with no CRC)
# Here's the Linux kernel you just ran all this on so far:
$ file -L /boot/vmlinuz*
<need password to mess with it.>
/boot/vmlinuz: Linux kernel x86 boot executable bzImage, version 5.15.0-69-generic...

# And last, but not least, here is a symbolic link to the current working
# directory of the shell you're running this in:
$ file /proc/$BASHPID/cwd
/proc/6933/cwd: symbolic link to /where/ever/you/are
$ cd ../..
$ file /proc/$BASHPID/cwd
/proc/6933/cwd: symbolic link to /where/ever/
<PANIC...>
$ cd ../../../../../../../../../../../../
$ file /proc/$BASHPID/cwd
/proc/6933/cwd: symbolic link to /
<you can never escape the root, nor the system..>
```

Listing 34: Everything is a file.

Let's unpack all this weirdness one at a time.

7.2 The places with actually real files I promise.

Most files you'll be "ls"-ing will, thankfully, be perfectly normal files (text or executables or something else), with the odd symbolic link here and there... Let's have a look at the more notable "normal places":

- the directories with **binary executables**:
  - `/bin`: system-standard binaries which are generally useful to all users, contains all our favorites like "ls" and "which".
  - `/sbin`: system-standard binaries for administrative duties, such as "fsck" for checking disk integrity, "mkfs" for formatting new disks, etc...
  - `/usr/bin` and `/usr/sbin`: same principle as above, but usually contain binary executables which were installed later, and cannot be considered "crucial" to the system.
- **library directories**: these contain linkable or callable libraries (e.g. every 'include¡somm.h¿' you ever did) in different flavors: `/lib`, `/lib32`, and `/lib64`. (for general, 32-bit, and 64-bits respectively)
  - `/etc`: configuration files for basically everything.
- `/home/<username>`: directories where each normal user gets to do
- `/root`: this is the home directory of the "root" (admin) user.
- `/media` and `/mnt`: directories meant to mount and access external storage like CDs or USBs.
- `/var`: semi-permanent "app data" files programs freely create and manage for themselves. Check out all the logs in `/var/log`!

7.3 /proc (Procfs)

Procfs, which you can find under `/proc`, is a complete lie. The files and directories within are auto-populated by the kernel, and whenever you read/write to them, it does not go to a real file on any disk, or even to some "fake RAM filesystem". Instead, the **files in /proc are a direct window** into the state of the kernel and its processes.

```
# First off, there's a couple of files with information about the system itself:
# CPU info like model and the number of cores.
$ cat /proc/cpuinfo
# Runtime stats like core percentage used.
$ cat /proc/stat
# Memory (RAM) usage statistics.
$ cat /proc/memory
# Memory (RAM) stats like which areas are allocated.
$ cat /proc/zoneinfo
# List of filesystems supported by the kernel. See "proc"?
$ cat /proc/filesystems

# Apart from the top-level files which contain system information, there's a
# bunch of randomly-numbered directories, what's up with those?
# Those are actually directories with properties relating to each and every
# process which is running on the system in the instant your "ls" command runs.
# You ca verify that running "ls" multiple times will change at least one PID.
$ ls && ls && ls
<at least one process ID changes between the lists>

# Anyway, let's go check out the /proc entry for our shell:
$ cd /proc/$BASHPID
# Here's a symlink to the process (aka Bash) executable:
$ file exe
exe: symbolic link to /usr/bin/bash
# Here's the arguments Bash received on startup:
$ cat cmdline
# Here's all the resource limits set by the kernel for this process:
$ cat limits
# Here are all the environment variables the process has currently set:
$ cat environ
# Here's stats on how many characters the process has read/written.
$ cat io; cat io; cat io  # Run it multiple times to see it grow.

# Here's a list of file descriptors currently used by our Bash.
$ ls fd  # I dare you to find a process without the 0, 1, and 2 FDs!
# Also, notice that all the FDs are links to the same Pseudo-TTY. ("pts")
# That is the virtual console session you are using right now.
$ file fd/*
file fd/*
fd/0:   symbolic link to /dev/pts/0
fd/1:   symbolic link to /dev/pts/0
fd/2:   symbolic link to /dev/pts/0
fd/255: symbolic link to /dev/pts/0
fd/3:   cannot open 'fd/3' (No such file or directory)

# What's up with that funky "No such file or directory" FD? I'm glad you asked!
# That's actually the open File Descriptor of the "fd/" directory which Bash
# needed to open in order to auto-complete the 'fd/*' glob pattern.
# After Bash completes the pattern, it closes the directory (thus closing fd/3)
# and the passes the whole list (including fd/3) to the "file" command.
# You can verify this by running a separate Bash process to complete the glob:
$ bash -c "file ./fd/*"  # Should not show a "No such file or directory" error.
```

Listing 35: Looking into /proc.

7.4 /sys (Sysfs)

Similar to `/proc` being a window into the state of the system's processes, the equally fake (and equally fun) `/sys` is a window into the state of the system's devices through the Sysfs pseudo-filesystem. The fake files in `/sys` are actually even more fun, since beyond just reading the system's state, you can also alter it by echo-ing the right info to the right files.

```
# First off, let's see what we got:
$ cd /sys; ls
block  bus  class  dev  devices  firmware  fs  hypervisor  kernel  module power

# Way more organized than /proc, let's take them one step at a time.

# Contains symlinks to all block devices. (aka disk or disk-like devices)
# The contents of block devices will be explored deeper in the Storage chapter.
$ file ./block/*
...
# NOTE: there's my root disk, but yours may be named differently.
./block/sda:    symbolic link to <the real awful-looking ../devices/... path>
...

# /bus directory organizes devices by bus type. Let's look for my disk:
$ file bus/scsi/devices/*
...
./bus/scsi/devices/<your_disk_bus_id>:    symbolic link to </sys/devices/...>
...

# /class directory organizes devices by human-friendly device class names:
$ file class/scsi_disk/<your_disk_bus_id>
./class/scsi_disk/<your_disk_bus_id>:    symbolic link to </sys/devices/...>

# /dev derectory organizes your devices by Major:Minor kernel IDs.
$ ls dev/*/

# /devices directory is the real Kernel-driver-based names of the devices.
# All the above directories just link to some device somewhere in here.
$ ls devices/*/

# /fs directory contains all the FSes mounted by the system.
$ ls fs/*/

# /kernel, /module, and /power contain data about the kernel, its modules,
# and the power state of the various devices of this machine.
$ ls kernel; ls module; ls power

# Don't believe these files affect your real system? Just run this:
$ echo 1 > /sys/block/<your root disk>/device/delete
```

Listing 36: Looking into /sys.

Joke aside, Sysfs is extremely handy when you just wanna control your devices from Bash. In fact, read all about how you can control your Raspberry PI GPIO pins entirely from Bash!

Chapter 8

Package Management

Thanks to the Open Source nature of GNU/Linux's ecosystem, the biggest hurdle users have in obtaining and using software is that they need to compile it too...

Fortunately, all Linux distros worth their salt will come pre-equipped with a **Package Management System** (2.4.2) to help with finding, downloading, installing, and removing pre-built packages for us.

8.1 The question of format.

The choice of package management format and system is one of the most influential decisions made during the building of any Linux distribution, and the space is dominated by the following main "lineages":

Format	First Released in	Packaging Tool	Main Package Manager	External Manager
`.deb`	Debian	`dpkg`	`apt`	`add-apt-repository`
`.rpm`	Red Hat	`rpm`	`yum/dnf`	`yum-config-manager`
`.pkg.tar.xz`	Arch	`pacman`	`pacman`	`yaourt`
`.snap`	Ubuntu	`snap`	`snap`	N/A

Regardless of the chosen format however, the end result is the same: the files associated with the application(s) the user wants to install are unpacked from whatever package format is being used (they're all basically archives anyway), and placed in the correct places for said application to work.

8.2 Meeting the Package Manager.

Because we are using Ubuntu, which is a Debian-based distribution, we will be using "apt" throughout the following examples, as "snap"s are relatively new and barely adopted outside of Ubuntu, and thus generally less useful to know.

```
# NOTE: you will need to be admin whenever running "apt", so use "sudo apt ...".
# First off, let's ask "apt" to see if we have any updates available:
$ sudo apt update
# NOTE: your repository may be different.
Hit:1 http://ro.archive.ubuntu.com/ubuntu jammy InRelease
Get:2 http://ro.archive.ubuntu.com/ubuntu jammy-updates InRelease [119 kB]
...
Fetched 2442 kB in 1s (2155 kB/s)
...
Reading state information... Done
13 packages can be upgraded. Run 'apt list --upgradable' to see them.

# Whoa that's not a very lucky number of updates to have pending, we should update:
$ sudo apt list --upgradeable  # lists which packages can be upgraded
$ sudo apt upgrade
...
# NOTE: yours may be different.
The following packages will be upgraded:
  python3-tz sosreport
2 upgraded, 1 newly installed, 0 to remove and 11 not upgraded.
...
Need to get 351 kB of archives.
After this operation, 92.2 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y  # Press "Y" and Enter.
...
Get:1 http://ro.archive.ubuntu.com/ubuntu jammy-updates/main amd64 python3-tz all 2022.1-1ubun
...
Preparing to unpack .../python3-tz_2022.1-1ubuntu0.22.04.1_all.deb ...
Unpacking python3-tz (2022.1-1ubuntu0.22.04.1) over (2022.1-1ubuntu0.22.04.0) ...
Setting up python3-tz (2022.1-1ubuntu0.22.04.1) ...
...

# Wow, apt took care of everything for us! But how does "apt" know where to
# look for updates? (more precisely, those "ro.archive.ubuntu.com" repo URLs?)
# The "/etc/apt" directory contains all of "apt"s configs, including its sources:
$ cat /etc/apt/sources.list
...
deb http://ro.archive.ubuntu.com/ubuntu jammy main restricted
# deb-src http://ro.archive.ubuntu.com/ubuntu jammy main restricted
...

# Let's see if those archives contains the best and most important package ever written:
$ sudo apt search "^sl$"  # NOTE: the "^" and "$" are a regex, and "sl" is our package name.
$ sudo apt install sl
$ sl  # < This is very important.

# Once you've had your fun, you can remove "sl" with:
$ sudo apt remove sl
$ sudo apt autoremove  # and delete any packages *not* used by other packages:
```

Listing 37: Familiarizing ourselves with apt.

8.3 Investigating package contents.

All Linux packages are, at the end of the day, just collections of pre-built files and their associated *metadata* like authors, dependencies, and data on where each file should go.

```
# First, let's install another absolutely indispensable package:
$ apt-get install -y cowsay
# Try it out, it's important:
$ cowsay "Hey there, $USER"

# Nice, it's been installed, but how and where, I wonder? Check the package cache:
$ file /var/cache/apt/archives/cowsay*
/var/cache/apt/archives/cowsay_3.03+dfsg2-8_all.deb:
Debian binary package (format 2.0), with control.tar.xz, data compression xz

# Let's go into /tmp to copy and mess around with the package:
$ pushd /tmp && cp /var/cache/apt/archives/cowsay* ./cowsay.deb
# Unpack it with the "ar" archive unpacking utility.
$ ar vx cowsay.deb
x - debian-binary
x - control.tar.xz
x - data.tar.xz

# First off, "debian-binary" will contain the format version.
# This lets "dpkg" know how to treat older and newer packages correctly.
$ cat debian-binary
2.0

# Let's check out "data.tar.xz", which looks like a TAR archive:
$ file data.tar.xz
data.tar.xz: XZ compressed data, checksum CRC64
# "ar" won't work, we need "tar" to unpack it:
$ tar -xvf data.tar.xz
./usr/games/cowsay
...

# Indeed that had all our files! What about "control"?
$ tar -xvf control.tar.xz
./control  ./md5sums
# "control" has all the metadata like its deps, we can see it's a Perl script!
$ cat ./control
Package: cowsay
Version: 3.03+dfsg2-8
Depends: libtext-charwidth-perl, perl:any
...

# And "md5sums" has all the checksums for the files in "data.tar.xz".
$ cat ./md5sums
4a3fcaf4ae6c1758b55cc04b075a4007  usr/games/cowsay
dd9004601a67345d76e91ed232676f60  usr/share/cowsay/cows/apt.cow
...
```

Listing 38: Finding deb packs and unpacking.

Chapter 9

All about storage

In this chapter, we will explore the extremely high number of hoops the OS needs to jump through to safely store your diskpics on the disks you picked.

9.1 Organizing bits on tapes.

Storage has come a very long way from the days of tapes, but the fundamental problems of retrieving data which is everternal to the CPU and RAM are unavoidable: storage is considerably slower than RAM because it requires calling a device through a bus, interpreting its messages through a driver, and then presenting a logical file/directory to the User when, in fact, **all storage can be considered one contiguous tape**.

Here's all the "layers" required to actually commit your diskpics to disk:

- **busses**: because disks are devices which "live outside of the CPU of our computer", the CPU has to "call" the device and communicate with it somehow. This is done through the uses of special System Buses on the motherboard. As we'll soon see, Linux addresses disks through Bus IDs.
- **drivers**: this may be obvious, but once the CPU manages to communicate with the disk via a bus, it still has to know *how* to tell it to do stuff. This is done via drivers which are loaded into the Linux kernel through the initrd 2.4.1.
- **partitioning schemes**: because disks are fickle things, and you wouldn't want data corruption, it would be handy if we could *abstract* single disks into multiple, smaller virtual disks. This is exactly what partitions (like the C: or D: drives on Windows) are for, and the OS manages paritions of top of the disk.
- **file systems**: neither disks nor partitions have any idea what files are, all they can do is store or load the data D in disk location X, nothing more. This means that the OS needs to maintain a "set of rules" for storing/accessing files, which lives as code within the OS, called "file system code".
- **file metadata**: storing my diskpics is cool and all, but I want to be able to be know when each diskpic was taken. For this, most file systems reserve some space for "metadata" (means "data about data"), such as the date the diskpic was made, which user the diskpic belongs to, and more...

We'll be going through each logical layer as we go up the "storage stack".

9.2 Busses and external I/O.

The need for busses is obvious: we have a CPU and want to connect things to it. Most systems have the busses handled transparently by the motherboard, with various bus controllers and bends everywhere.

The most popular buses you might have inadvertedly been using are:

- **SPI** (Serial Peripheral Interface): is an old bus which links *all* devices in *series*. This means that there is a "ring of devices" and commands from the CPU to device N will take N shifts of the "ring buffer" for the message to reach the device in question.
- **SCSI** (Small Computer Serial Interface): similar to SPI in that it is serial, but provides more power and data throughput, and thus is extensively used in old spinny hard drives.
- **ATA and SATA** ([Serial] AT Attachment): the new post-2000 standard for storage device interconnectivity in PCs.
- **USB** (Universal Serial Bus): the industry standard designed around compatibility and operability. USB itself has many revisions, the latest common one being USB 3.X, whose ports are USB-C ports (NOTE: the port shape and bus protocol are NOT strictly linked, USB 3.X runs in both USB3 and USB-C ports)
- **Thunderbolt**: made by Intel almost exclusively for Apple devices, with the main selling point being increased speeds over USB, but at the cost of needing an Intel CPU chip.
- **PCI** (Peripheral Component Interconnect (sic)): developed in the late 90s, it and its descendant PCIe are the industry-standard for PC parts.

A note on things which are NOT busses:

- NVME (Non-Volatile Memory express): is a "protocol" for *driving* (not connecting to!) devices (usually SSDs). The actual disk access still happens through existing buses like PICe.
- mSATA (mini SATA): a smaller port, but still using the SATA bus.
- M.2 (smaller mSATA): refers to the size specifications of a port called "M.2", but the underlying busses M.2 ports support are PCIe, SATA, and even USB3.

Let's see how we can query the storage device and driver stack in Linux.

```
# NOTE: this example was run on an Ubuntu VM using a SATA virtual bus. YMMV.
# First, let's list our disks (represented as "block devices", or "blk").
$ lsblk
NAME                      MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
fd0                         2:0    1     4K  0 disk
loop0                       7:0    0  63.3M  1 loop /snap/core20/1828
...
sda                         8:0    0   100G  0 disk
  sda1                      8:1    0     1M  0 part
  sda2                      8:2    0     2G  0 part /boot
  sda3                      8:3    0    98G  0 part
    ubuntu--vg-ubuntu--lv 253:0    0    49G  0 lvm  /
sr0                        11:0    1  1024M  0 rom

# We can see that our root ("/") is mounted in a disk named "sda".
# The "sda" nomenclature comes from SATA Disk A. If this VM had a second
# disk, it would be named "sdb". Don't worry about "sdb{1,2,3}" for now.

# All devices should live in the "/dev" directory, so let's check for "sda":
$ file /dev/sda
/dev/sda: block special (8/0)
```

Listing 39: Showcasing device and bus controls. (1)

```
# It exists! And it's indeed a "block special device", but what is it?
# The (8/0) indicates the Major and Minor numbering scheme the Linux kernel
# uses to identify devices for itself. Rebooting the machine might change it!

# We would need to check every possibile bus to find what it's on: SCSI, PCI, and USB.
# However, the kernel maintains a list of "better-named" symbolic links for us
# to work with. Note the '-l' to tell "ls" to show us where links go.
$ ls -l /dev/disk/by-id
ls -l /dev/disk/by-id
total 0
# NOTE: you may see multiple links to your disk, that's because the kernel
# has multiple ways of addressing devices, and it's showing you all of them.
...
... scsi-14d53465420202020731e0c2c3707cd4b9ea59492c88c926e -> ../../sda
...

# Now that I know my disk is using SCSI, I can list all devices for that bus:
$ lsscsi   # or "lspci" or "lsusb".
[1:0:0:0]    cd/dvd  Msft     Virtual CD/ROM   1.0   /dev/sr0
[2:0:0:0]    disk    Msft     Virtual Disk     1.0   /dev/sda
```

Listing 40: Showcasing device and bus controls. (2)

9.3 Storage drivers.

As mentioned, busses only allow the CPU to talk to devices, but *what* the CPU tells devices is up to the device driver in the Linux kernel.

Device drivers are, obviously, device-dependent, so attempting to list them here would be futile. One thing worth exploring though is how things look like under the hood of a Hyper-V virtual machine. (which is what this sample wa run on)

```
# NOTE: this example was run on an Ubuntu VM using a SATA virtual bus. YMMV.
# First off, run "lsblk" and check what device your "/" MOUNTPOINT is in:
$ lsblk
NAME                      MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
...
sda                         8:0    0   100G  0 disk  # NOTE: you need this name
  sda1                      8:1    0     1M  0 part
  sda2                      8:2    0     2G  0 part /boot
  sda3                      8:3    0    98G  0 part
    ubuntu--vg-ubuntu--lv 253:0    0    49G  0 lvm  / # NOTE: / = root FS

# Now that we have the device name ("/dev/sda" in this example), we can start
# poking "/sys" which is a pseudo-filesystem (those files aren't real, they are
# placed there by the kernel while the system is running) with system info.
$ cd /sys/class/block

# Now that we are the Block Device class subdir of /sys, we can look for "sda":
$ file sda
sda: symbolic link to ../../devices/LNXSYSTM:00/LNXSYBUS:00/PNP0A03:00/device:07/
      /VMBUS:01/00000000-0000-8899-0000-000000000000/host2/target2:0:0/2:0:0:0/block/sda

# Whoa, what a mouthful, that is a link to the actual place "sda" lives in
# /sys/devices, where all the devices are organized in a "tree" of their bus
# and device model IDs. Let's poke around:
$ cd sda
$ ls

# Lotta stuff here, but here's the more notable files:
# Show the Minor:Major numbers used by the kernel to identify this device.
$ cat ./dev  # Mine were "8:0".
# Whether the device is removable (1) or not (0):
$ cat ./removable  # Mine was not (0).
# Lots of links everywhere, this is just a link to /sys/class/block where we started.
$ file subsystem
subsystem: symbolic link to ../../../../../../../../../../../class/block
# "device" is another link to the actual device info.
$ file device
device: symbolic link to ../../../2:0:0:0
# Whether the disk is running.
$ cat device/running
# The disk model:
$ cat device/model  # "Virtual Disk" for me, as it's a VM.
# The disk manufacturer
$ cat device/vendor  # "Msft" for me, as my VM is running on Microsoft Hyper-V.
# Whether the device is busy (1) or not (0).
$ cat device/device_busy  # Mine was lazy (0).
# How many serial-connected (ring) SCSI devices away this device is:
$ cat device/scsi_level  # Mine was 6 devices away.

# And many more, poke around as much as you'd like!
```

Listing 41: Exploring block devices drivers.

9.4 Partitioning schemes.

Now that we now how to talk to a disk over a bus, and also know what to actually say to it thanks to the driver... now what?

Well now we can do whatever we want with the disk, and the first and most obvious thing any logical person would do is to... invent a whole system for spliting the real disk into multiple smaller disks called a **partitioning scheme**.

While it may initially seem counter-intuitive, the "need" for partitions is something the world has inherited from the 90s when filesystems would corrupt themselves every Tuesday, and the only way to somehwat protect yourself is to have multiple separate filesystems on the same disk. (but more than 4 lol)

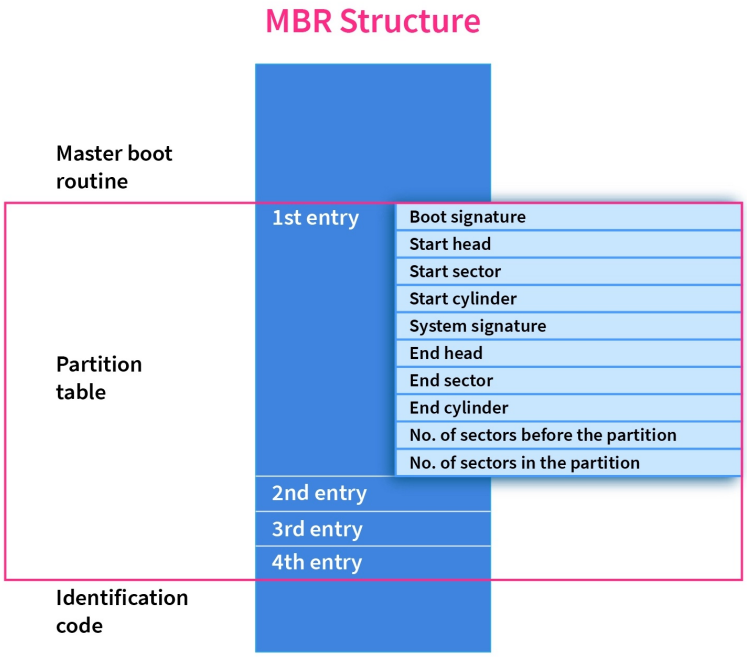
9.4.1 Master Boot Record (MBR) partitioning.

The Master Boot Record (MBR) partitioning scheme was introduced in the 80s and is honestly as simple as it gets.

It is a 512-byte structure which lives at the *very beginning* of the disk (address 0x00 (zero)), and contains nothing more and nothing less than:

- **Executable Code area** (446 bytes): this area contains actual machine code meant to be executed by the `firmware` when the machine starts. In practice, this area is a place where GRUB 2.4.1 places a small piece of code to mount/execute the rest of GRUB which lives in "/boot" on Linux systems.
- **4 (yes, four) partitions entries** (4x16 bytes): here lie general information about the 4 partitions this MBR defines, such as where is the start and end sectors of said partitions on the disk.
- **2 bytes for a special "signature"**: this fixed signature (`0x55AA` or `0xAA55`) which is read by the Firmware to:
  - determine if there's an actual MBR there: since disks have no clue what data they're storing, we need to decide for ourselves whether a disk has a MBR we want to read or not.
  - determine the endiannes (i.e. byte reading order) for this MBR, as different CPU architectures have different byte ordering standards.

Figure 9.1: Master Boot Record header structure.



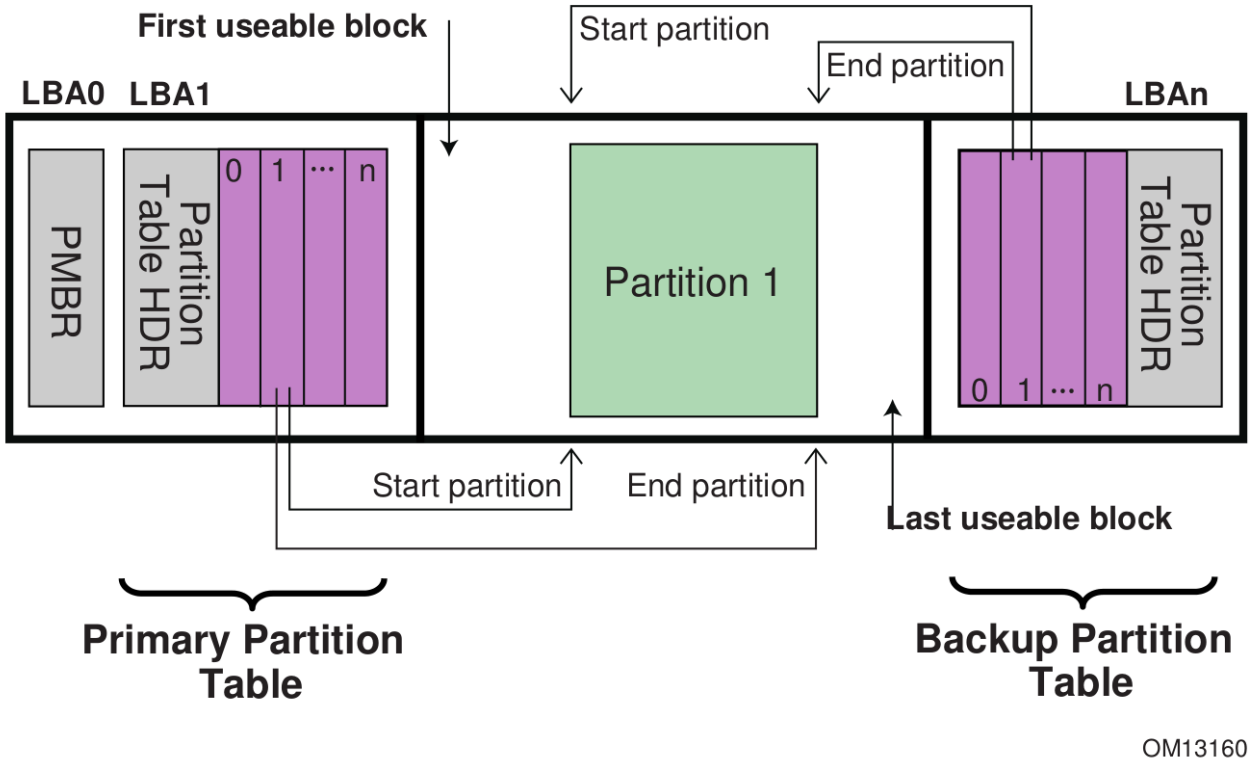
One thing worth noting is the existence of so-called "Logical Paritioning schemes" under Windows systems using MBR which allows one to create more than 4 partitions by having one partition be further partitioned, leading to partition-ception. These schemes have been almost entirely phased out in favor of GPT. (segway...)

9.4.2 GUID Partition Table (GPT) partitioning.

The **G**lobally **U**nique **Id**entifier **P**artition **T**able scheme (GPT), is a direct upgrade to MBR in every way:

- **can declare up to 128 partitions**: already won over MBR.
- **any partition can hold another GPT table**: 128 ^ 128 partitions anyone?
- **reliability built-in**: the GPT is stored both at the start and the end of disks, and so system recovery from partial disk damage is possible.
- **needs firmware support!!!**: this means that for many old laptops/PC motherboards, GPT simply isn't an option, so back to exactly 4 MBR partitions they go...

Figure 9.2: GPT Partitioning layout structure.



9.5 Filesystems.

Now that our disk is all nicely partitioned, we can start worrying about how to organize all our files and directories on them. (remember, disks have no concept of files, just of bytes in cells)

For this, we (well, technically the kernel) would need to define a *system* of rules for how to store the files.

Any good ***filesystem*** must consider the following aspects:

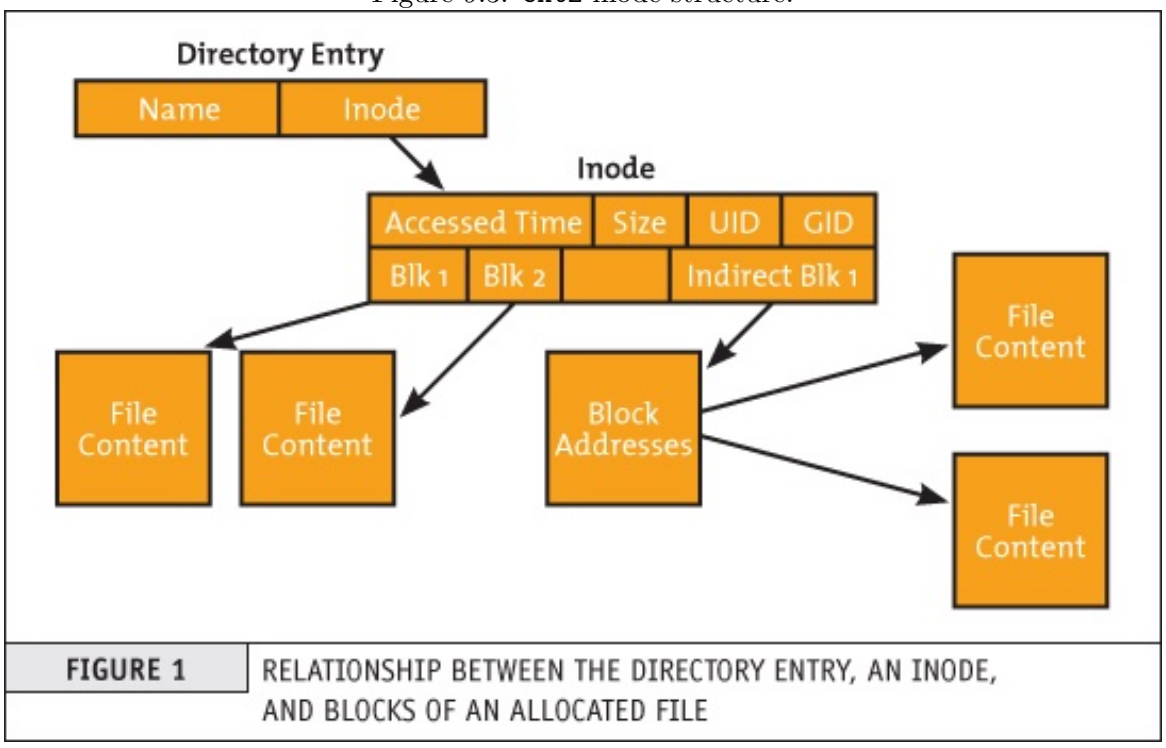
- **types of files**: a filesystem must define a set of possible file types under it. On POSIX systems, that's basically just regular files, directories to organize those files in, and symklinks to provide some indirection. (3.6)
- **file metadata**: meaning "data about data", file metadata includes all the things related to the file which are not the file data itself, such as:
  - **name**: yes, if you think about it, the name of a file has nothing to do with the actual data within it, and thus counts as metadata.
  - **permissions**: file ownership (owning User/Group) and permissions are also metadata.
  - **timestamps**: all filesystems worth their salt store metadata on when a file was created or modified.
- **integrity assurances**: a good filesystem offers at least some guarantees on the metadata's integrity, with some also offering assurances on the actual file data's consistency as well.

9.5.1 Files are just inodes listing blocks.

In order to achieve their goals, filesystems employ quite a bit of abstracting high-level human concepts (say, a 10MB JPEG picture of a disk) to very low-level storage concepts. (like **blocks** number 200 to 300 on an actual disk)

One of the most notable "middle-men" in this whole chain of abstraction is the humble **inode** (or "Index Node"), which is the glue between the user-graspable concept of a file (aka its name), and the actual data on the disk. Have a look at the inode for the ext2 filesystem to better visualise it:

Figure 9.3: `ext2` inode structure.



Understanding the structure of the inode really is half the battle, let's see how this looks in practice:

```
# First off, let's create a testdir and some files:
$ mkdir testdir; cd testdir
$ echo "file1" > file1.txt
$ echo "file2" > file2.txt
$ mkdir subdir

# Passing "-i" to "ls" will have it list the ID of the inode of each entry:
$ ls -i
2375661 file1.txt  2375662 file2.txt  2375663 subdir

# We can see even more detailed info about a file using "stat":
$ stat file1.txt
  File: file1.txt
  Size: 6                 Blocks: 8          IO Block: 4096   regular file
Device: fd00h/64768d        Inode: 2375661     Links: 2
Access: (0664/-rw-rw-r--)  Uid: ( 1000/   <you>)   Gid: ( 1000/   <group you>)
Access: 2023-04-16 20:09:00.359492593 +0000

# Let's try to move the second file into the subdirectory:
$ mv file2.txt subdir
$ ls -i
2375661 file1.txt  2375663 subdir
# We can see that file2.txt has kept its inode. No actual physical data got
# moved, just the "file2.txt" entry from this directory got moved to "subdir".
# The data would have been moved only if "subdir" were on another partition.
$ ls -i subdir/
2375662 file2.txt

# We can demonstrate that file names are just directory entries with hard links:
$ ln file1.txt hardlink
$ ls -i
# Note inodes are the same, so the actual data both "filenames" point to is too.
2375661 file1.txt  2375661 hardlink

# Symlinks, however, are special, they are separate inodes completely:
$ ln -s file1.txt symlink
$ ls -i
2375661 file1.txt  2375661 hardlink 2375664 symlink

# "stat"-ing will show that it has a special "symbolink link" inode type.
# Whenever you try to do something on a symlink, the OS will pull a switcharoo
# on your program and actually pass it the inode of the symlink's target, which
# is why most commands need you to explicitly tell them *not* to follow links.
$ stat symlink
  File: symlink -> file1.txt
  Size: 9                 Blocks: 0          IO Block: 4096   symbolic link
Device: fd00h/64768d        Inode: 2375664     Links: 1
Access: (0777/lrwxrwxrwx)  Uid: ( 1000/   <you>)   Gid: ( 1000/   <group you)
```

Listing 42: `inodes` showcase.

9.5.2 Popular filesystems.

As mentioned, filesystems are OS-level concepts, which means that different types of OSes implement different filesystems, each with their own design purposes.

Filesystem	OS	Released	Max Partition	Max File	Design Goal
FAT	MS-DOS	1977	256MB	4KB	Floppy disks.
FAT32	Everything	1998	2TB	4GB	Easy and portable.
ISO	Everything	1986	8TB	4.2GB	Industry standard for CDs.
NTFS	Windows NT	1993	8PB	8PB	Windows default FS.
ReFS	Windows	2012	2**80B	16EiB	Modern Copy-on-write Windows FS.
exFAT	Windows	2006	128PB	128PB	Big files, flash drives.
HFS+	macOS	1998	8EB	8EB	Journaled Apple FS.
APFS	macOS	2017	8EiB	8EiB	SSD-optimized Copy-on-write FS
ext2	Linux	1993	2TiB	2TiB	Linux default, adds journaling.
ext4	Linux	2006	1EiB	16TiB	Linux default iteration on ext2.
btrfs	Linux	2009	16EiB	16EiB	Modern Copy-on-write Linux FS.
ZFS	Solaris	2006	256Trn. yobibytes	16EiB	The God FS

Consider taking the time of playing around with one of the more modern Copy-on-write-endowed filesystems like ZFS, Btrfs, or ReFS, the amount of features you can get directly from the filesystem layer is unreal!

9.6 Applied storage concepts.

In this section, we'll be putting all the above wall of theory into practice by creating a "disk in a file", setting it up, mounting it, and saving files into it. (file-ception!)

```
# First off, we need to pre-allocate an empty file, we can use "truncate":
$ truncate -s 5G ./testdisk.img

# Then, we need to partition the (currently empty) disk file. "fdisk" is the
# default tool guranteed to be on EVERY system nowadays, but it's a bit
# annoying to use. "cfdisk" is a more user-friendly alternative:
$ cfdisk ./testdisk.img
<select DOS for now, and create 3 Primary partitions, sized 2G-2G-1G>
# NOTE: REMEMBER TO EXPLICITLY SELECT "WRITE" IN CFDISK.

# Now we need a way to have the OS try to read the MBR on the file
# and allow us access to that new partition. For this, we can set up
# a "loopback device", i.e. a fake-disk-like block device which is actually
# writing data to our file. The loopback device will be one of
# /dev/loopXYZ, but we can ask for a specific number ourselves too.
# The "-P" argument has "losetup" ask the kernel to scan it for partitions.
$ sudo losetup -P loop23 ./testdisk.img
$ file /dev/loop23
/dev/loop23: block special (7/23)
$ lsblk
NAME                      MAJ:MIN RM   SIZE RO TYPE MOUNTPOINTS
loop23                      7:23   0     5G  0 loop
 -loop23p1                259:1    0     2G  0 part
 -loop23p2                259:1    0     2G  0 part
 -loop23p3                259:1    0     1G  0 part

# Bingo! There's our partitions! Now we just need format them to a filesystem.
$ sudo mkfs -t ext4 /dev/loop23p1
<info on how it went>

# Now finally, let's mount our shiny new filesystem!
# /mnt is a pre-existing directory on most Linux dirstros which is intentionally
# empty so users can mount stuff in them, but it can be any directory.
$ sudo mount /dev/loop23p1 /mnt
$ cd /mnt
$ sudo touch ./mine-now
# NOTE: files are NOT necessarily saved to disk (aka "commited") immeditely.
$ sudo sync  # this will forcefully "flush" all FSes to "commit" all files to disk.
```

Listing 43: Simulating and formatting our own storage.

```
# But what if we want to have this FS automatically mounted every time we restart?
# The "/etc/fstab" file (short for "filesystem table") is the configuration
# file read by the Linux kernel during its boot process, and defines which
# partitions have which filesystems and should be mounted where.
$ cat /etc/fstab
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/ubuntu-vg/ubuntu-lv during curtin installation
/dev/disk/by-id/dm-uuid-LVM-KtGR91lcnBTcOH8Y50ZE0... / ext4 defaults 0 1
# /boot was on /dev/sda2 during curtin installation
/dev/disk/by-uuid/f10f9532-329d-4362-ae25-b9179f8630e3 /boot ext4 defaults 0 1
/swap.img       none        swap        sw        0       0

# We can see our "/" and "/boot" partitions here, but they're identified via
# a "/dev/disk/by-{,uu}id" scheme for reliability (/dev/sda won't do).
# Let\s check out the ID of the ext3 filesystem of the loopback partition:
$ sudo blkid /dev/loop20p1
/dev/loop20p1: UUID="e466f9a9-df04-46cc-9175-caa641405e42"
$ file /dev/disk/by-uuid/e466f9a9-df04-46cc-9175-caa641405e42
/dev/disk/by-uuid/e466f9a9-df04-46cc-9175-caa641405e42: symbolic link to ../../loop23p1

# Now that we know the device, we can edit fstab and add this line at the end:
/dev/disk/by-uuid/e466f9a9-df04-46cc-9175-caa641405e42 /mnt ext4 defaults 0 1

# NOTE: the partition (/dev/loop23p1) still needs to be visible during
# startup for the kernel to mount it! This means that:
# 1) we must mount it AFTER / is mounted, since / contains the actual loopback file.
# 2) we must run the "losetup" automatically on boot (e.g. adding it to "/etc/initrc")

# Once we're done, we can (and should!) safely unmount the filesystem and
# deactivate the loopback device once we've had our fun with it.
$ sudo umount /dev/loop23p1
umount: /mnt: target is busy.
# Oh right, it won't let you unmount with a process (our shell) in it.
$ cd /  # move away to any other directory you'd like.
$ sudo umount /dev/loop23p1
$ sudo losetup -D loop23

# Repeat for /dev/loop23p{2,3} to practice.
# Remember that running "mount" in an already-mounted directory will "mask"
# the original mount, so do not mount p2 and p3 into the same "/mnt" drectory.
```

Listing 44: `/etc/fstab` auto-mounting.

9.7 Logical Volume Manager. (LVM)

Now that we've worked so hard to create and manage 3 individual partitions on the loopback device, it's time to immediately regret everything and wish we could just have one big partitions again so we can store bigger files...

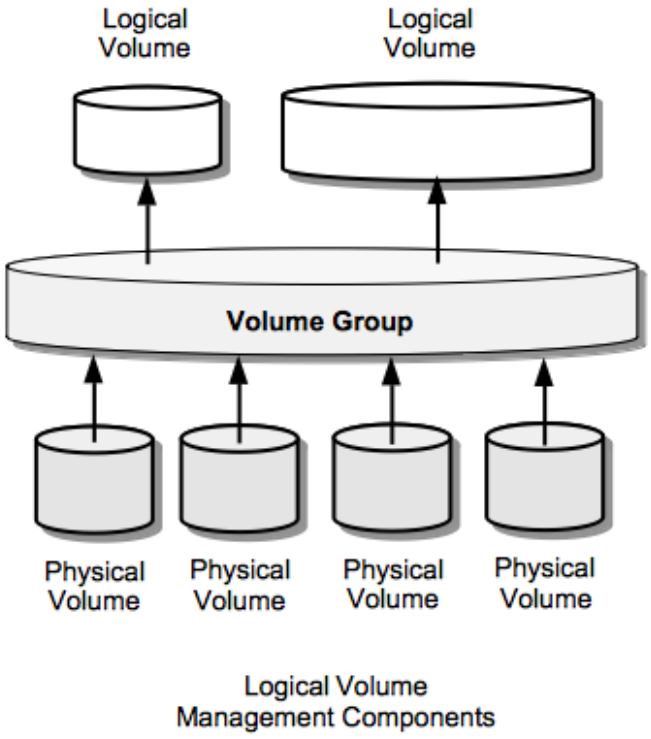
Apart from our indecisiveness, there are several other reasons why it would be nice for us to have a way to "group" all our disks in one giant "storage pool":

- the ability to have filesystems larger than the maximum disk/partition size: a technique often referred to as `spanning`.
- the ability to "back up" whole filesystems by maintaining multiple copies on multiple spanned disk in case one of the disks catches fire.
- better allocating small pieces of leftover space at the end of existing disks.

In Linux, the solution is the **L**ogial **V**olume **M**anager (or **LVM**), a storage management system which is pre-built-into Linux and allows to take individual paritions or whole disks and layer onto them the following abstractions:

- **Physical Volumes**: existing disks/partitions which we want to give LVM full control over. Note that they are only called "Physical Volumes" by LVM, and in reality can be any disk, partition, loopbackfile , or similar device. We should **never try to mount or use the disks directly after adding** them to LVM, as we may ruin the *meta-data* LVM stores on the disk for it to work.
- **Volume Groups**: an aggregation of one or more Physical Volumes. The Volume Group concept is what "aggregates" multiple Physical Volumes into one seemingly-contiguous data range out of which we can then allocate:
- **Logical Volumes**: a pre-allocated range from one or more Volume Groups which we can now use as a single volume. We can resize and move the Logical Volume between Volume Groups any time, as well as have it "span" multiple Physical Volumes if needed. (handled by the Volume Group)

Figure 9.4: Logical Volume Manager (LVM) concepts.



```
# Let's reuse the disk file from earlier:
$ file ./testdisk.img
./testdisk.img: DOS/MBR boot sector ...
# Ensure the loopback is set up:
$ sudo losetup -P loop23 ./testdisk.img
$ ls /dev/loop23p*
/dev/loop23p1  /dev/loop23p2  /dev/loop23p3

# Now let's add these paritions as LVM "Physical Volumes"
# (which don NOT necessarily need to be actually physical lol)
$ pvcreate /dev/loop23p*
$ pvdisplay | grep loop
  PV Name               /dev/loop23p1
  PV Name               /dev/loop23p2
  PV Name               /dev/loop23p3

# Nice, they've now been "adopted" by the LVM system, what now?
# Well, despite there being 3 separate partitions, we can ask LVM
# to join all 3 "physical volumes" into a single "volume group":
$ vgcreate loop23vg /dev/loop23p*
$ vgdisplay loop23vg
  --- Volume group ---
  VG Name               loop23vg
  Format                lvm2
  ...
  VG Size               <4.99 GiB
  PE Size               4.00 MiB
  Total PE              1277
  Free  PE / Size       1277 / <4.99 GiB

# Sweet, we now have 5 *whole* GBs to play with now, guess we should create
# a 5GB volume to wrap up this tutorial then, eh?
$ lvcreate -n look-mom-5-whole-gbs -L 5G loop23vg
Volume group "loop23vg" has insufficient free space (1277 extents): 1280 required.

# Ah dammit, I forgot LVM isn't free, it needs to store some "metadata"  on
# the "physical volumes" it manages, so the more volume groups and logical
# volumes you define, the less actual usable disk size you get, shame...
$ lvcreate -n sorry-mom-3-extents-short -l 100%FREE loop23vg
  Rounding up size to full physical extent 1.25 GiB
  Logical volume "sorry-mom-3-extents-short" created.
$ lvdisplay
...
  --- Logical volume ---
  LV Path                /dev/loop23vg/sorry-mom-3-extents-short
  LV Name                sorry-mom-3-extents-short
  VG Name                loop23vg
  LV UUID                3zC0Tk-XDQv-Irrh-REYT-MZY0-9jSs-Ldiuzp
  ...
  LV Size                <4.99 GiB
  Current LE             1277

# Now we can freely "mkfs", "mount", and enjoy our *almost* 5GB volume.
$ file /dev/loop23vg/sorry-mom-3-extents-short
/dev/loop23vg/sorry-mom-3-extents-short: symbolic link to ../dm-1
```


Listing 45: LVM showcase.

Chapter 10

The Linux boot process.

Finally, after all this long and arduous journey, we can finally put everything into perspective. This chapter will walk you through your average computer's boot process from the initial power button press to the second you can interact with your favorite spreadsheet.

10.1 Firmware.

The initial part of the powering up process is handled by a small piece of software called the **firmware**. It is usually harware-specific and provided by your motherboard manufacturer for these startup tasks:

- **POST**: the **P**ower **O**n **S**elf **T**est is a quick set of checks the firmware runs to ensure that RAM and attached bus devices are working properly.
- **hardware menu**: the firmware provides a quick menu for low-level system settings like CPU overclocking and *Secure Boot*.
- **execute bootloader**: after all POST checks have passed, the firmware will then iterate through all the configured bootable disks, look for a partitioning scheme, and point the CPU to start executing the boot code from the disk.

10.1.1 BIOS vs EFI.

It's worth noting the relatively recent evolution of "firmware types":

- **BIOS**: the so-called "**B**asic **I/O S**ystem" is the classic form that firmware has always taken. A classic BIOS is characterized by:
  - only being able to read Master Boot Record (9.4) partitioned disks and boot OSes installed on them.
  - very manufacturer-proprietary with little crossover or portability, leading to the need for many models to be supported in Linux and other OSes, as the kernel interacts with the firmware too.
  - does not support *Secure Boot*
- **(U)EFI**: the **U**nified **E**xtensible **F**irmware **I**nterface is the modern BIOS, bringing cross-industry consensus on a number of high-level features:
  - *Secure Boot*: a scheme for ensuring only code created by or signed by established OS manufacturers like Microsoft is allowed to run on your system.
  - ability to boot both MBR and GUID (9.4.1) partitioning schemes. MBR-based systems (often called "legacy boot") might not offer some GUID features like *Secure Boot*.
  - easier for manufacturers and programmers to write and share, since UEFI is based on a specification which is managed by the so-called "UEFI Forum".

10.2 Bootloader.

The **bootloader** is the next link in the chain, with its main purpose being to find and load the kernel and its needed drivers into RAM, and execute it.

The most commonly-used bootloader in the Linux space is GRUB, made by the GNU project to allow for booting any OS adhering to the so-called Multiboot spec.

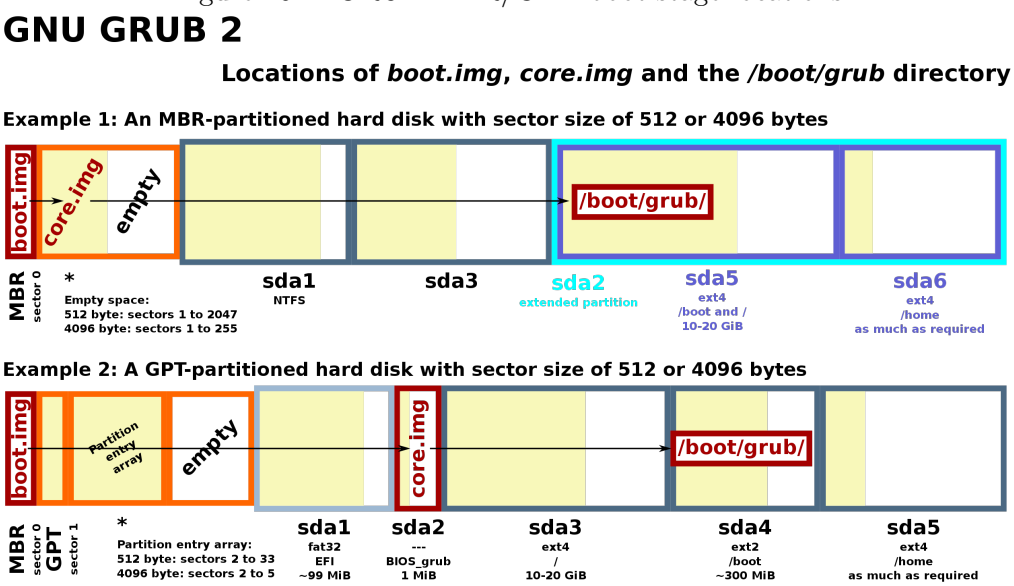
GRUB achieves its mission in a couple of separate stages:

- **Stage 1**: the code for GRUB's first stage (`boot.img`) actually lives within the MBR/GPT partitioning headers themselves. (that's how the firmware, which can only read MBR/GPT, knows where to find the bootloader: it's embedded in the MBR/GPT!) all stage 1 does is load GRUB's Stage 2 into RAM and execute it.
- **Stage 2**: stage 2's code (`core.img`) is capable of mounting filesystems in order to find the rest of GRUB in the `/boot` folder. Once Stage 2 finds and mounts `/boot`, it looks at `/boot/grub/grub.cfg`, which contains all the info on which kernel and Initial RAM disk files (2.4.1) to load and how.

```
# As mentioned, most of GRUB's files live in "/boot/grub".
$ cd /boot/grub

# GRUB contains a lot of modules which can be used during Stage 2 as necessary,
# ranging from hardware bus to Filesystem drivers and other tools:
$ ls /boot/grub/i386-pc
ext2.mod exfat.mod btrfs.mod   # Various filesystem modules.
elf.mod                        # Running ELF executables.
ls.mod                         # Even a module containing our boy "ls".
...

# The GRUB config file with the actual menu options and kernel infos is:
$ cat /boot/grub/grub.cfg
...
echo        'Loading Linux 5.15.0-53-generic ...'
# Here's the Linux kernel being loaded:
linux        /vmlinuz-5.15.0-53-generic root=/dev/mapper/lvm ro recovery ...
echo        'Loading initial ramdisk ...'
# And the Linux kernel's Initial RAM Disk (initrd):
initrd        /initrd.img-5.15.0-53-generic

# The above config is actually auto-generated based on some rules in:
$ ls /etc/grub.d
00_header  05_debian_theme  10_linux  ...

# If you ever want to change your GRUB theme or name your Linux menu entry
# "Pengu" or whatever, you can just edit these files and run the following:
$ grub-mkconfig -o /tmp/grub.cfg

# Once it succeeded and you're sure you know what you're doing, you can:
$ mv /tmp/grub.cfg /boot/grub/grub.cfg
```

Listing 46: GRUB files overview.

Figure 10.1: GRUB MBR/GPT boot stage locations.



10.3 Kernel.

Once the bootloader loads the kernel and all its drivers into RAM, the system has technically booted. That doesn't mean it's any use to anybody just yet, however. Once it runs some basic checks for the root filesystem's consistency (which it initially mounts read-only), the kernel will run one single program: `/sbin/init`, aka the **init system**.

10.4 Init.

The **init system** is the first (PID 1) and only process ever directly started by the Linux kernel without a prior trigger. It is supposed to be an executable which later sets up and configures all the remaining system services. This includes:

- start services which set up any remaining devices like `NetworkManager`.
- start core services which may be used by other services. (e.g. an inter-service messaging service like `dbus`)
- start the core services for the Desktop Environment. (2.4.3)
- finally, start any User services like Steam or whatever.
- the hard part: **do everything in the correct order**!

10.4.1 `sysv`, `upstart`, and legacy init systems.

For the longest time, most POSIX-based systems were using a system derived from a UNIX release called "**System V**" (Roman numeral "V"), where each service had to have a dedicated script with only a handful of actions like `start/status/stop`, and said script was solely responsible for ensuring all of the dependencies of the service are started.

All that `sysvinit` did was call the right "start/stop/bla" action whener needed. This lead to quite a lot of trouble maintaining all of the scripts in between themselves, easpecially considering the authors of the init scripts were either authors or re-packagers of the software.

Canonical (the company behind Ubuntu) also tried their hand at an init system called `upstart`, which did bring some benefits to how/when the init scripts were triggered, but not to how they were written and how unmaintainable they are.

10.4.2 `systemd`

Back in 2010 at Red Hat Linux HQ, a revolution was brewing: a band of ambitious engineers set off to create a new init system with the goal of it not being bad.

Out came systemd (short for System Daemon, aka "Serviciul Serviciilor"), which brought about a considerable number of improvements over the init systems of days past:

- considerably easier service definition using `.unit` files instead of scripts.
- direct monitoring of processes: `systemd` instantly knows when a service has crashed and can restart it or perform other appropriate actions.
- ability to use `.socket`-type services: systemd can monitor an internet port and only run the service while it's needed.

```
# First off, feel free to look through the abhorrent sysV init scripts in
# /etc/init.d, it will really help put systemd's features into perspective:
$ cat /etc/init.d/cron

# Back to the future: you can list running services with "systemctl status"
$ systemctl status

# Note how everything is a clear, logical, and centralised tree of services.
$ systemctl status cron
x cron.service - Regular background program processing daemon
     Loaded: loaded (/lib/systemd/system/cron.service; enabled; ...)
     Active: active (running) since Tue 2023-04-11 13:25:27 UTC; 5 days ago
...

# Hey, there's that "cron" service we just saw. Let's check out its unit file:
$ cat /lib/systemd/system/cron.service
[Unit]
Description=Regular background program processing daemon
After=remote-fs.target nss-user-lookup.target

[Service]
EnvironmentFile=-/etc/default/cron
ExecStart=/usr/sbin/cron -f -P $EXTRA_OPTS
IgnoreSIGPIPE=false
KillMode=process
Restart=on-failure

[Install]
WantedBy=multi-user.target

# You can mess with it using "systemctl":
$ systemctl start/stop/restart cron.service

# The above is functionally equivalent with "/etc/init.d/cron". It's that easy.
# "ExecStart" is the only line we really care about: it is the command that
# systemd will run whenever it starts the service.
# The "Restart=on-failure" is literally all it takes to have systemd
# monitor and automatically restart the service if it ever crashes.
# Life is suddenly good again, thanks systemd!
```

Listing 47: `systemd` and service initialization.

Chapter 11

Exercises for the Reader.

First off, an honest thank you for making it this, we hope this document was as informative as it was lighhearted, and it has sparked the joy of Open Source within you so that you may continue your Linux journey on your own.

This chapter will list some exercises which will greatly help in deepening some of the concepts covered, and it is recommended that you at least give them a shot. Who knows, maybe your fork of the `crash` shell will become the next Bash!

Remember: trying and failing miserably is the very first step towards succeeding. Questions are always welcome.

11.1 Scriptception.

Write a shell script which:

- only works as a non-root user, exits with a "must not be root" error.
- gets a directory or tar archive (*.tar.gz, *.tgz) as the main argument.
- traverses all of it and its subdirectories/archives to find all files containing a certain RegEx in them and print their file names. (**-n** argument must be given for filename printing)
- the RegEx pattern argument is provided as the key-value flag `-re "<the RegEx>"`.
- if tar archives are found, search for files in them as well do not leave any evidence that you unarchived anything anywhere on the filesystem. (i.e. remember to perform proper cleanup)
- print the number of found files. (**-c** argument must be given)
- check all arguments, both -c and -n may or may not have been given.

```
# "-n" prints the names (pathname) of the files containing the word "word".
./script -re "word" -n dir_or_tarfile
# "-c" prints the number (count) of arguments.
./script -c dir_or_tarfile -re "[Mm]ore\sComplicated[.!?]"
# Both prints filenames and counts them and displays the number at the end:
./script -n -c -re "[Mm]ore\sComplicated[.!?]" dir_or_tarfile
```

Listing 48: Scriptception arguments example.

NOTES:

- you'll have to research the `tar` command on your own, and set up your own testing "subjects" (aka directories, files with/without the pattern, tar archives with directories and files with and without the pattern).
- you'll have to research how to declare and use functions in Bash on your own check all your parameters at the beginning of the script, and save the "presence" of -n and -c in some variables

for later use. The parameters may come in ANY order so do be aware of that (hint: use a while-shift-case combo)

- you can use **$0 param1 param2 ...** to make your script "recursive".

11.2 Sleep service.

Your task is to write a `systemd .unit` file (47) which:

- simply executes `sleep N`, where N is an environment variable declared in a separate file. Do not just hardcode it to `sleep 100`!
- always restarts the sleep once it's done.
- is triggered after `network.target` is reached,
- add an `ExecStartPre` clause which logs whenever the service is restarted
- double-check using `systemctl status` and `journalctl` that the service is acting as expected.
- BONUS: pick some very simple pre-existing service (or just copy a trivial Flask app) and write a `.socket` systemd file for the service. Note that you'll still need to define a `.service` file as well as the `.socket`, since systemd just starts the regular service definition whenever a connection hits the socket definition.

11.3 Filesystems of legend.

We've went trough the concepts of a filesytsem (9.4.2) and created and formatted our own loopback device to ext4 (43), but ext4 and the filesystems everyone uses are really the boring ones.

I present unto you, ZFS, the literal Zetta-byte File System. (I don't even know what Zetta means!) It's a filesystem with SO MANY legendary features like file snapshots, automatic remote mirroring, pooling and spanning, and soooooo much more... ...but it only works on Solaris thanks to Oracle... :(

There is, however, an up-and-coming competitor in the Linux space as well: the **B-Tr**ee **F**ile **S**ystem (or BTRFS, read "butter eff ess"), which provides quite a few of the notable features of ZFS. Your task will be simple:

- set up a loopback device with GPT and several partitions as we've already shown (43) OR use an actual physical parition if you'd like.
- format it to BTRFS:
  - make the filesystem span at least 3 partitions.
  - set up **subvolumes** to actually mount later.
- mount the FS and save some files to it.
- make a "cron" job to automatically snapshot the FS.
- have the FS be auto-mounted at system startup.

ULTRA BONUS: set up a virtual machine with OpenIndiana (a descendant of OpenSolaris, you'll feel right at home if you like Linux) and set up some ZFS storage pools and play around with ZFS a bit.

11.4 Improving Upon a Completely Original Shell.

Below you'll find some code (49) for a very simple Bash ripoff written in Python. Its name is the Complete Ripoff Shell (or `crash` for short), but it's missing some completely original usability features which are really holding it back and we should probably do something about them:

- please for the love of Computer Science just make "cd" work please!!!
- comments would be nice too (i.e. ignoring any character after a)
  - passing envirommment variables to child processes as shown in 18. (aka how "export" works)

- Make variable declaration consistent with Bash (it curently isn't, since things like `VAR="multi word string"` don't work properly)
- Ctrl+C stops the whole shell instead of just the program it's running. You'll need to make `crash` properly handle the `SIGINT` OS signal. (16)
- there's no I/O redirecting (12), we should at least have our shell support `>` and `<` for file I/O.
- "*" globs as shown in 9. Not too fancy though, expanding one single "*" is more than useful enough!
- EPIC BONUS ROUND: implement command history (at most 10 commands with Ctrl+R searching option)

Hints:

- Python (and any other structured programming language for that matter) works a lot better if you define multiple small functions for pieces of logic you want to reuse and compose them up to one big set of logic. It's also easier to follow along with and debug this way!
- the `subprocess` module, which is already used to execute child programs in `crash`, is an extremely versatile module, make sure to skim trough the subprocess documentation and be amazed how easy it is to use once you know how process I/O works. (3.9)
- there's a builtin Python module named "shlex".
- for command history and general navigation: don't worry about "redrawing" text lines which were already printed out (like Bash does when you Ctrl+R), as that's relatively hard to implement for relatively little gain.

```
#!/usr/bin/env python3

import os
import subprocess
import sys


ENVIRONMENT = {
    "PROMPT": "Welcome to crash! >>> ",
    "UID": str(os.getuid()),
    "GID": str(os.getgid()),
}

def prompt():
    print(ENVIRONMENT["PROMPT"], end="", flush=True)

def substitute_vars(command):
    for var, val in ENVIRONMENT.items():
        command = command.replace(f"${var}", val)
    return command

def handle_command(command):
    # Check if variable definition and save it:
    if '=' in command:
        var, val = command.split("=", maxsplit=1)
        ENVIRONMENT[var.strip()] = val
        return

    # Run the command:
    subprocess.check_call(command.split(" "))

if __name__ == "__main__":
    prompt()
    for line in sys.stdin:  # Read.
        try:
            substituted = substitute_vars(line.strip()) # Eval.
            handle_command(substituted)                 # Print.
        except Exception as ex:
            print(f"Error occurred: {ex}")
        prompt()    # Loop.
```

Listing 49: The `crash` shell in all its glory.