Linux Workshop

Mitchell Dzurick

11/09/2022

What are we doing Today

Table of Contents

Presentation Source Code

This presentation and source code can be found at
`https://github.com/mitchdz/IEEE_Linux_Workshop`

About Me

- UA Alum - Masters in Computer Engineering '21
- Really likes Linux
- Currently Systems Validation Engineer for Intel QAT Compression service
- Used to build Custom Linux images for Intel Trusted Edge Platform

← Has a large space heater

Motivation

- Linux isn't taught well in school
  - often employers expect you can use it
- I love this stuff
- You might love this stuff too

Foreword

Who is this workshop for?

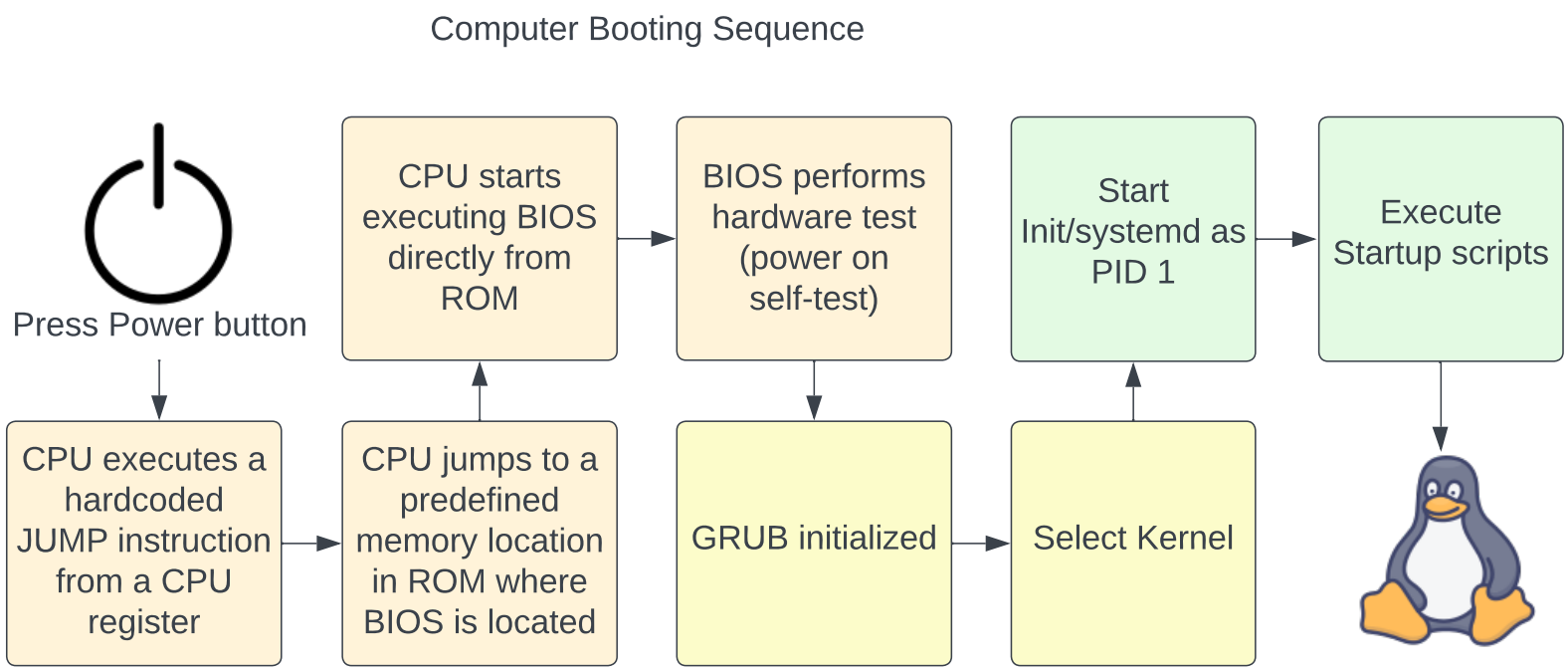
- New and intermediate Linux users alike

What can you expect to learn today?

- Enough to understand what Linux is
- Basic Linux commandline operation
- Tips on system usage

Table of Contents

Boot Sequence



Quick note on SysVInit/Systemd

SysVInit (init.d) and systemd are competing system and service managers. They are the most popular in Linux distros.
SysVInit commands look like so:

- sudo /etc/init.d/apache2 status
- sudo service apache2 status

systemd commands look like so:

- sudo systemctl status apache2.service

Arch/Debian/Ubuntu/CentOS/Fedora/Manjaro/RHEL are some examples that use systemd by default
Gentoo has SysV default

What is a system and service manager?

The system and service manager are two different jobs, but usually a single tool does both.

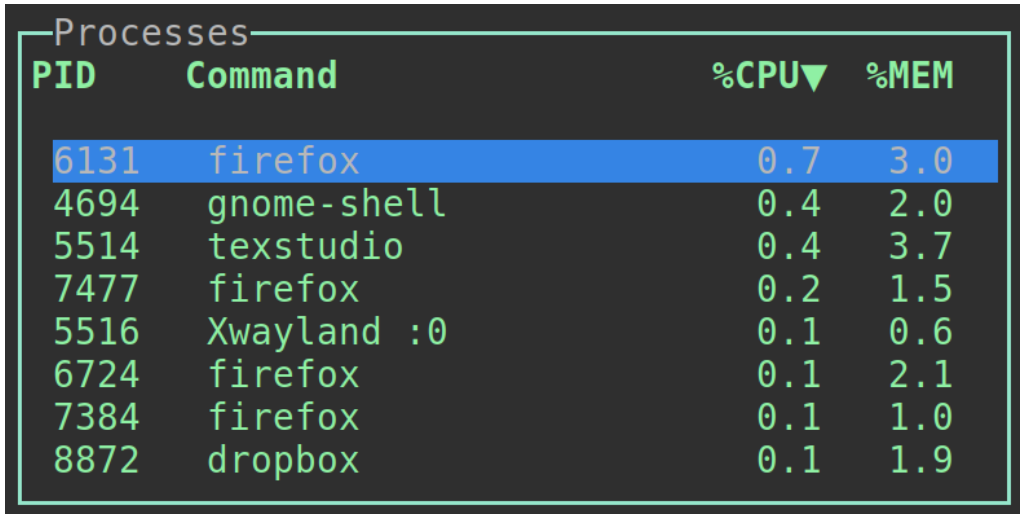
- System manager is an **init** system used to bootstrap user space
- Service manager is a tool to manage and interact with **daemons** and user processes

There are many alternatives to SysVInit and systemd, for example we have:
MSConfig, OpenRC, s6, Launchd, eudev, runit, bootchart

Linux Systems Processes

There are 3 types of Processes you will find in a Linux System

- User Processes
- Daemon Processes
- Kernel Processes



User Processes

- Initiated by a regular user
- Runs in user space
- often interact with it visually

Below is an example of running a command in a user process, the common update command on Debian distributions is one system administrators will know well.

```
sudo apt update -y && sudo apt upgrade -y
```

Daemon Processes

- Designed to be ran in the background
- No user interface

You can easily convert a user process into a daemon by simply appending & to the end of the command

```
sudo apt upgrade -y &
```

NOTE: the daemon is attached to the shell in this case, so if you close the shell, the process will stop. You can prepend the utility 'nohup' before the command to alleviate that

Kernel Process

AKA kproc

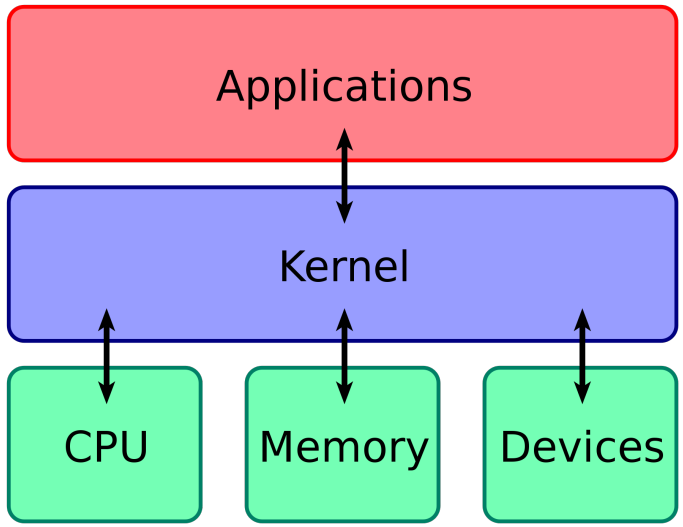
- Only run in Kernel Space
- Very similar to Daemon Processes, except they can access Kernel Data Structures
- Just need to know that they are very powerful and we won't be creating our own today

```
 ps --ppid 2 -p 2 -o uname,pid,ppid,cmd
USER            PID    PPID CMD
root              2       0 [kthreadd]
root              3       2 [rcu_gp]
root              4       2 [rcu_par_gp]
root              5       2 [netns]
```

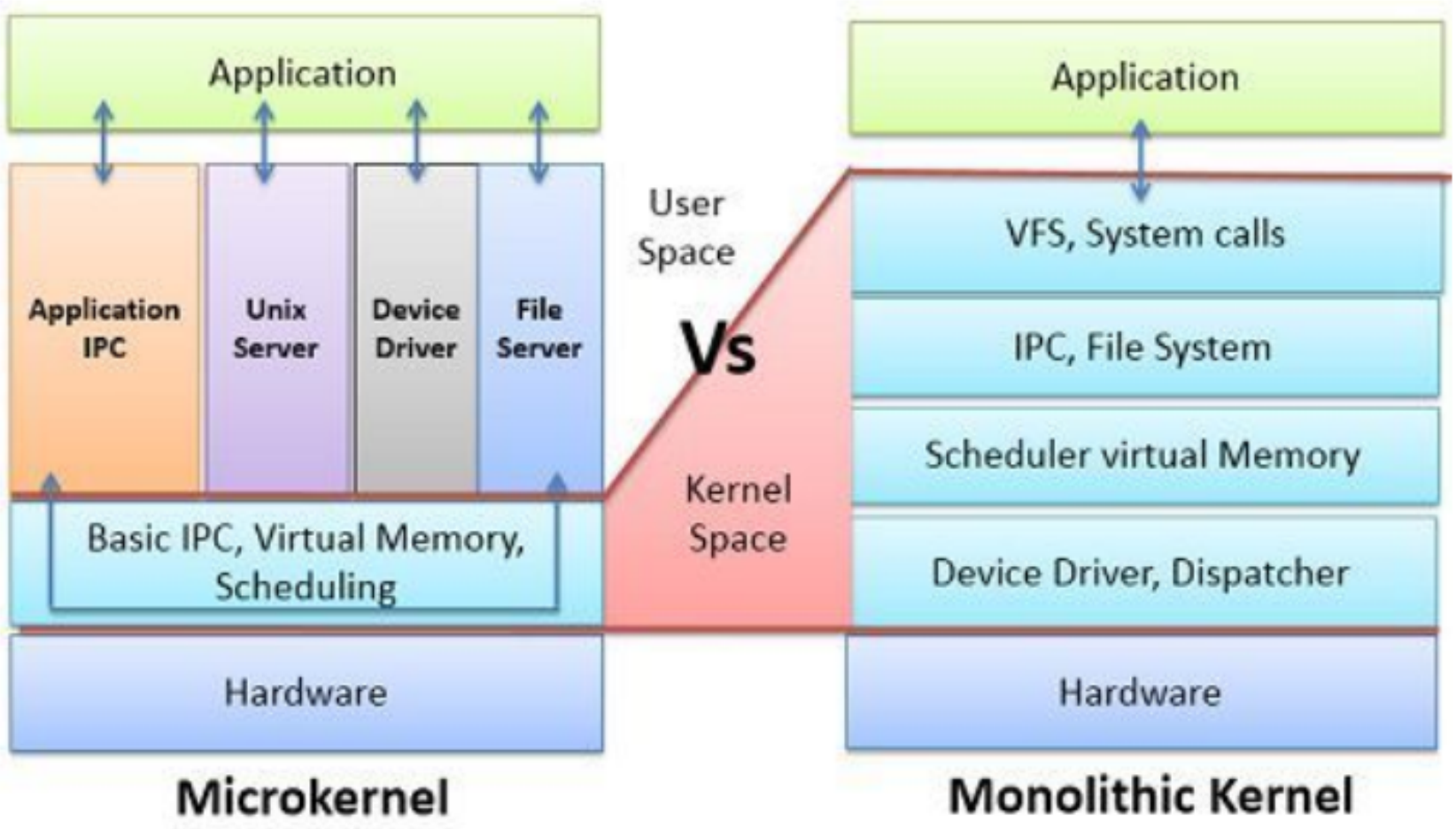
Table of Contents

What is Linux?

- Linux is a Kernel
- All operating systems have a kernel
- The kernel is a piece of software that tells your userspace applications how to talk to the hardware



Monolithic versus microkernel



What is a Linux Distribution?

A Linux distribution is a **collection of software packages** that make up a full-fledged Operating System. Here's just a few: Ubuntu, Fedora, Arch, Kubuntu, Scientific Linux, Parrot OS, NixOS, Hannah Montana Linux, Puppy Linux

What is a Linux Distribution? - Package Manager

Package management is a very important part of choosing a linux distro, and each package manger has it's own pros/cons

- **apt** - Debian based distros (ubuntu, kali linux, ect...) (deb based)
- **yum/dnf** - Fedora, CentOS, RHEL (rpm based)
- **pacman** - Arch
- **zypp** - OpenSUSE (rpm based)

What is a Linux Distribution? - Linux Distribution Timeline

https://upload.wikimedia.org/wikipedia/commons/1/1b/Linux_Distribution_Timeline.svg

Introduction to Filesystems

The filesystem (often referred to as fs) is a very important part of Linux (and all operating systems)
The fs is just a **method and data structure that the OS uses to control how data is stored and retrieved**. Common filesystems are ext2, ext3, ext4, btrfs, squashfs, FAT, NTFS

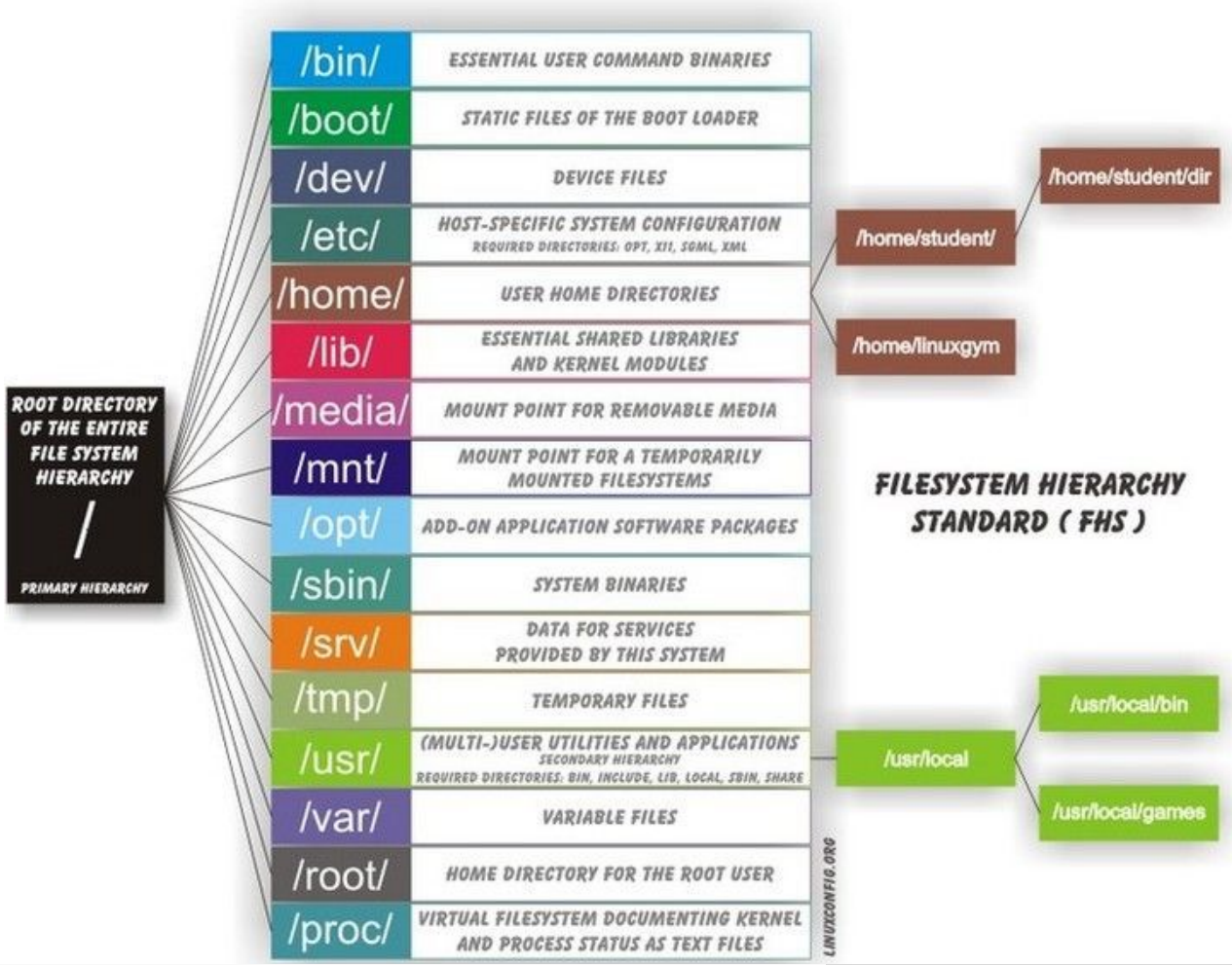
```
sudo mkfs.vfat /dev/sda1
mkfs.fat 4.2 (2021-01-31)
```

Everything is a file

Just like the title says, everything is a file.

- Text document? file
- driver? file
- flash drive? file
- network attached storage? file
- hardware accelerator (TPM,QAT)? file
- 2D printer attached via USB cable? file

Introduction to Filesystem Hierarchy Standard (FHS)



Linux Kernel vs. Userspace

The kernel is the core of the operating system and has full access to all memory and machine hardware, whereas userspace does not have that access

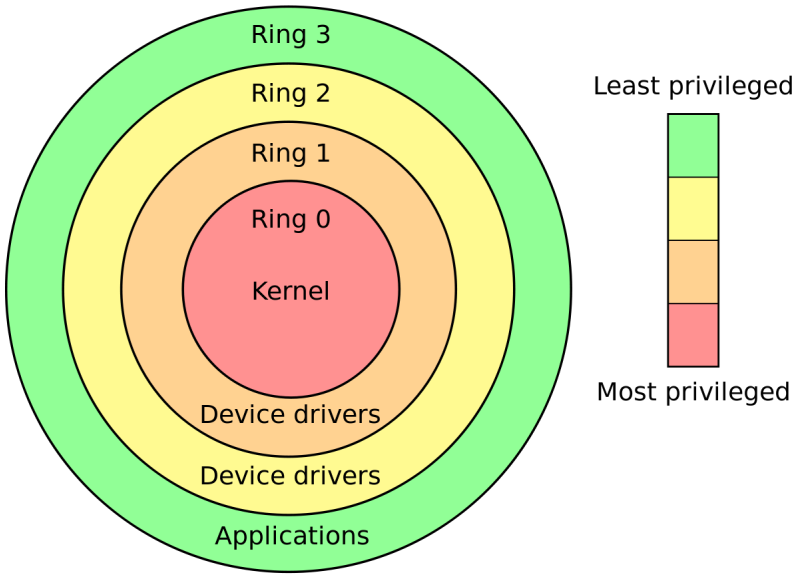


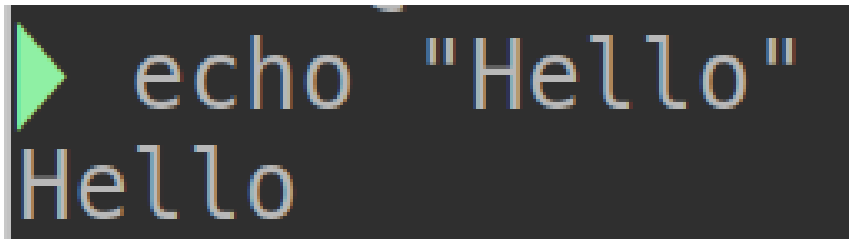
Table of Contents

Well, what is a shell?

The shell is an user space program that takes commands from the keyboard and gives them to your operating system to perform

Commandline

The commandline has a prompt which is what you type commands into



Bourne Shell

- The Bourne shell was the default shell in version 7 of unix, and was created in **1977**
- still present in most systems today usually at **/bin/sh**
- Created to supercede the Thompsone shell AKA **/bin/tsh** which was created in **1971**

```
$ echo $0
/bin/sh
```

Bourne-Again Shell

- Commonly located at **/bin/bash** and is the default shell for many Linux distributions
- The shell we will be looking at a lot today :)

BASH

THE BOURNE-AGAIN SHELL

Quick note about other shells

There is so many other shells out there, but we will be focusing on BASH today. Personally I use oh my zsh at work and home, but it's all mostly a preference thing.

Other shells

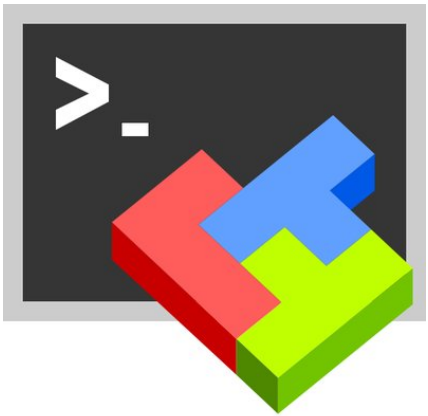
- zsh, fish are popular alternatives to bash
- Tcsh and Ksh are old favorites with strict POSIX compliance
- Ash, Dash lightweight so useful for embedded systems
- Powershell exists

Table of Contents

SSH Client

If on Mac/Linux just open a terminal and use that

If on Windows, use a SSH Client such as MobaXTerm/PuTTy



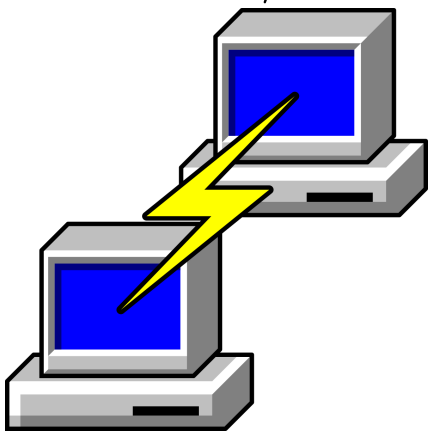


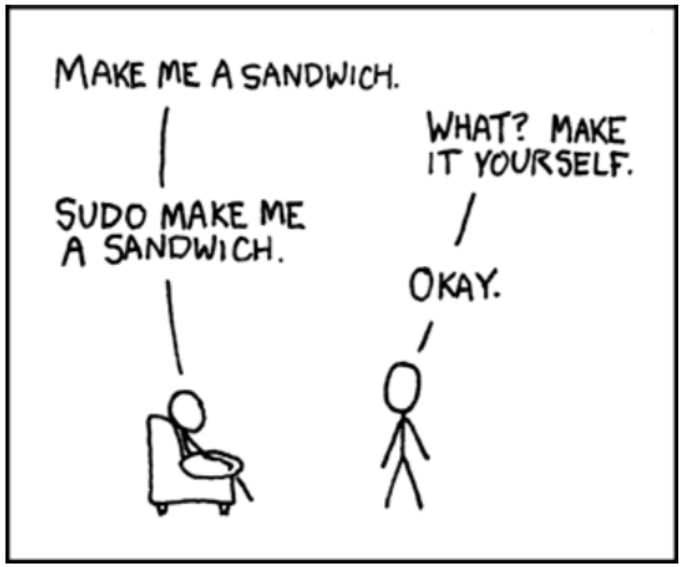
Table of Contents

sudo

sudo is a **program** that enables regular users to run programs with security privileges of **another user**

- A good example is ‘sudo apt update‘

Disclaimer: If a random web page tells you to use sudo and you don’t know what you’re doing, don’t do it.



sudo - quick note

When using sudo you will be **prompted for a password**. There will be a grace period where when you use sudo again you won't have to input password. This can be dangerous if you, say, update your system, and then run some sketchy shell script that you aren't reading the contents of. If said shell script uses sudo for something then it will just get free access.
This is why I always recommend running new shell scripts in a new shell (a container/vm is even better).

Basic Commands

- '**pwd**' print working directory
- '**ls -l**' list contents of current directory with listing format
- '**mkdir test**' make a directory named "test"
- '**cd test**' change directory to the dir "test"
- '**touch file.txt**' create a file named "file.txt"
- '**file file.txt**' inspect what the file is
- '**rm file.txt**' remove the file
- '**echo "Hello!"** $>$ **file.txt**' redirect STDOUT to the file "file.txt"

Shell Expansions

There are 8 different types of expansions: **brace expansion, tilde expansion, parameter expansion, arithmetic expansion, command substitution, word splitting, filename expansion**

```
 echo /{var,opt}/
/var/ /opt/
 mitch@lightning  ~/test
 echo /{var,opt}
/var /opt
 mitch@lightning  ~/test
 echo ~
/home/mitch
```

Shell Expansions - filename and word splitting

We can expand filenames (*) and do word splitting [@] to find how many files there are (#) in cwd

```
 words=(*)
 mitch@lightning  ~/tmp
 echo ${words[@]}
dir1 file1 file2 test.sh
 mitch@lightning  ~/tmp
 echo ${#words[@]}
4
```

Parameter Expansion

In our shell, we can set and use variables

```
var1=5
mitch@lightning ~/test
echo $var1
5
```

Parameter Expansion - sub strings

We can also print only specific parts of a string with the ":offset:length" operator

```
 string="/home/user/bin/"
 mitch@lightning  ~/tmp
 echo ${string:6:2}
us
 mitch@lightning  ~/tmp
```

Math is fun

We can even do math with variables!

```
 var1=5
mitch@lightning ~/test
 echo $(( (${var1}+3) / 2 ))
4
mitch@lightning ~/test
```

Redirections

Basic Redirections

- echo "hello" > file.txt # overwrites file and writes hello
- date >> file.txt # appends output of date command to file

File Descriptors STDIN (0), STDOUT (1), STDERR (2)

```
ls -l /dev/fd/
total 0
lrwx------ 1 mitch mitch 64 Nov  9 13:41 0 -> /dev/pts/0
lrwx------ 1 mitch mitch 64 Nov  9 13:41 1 -> /dev/pts/0
lrwx------ 1 mitch mitch 64 Nov  9 13:41 2 -> /dev/pts/0
```

Redirection - file descriptors

I can redirect only STDOUT to a file

```
ls ~ 1> file
```

I can redirect only STDERR to a file

```
ls undefined 2> file
✘ mitch@lightning ~/tmp
cat file
ls: cannot access 'undefined': No such file or directory
```

Redirection - file descriptors 2

What if I want BOTH STDOUT and STDERR?

```
 ls ./ undefined 1>file 2>&1
 ✘ mitch@lightning  ~/tmp
 cat file
ls: cannot access 'undefined':
./:
dir1
file
```

Redirection - file descriptors 3

What if I just want to see STDOUT and not STDERR?

```
ls ./ undefined 2>/dev/null
./:
dir1  file  file1  file2  test.sh
```

Redirection - Multiple Redirections

This is a fun example of something I do at work all the time

```
head -c 32 < /dev/random > 32B_random.bin
```

Read left to right, we take the first 32 Bytes of /dev/random and redirect that to a file

Pipes

Pipes are similar to redirection commands, except it works to connect the STDOUT of one comment to the STDIN of another command

```
printenv | grep SHELL
GNOME_SHELL_SESSION_MODE=ubuntu
SHELL=/bin/zsh
```

In this case, printenv writes to STDOUT, and that is redirected to STDIN of grep

Pipes - real life example

This is a string of commands that will kill all firefox instances

```
ps aux | grep firefox | grep -v grep | awk '{print $2}' | xargs kill
```

Exit Status

All commands have an exit status

```
mitch@lightning ~/tmp
(exit 255)
✘ mitch@lightning ~/tmp
echo $?
255
```

These are convenient ways for scripts to know how the last command went

&& and ||

&& (AND) will do the second command only if the first command is successful
|| (OR) will do the second command if the first command fails

```
 echo a && echo b && (exit 1) && echo c
a
b
 ✘ mitch@lightning  ~/tmp
 (exit 1) || echo a || echo b
a
```

Conditional Expressions

```
 cat test.sh
#!/bin/bash
if [ 0 -lt 1 ]
then
        echo "0 is lower than 1"
elif [ 1 -lt 2 ]
then
        echo "1 is lower than 2"
fi

 mitch@lightning  ~/tmp
 ./test.sh
0 is lower than 1
```

Conditional Expressions 2

You bet we got case statements too!

```
case "orange" in
orange|apple|pineapple) echo "yay";;
banana) echo "gross";;
esac
yay
```

Conditional Expressions 3

We can also check that a file exists with -a

```
echo -n "" > file \
[ -a file ] && echo "file already exists"
file already exists
```

Loops

We have just basic for loops

```
for i in {1..3}; do echo $i; done
1
2
3
```

Loops

We can mix file expansion to do a certain operation with multiple files

```
for file in *; do echo ${file}.bak; done
32B_random.bin.bak
dir1.bak
file.bak
file1.bak
file2.bak
test.sh.bak
```

Table of Contents

Shebang

A sheband (haSH BANG) is at the top of script files to be ran with a certain interpreter

```
#!/bin/bash
```

In the above example, **/bin/bash** is being used as the interpreter

Functions

Functions are a way for you to define a set of commands to reuse

```
hello() {
echo "Welcome ${1}, today is $(date)."
}
 mitch@lightning ~/tmp
hello $USER
Welcome mitch, today is Wed Nov  9 03:59:35 PM MST 2022.
```

Special Parameters

The first special parameter is positional arguments, this is how you can receive arguments from the commandline

```
  1 #!/bin/bash
  2 echo $1,$2,$3
  3
"test.sh" 3 lines --100%--
 ./test.sh Tomatoes Potatoes
Tomatoes,Potatoes,
```

In the above image I give the positional arguments Tomatoes ($1) and Potatoes ($2) and nothing to $3
What do you think $0 will print out?

Special Parameters - Expand

What if you want each positional argument printed as a word?

```
 test() {
function> echo $@
function> }
 mitch@lightning  ~/tmp
 test Tomatoes Potatoes
Tomatoes Potatoes
```

$@ got you covered

Special Parameters - IFS

IFS stands for Internal Field Separator

```
 test() {
function> IFS="~|-"
function> echo "$*"
function> }
 mitch@lightning  ~/tmp
 test Tomatoes Potatoes
Tomatoes~Potatoes
```

Special Parameters - num arguments, exit status

You can also get the number of arguments

```
 test(){
function> echo $#
function> }
 mitch@lightning  ~/tmp
 test 1 2 3
3
```

And you can also print the return code from the last ran command

```
 mitch@lightning  ~/tmp
 (exit 255)
✘ mitch@lightning  ~/tmp
 echo $?
255
```

Special Parameters - Recap

Recap of special parameters

- $1, $2, $3 are the value of positional arguments
- $0 expands the name of the bash script
- $@ turns all positional arguments into a single word
- echo "$*" expands positional arguments by IFS
- $# expands number of positional arguments
- $? Exit status from recent command

lstree

```
#!/bin/bash

function rlist() {
  local dir=${1:-.}
  local level=${2:-0}
  ls -1 -A $dir | while read -r entry # pipe output to loop
  do
    if [ -d "$dir/$entry" ] # file test
    then
      printf "%${level}s + %s\n" "" "$entry" # formatted print
      rlist "$dir/$entry" $((${level}+1)); # recursive function call
    else
      printf "%${level}s |- %s\n" "" "$entry"
    fi
  done
}

rlist $1; # function call
```

Thanks

Thanks Peter Andreas Möller, as I stole your lstree source code and some lesson structure.