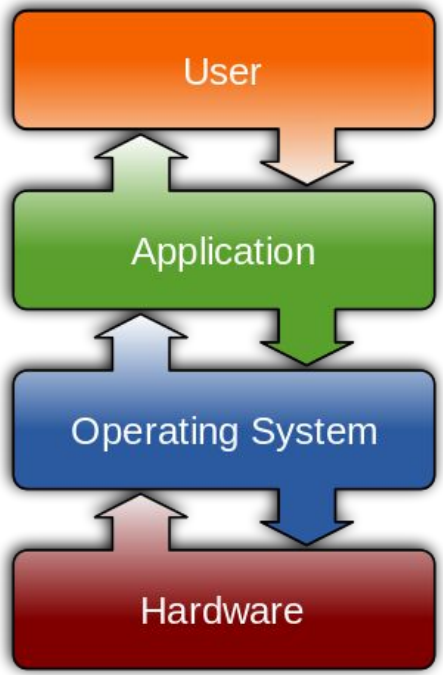
# DevOps basics: introduction to operating systems

# What an Operating System is ?





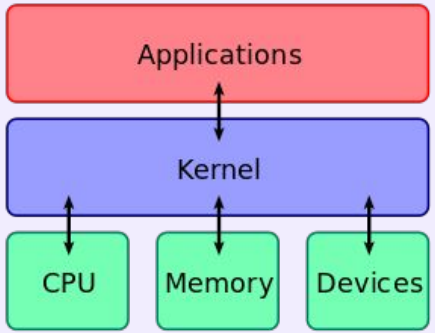
# What an Operating System is ?

An Operating System is the software which manages physical computing resources, interfaces between the hardware and the applications on a computer, and what exposes a creates a number of APIs for giving developers access to low-level applications / hardware. The OS allows application developers and hardware manufacturers to do their jobs and not worry about *“How does this spinning disk affect my browser”* and *“How will this networking card interact with my game engine”*.

# Anatomy of an OS

The OS is not *always* one thing or another. Some Operating systems are behemoths while others are minimal. Some are designed for teaching purposes while others are optimized for managing data centers.

The general diagram, from **you -> hardware**, looks like this:




- **User Interface:** What you interact with. Window Managers for instance.
- **Application Layer:** What developers use to make software run.
- **Kernel:** *The Core of the OS*. Makes communication between hardware and applications sane.
- **Hardware:** What does the actual computations. The thing your keyboard is plugged into.

# Anatomy of an OS (1)

The two middle parts (The Kernel and the Application Layer) of the diagram are often put together and called an Operating System. However, the scope of each layer is one of Computer Science's oldest and most contentious philosophical debates. *Microkernels* such as Mach and MINIX only implement a bare minimum interface to bridge the gap between software and hardware. In a Microkernel, software such as device drivers and file systems are separate from the kernel, and instead run in the Application Layer. On the other hand, *Monolithic Kernels* such as Linux include drivers, file systems, and other software as a part of the kernel.

# What is a Kernel?



The kernel of an operating system is something you will never see. It basically enables any other programs to execute. It handles events generated by hardware (called *interrupts*) and software (called *system calls*), and manages access to resources.

The hardware event handlers (*interrupt handlers*) will for instance get the number of the key you just pressed, and convert it to the corresponding character stored in a buffer so some program can retrieve it.

The *system calls* are initiated by user-level programs, for opening files, starting other programs, etc. Each system call handler will have to check whether the arguments passed are valid, then perform the internal operation to complete the request.

Most user programs do not directly issue system calls (except for asm programs, for instance), but instead use a *standard library* which does the ugly job of formatting arguments as required by the kernel and generating the system call. (For example, the C function *fopen()* eventually calls a kernel function that actually opens the file.)

The kernel usually defines a few *abstractions* like files, processes, sockets, directories, etc. which correspond to an internal state it remembers about last operations, so that a program may issue a session of operation more efficiently.

# What is a shell?

A shell is a special program that is usually integrated in an OS distribution, and which offers humans an interface with the computer. The way it appears to users may vary from system to system (command line, file explorer, etc), but the concept is always the same:

- Allow the user to select a program to be started, and optionally give it session-specific arguments.
- Allow trivial operation on the local storage like listing the content of directories, moving and copying files across the system.

In order to complete those actions, the shell may have to issue numerous system calls, like "open file 'x'; open file 'y' and create it if it doesn't exist; read content from X, write into Y, close both files, write 'done' to standard output".

The shell may also be used by programs that want to start other programs but do not want to do this themselves (e.g. completing file patterns like "*.mp3", retrieving the exact path of the program, etc.).

Modern shells can also have various extra features, such as the following:

- Auto-Completion: By pressing the TAB (or any preferred) key, the word the user is typing will be completed to a valid shell command, a file, directory, or something else. Pressing the auto-complete key multiple times cycles through other completion possibilities.
- Character Insertion: The user can move around in what he or she entered by using the arrow keys. When typing new characters in the middle of a sentence, characters will get "inserted".
- Shell History: By using the up and down arrow keys, the user can scroll through previous input.
- Scrolling: When there are more lines than the console is high, save the output in a buffer and allow the user to scroll up and down in the console.
- Scripting: Some shells have custom scripting languages. Examples of scripting languages are Bash or DOS batch.

# Types of Operating Systems

Most of us only interact with one or two OS's in a day: our phone OS and personal computer OS. There are many other types of OS's depending on a variety of needs. Scientific computing for instance has different requirements than a pace-maker or a GameBoy. Each of these areas has their own *types* of applications they run and as a result they have specialized OS's to make those applications operate optimally.

*Note: We list these as a separate types of OS, but rarely will an OS have mutually exclusive types.*

# Types of Operating Systems

Talent Accelerator powered by START

**Single/Multi-tasking**

An OS may only need to run one task at a time while another OS needs to work on many tasks in parallel.

*Ex: DOS vs Linux and moderns Windows.*

**Single/Multi-user**

Some OS host many users interacting with one-another. More specialized OS don't need to handle that.

**Embedded**

A *very* simple OS capable of doing one job well. (Arduino, pace-maker, etc).

**Real-time**

For precise timing applications (e.g., life or death situations, or music production!).

# Popular Operating System

There are many popular (used daily by many people) Operating Systems out there. The ones listed below all get the same jobs done (browsing the web, editing documents, playing games, etc), but they approach the problem in a **technically** or **philosophically** different way.

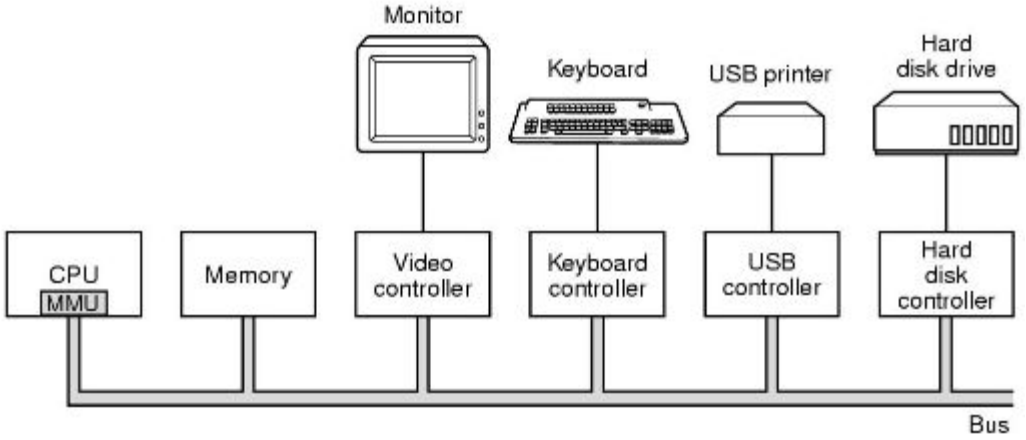
- UNIX
  - Linux
    - Android
    - Debian
    - RHEL
  - MacOS / Darwin
  - FreeBSD
- Windows

# Popular Operating System (1)

The **UNIX family** of operating systems all implement the **POSIX standard** OS interfaces, which has been around since ~1980. POSIX is a very common standard of OS design which makes it easier to write one program that works on different (POSIX) operating systems. This might not seem like such a big deal today but it was revolutionary in the 70's and 80's.

This isn't to say Windows is *bad* for being different, it implements a lot of interesting and smart designs of its own, and it's mostly POSIX compliant!

# Computer Hardware review




Some of the components of a simple personal computer.

# ZShell

# What is Zsh?

A shell is just an interface to your operating system. An interactive shell allows you to type in commands through what is called *standard input*, or stdin, and get output through *standard output* and *standard error*, or stdout and stderr. There are many shells, including Bash, Csh, Ksh, Tcsh, Dash, and Zsh. Each has features based on what its programmers thought would be best for a shell. Whether those features are good or bad is up to you, the end user.

# Installing Zsh

On MacOS, you can install it using MacPorts:

```
$ sudo port install zsh
```

# Setting up Zsh

Zsh is not a terminal emulator; it's a shell that runs inside a terminal emulator. So, to launch Zsh, you must first launch a terminal.Then you can launch Zsh by typing:

```
$ zsh
```

The first time you launch Zsh, you're asked to choose some configuration options. These can all be changed later, so press 1 to continue.

# Setting up Zsh

```
This is the Z Shell configuration function for new
users, zsh-newuser-install.
(q) Quit and do nothing.
(0) Exit, creating the file ~/.zshrc
(1) Continue to the main menu.
```



# Using Zsh

At first, Zsh feels a lot like using Bash, which is unmistakably one of its many features. There are serious differences between, for instance, Bash and Tcsh, so being able to switch between Bash and Zsh is a convenience that makes Zsh easy to try and easy to use at home if you have to use Bash at work or on your server.

# Change directory with Zsh

It's the small differences that make Zsh nice. First, try changing the directory to your Documents folder *without the **cd** command*. It seems too good to be true; but if you enter a directory path with no further instruction, Zsh changes to that directory:

```
% Documents
% pwd
/home/seth/Documents
```

That renders an error in Bash or any other normal shell. But Zsh is far from normal, and this is just the beginning.

# Search with Zsh

When you want to find a file using a normal shell, you probably resort to the find or locate command. At the very least, you may have used ls -R for a recursive listing of a set of directories. Zsh has a built-in feature allowing it to find a file in the current or any other subdirectory.

For instance, assume you have two files called foo.txt. One is located in your current directory, and the other is in a subdirectory called foo. In a Bash shell, you can list the file in the current directory with:

```
$ ls
foo.txt
```

# Search with Zsh

and you can list the other one by stating the subdirectory's path explicitly:

```
$ ls foo
 foo.txt
```

To list both, you must use the -R switch, maybe combined with grep:

```
$ ls -R | grep foo.txt
foo.txt
foo.txt
```

But in Zsh, you can use the ** shorthand:

```
% ls **/foo.txt
 foo.txt
 foo.txt
```


# Break 10 minutes

# Questions?

Thank you for your attention!