Course Introduction

What does a program doing it runs? It executes commands.

Millions (even billions) of times each second, the processor:
* **Fetches** an instruction from memory,
* **Decodes** it (i.e., figures out which instruction this is), and
* **Executes** it (i.e., it does the thing that it is supposed to do, like add two numbers together, access memory, check a condition, jump to a function, and so forth).

After one instruction, the processor moves on to the next, and so on, until the program is finished.

Throughout these courses, we will learn that while a program is running, many other things are happening to make the system easier to use.

What is an Operating System?

The **operating system (OS)** is a piece of software that ensures the system runs smoothly and efficiently. It is responsible for:
* Making it easy to run programs (even allowing you to seemingly run many at the same time)
* Allowing programs to share memory
* Enabling programs to interact with devices
* And other fun, interesting things.

Virtualization

For the most part, the OS performs this via **virtualization**. This means that the OS transforms a physical resource (such a processor, memory, or disk) into a more general, powerful, and user-friendly virtual form. Because of this, the operating system is sometimes called a **Virtual Machine**.

The OS also provides various interfaces (APIs) that you can call in order to tell it what to do (like launch a program, allocate memory, or access a file).

Because **virtualizatio**n allows many programs to run simultaneously (sharing the CPU), access their own instructions and data (sharing memory), and access devices (sharing storage, etc.), the OS is commonly referred to as a **resource manager**.

The operating system's job is to manage the system's resources efficiently, fairly, and with many additional goals in mind, including those mentioned above.

Let's explore some examples to better understand the OS's role.

Fill in the blank to complete the statement below

Which of the following statements describe an Operating System?

CPU Virtualization

The program to the left, `cpu.c` is a simple example code that calls a function, `Spin()`. This function:

- Checks the time repeatedly and returns once it has run for 5 seconds.
- Prints out the string the user passed on the command line
- Repeats forever

Copy and paste the code below into the terminal to compile and run this program as if it were running on a single-processor system. Our program will pass in the string “A” as input.

```
gcc -o cpu cpu.c -Wall
./cpu "A"
```

You should see the following output:

```
A
A
A
A
A
...
```

This program will run forever, so type `^+C` (control + C) to terminate the program.

Running Several Programs

This time, let’s run several instances of the same program, `cpu.c`.

```
./cpu A & ./cpu B & ./cpu C & ./cpu D &
```

Your output should look something like this:

```
[1] 558
[2] 559
[3] 560
[4] 561
C
A
B
D
C
A
B
D
C
...
```

Even though we only have one CPU, all four programs seem to be operating simultaneously! How does it work?

With help from the hardware, the operating system creates the illusion of having a bunch of virtual CPUs.

This **virtualizing** of the CPU, allows several applications to run simultaneously on a single CPU (or a limited collection of them), is the subject of our first course.

We'll use the `pkill` command to terminate all instances of the `cpu` program.

```
pkill cpu
```

To execute applications, stop them, or otherwise instruct the OS which programs to run, certain interfaces (APIs) must be available. These are how most users interact with operating systems.

The ability to run multiple applications at the same time presents new challenges, like **which of the two programs should execute first?**

Policies are used throughout an operating system to address issues like these. We'll explore them during our study of the core procedures used by operating systems (like the ability to run multiple programs at once).

With all this considered, the operating system is, essentially, a **resource manager**.

Memory Virtualization

Memory is a byte array. To read memory, you have to supply an address; to write memory, and you have to specify the data to be written to the given address.

A program's memory is accessed constantly. A program maintains all of its data structures in memory and accesses them via **loads** and **stores** or other explicit memory-accessing instructions. Each program instruction lives in memory, so memory is accessed on each instruction fetch.

Let's look at the program to the left that uses `malloc()` to allocate memory (). Compile and run program `mem.c` using the commands below.

```
gcc -o mem mem.c -Wall
./mem 1
```

This program does a few things:

1. It first allocates memory.
2. Displays the memory address, inserting a zero in the first slot of the freshly allocated
3. Displays the memory address, inserting a zero in the first slot of the freshly allocated memory

Each time it loops, it delays for 1 second and increments the value stored at the address stored in p. Every print statement also prints the **PID (Process Identifier)** for the program. **This PID is unique per running process.**

This program's output should look something like this:

```
(732) addr pointed to by p: 0x5563738f0260
(732) value of p: 1
(732) value of p: 2
(732) value of p: 3
(732) value of p: 4
...
```

Similarly, when we run more than one instance:

```
./mem 1 & ./mem 2
```

The output result should resemble the following:

```
(734) addr pointed to by p: 0x56332d744260
(735) addr pointed to by p: 0x55f921492260
(734) value of p: 1
(735) value of p: 1
(734) value of p: 2
(735) value of p: 2
(734) value of p: 3
(735) value of p: 3
...
```

Each process has its own private **virtual address space** (also known as its address space), which the operating system (OS) maps into the machine's **physical memory**.

A **memory reference** within one running program has no effect on the address space of other processes (or the operating system); the running program has its own physical memory. **Physical memory**, on the other hand, is a shared resource managed by the operating system.

Use the `pkill mem` command to terminate the programs.

Concurrency

Concurrency refers to many jobs that need to be handled simultaneously while working on each one in the same program (i.e., concurrently). Concurrency issues initially originated inside the operating system; as you can see in the **virtualization** examples, the OS juggles many tasks at once, first running one process, then another, and so on. Doing so causes a slew of complex and intriguing issues.

Unfortunately, **concurrency** issues are no longer restricted to the operating system. The same issues may be seen in current **multi-threaded systems**. Let's look at a multi-threaded application as an example.

```
gcc -o thread threads.c -Wall -pthread
./thread 1000
```

Our output should be as follows:

```
Initial value : 0
Final value   : 2000
```

Because each thread increased the counter 1000 times, the counter's final value is 2000 when the two threads are done. Indeed, if the loops' input value is set to N, we should anticipate the program's ultimate output to be 2N. But, as it turns out, life is not so easy. So let's rerun the same program, but this time with larger loop values to see what happens. Run the following program 3 or 4 times and note the output.

```
./thread 100000
```

Are the outputs what you expect? Why do you think this is happening?

The results are related to the execution of the instructions one by one. A crucial part of the program takes three instructions to increment the shared counter:

- one for loading the counter value from memory into a register
- one for incrementing it
- one for storing it back in memory

Funny things happen because these three instructions do not execute at the same time. This is a concurrency problem that we will address in the second course of this certification.

Persistence

Persistence is the specializations's third main subject and fourth course.

Data in system memory can be easily lost due to volatile storage technology like DRAM. When power is off, or the system crashes, all data in memory is lost. As a result, **we require hardware and software to store data persistently**, which is important for any system since users value their data.

The hardware is an **input/output (I/O) device**. Hard drives are popular for storing long-term data, but so are SSDs.

The **file system** is the part of the operating system that handles the disks. It is responsible for storing user files reliably and efficiently.

Unlike the OS's CPU and memory abstractions, the OS does not create a private, virtualized drive for each program. Instead, **users are expected to share file-based information regularly**. For example, while writing a C program, you may use an editor (like Emacs7) to create and change the C file (`emacs -nw main.c`). The compiler may then turn the source code into an executable (e.g., `gcc -o main main.c`). When you're finished, you may start the new executable (for example, `./main`).

So you can see how files are exchanged between processes.

1. Emacs first generates a file that will be used as input by the compiler;
2. the compiler will then use that input file to create a new executable file
3. The new executable will be launched.

As a result, a new program is born!

I/O

The program to the left, `io.c`, creates a file `/tmp/file` containing the string "hello world."

This is done making three **system calls** to the OS.

1. `open()` - opens and creates the file
2. `write()` - writes data to the file
3. `close()` - closes the file and ends writing.

These **system calls** are routed to the **file system** part of the OS. The filesystem then handles the requests and returns a response code to the user.

The file system has to do a lot of work to actually write to disk:

- Figuring out where on disk this new data will live, and
- Keeping track of it in various structures maintained by the file system.
    - This requires sending I/O requests to the underlying storage device to either read or update existing structures.

This OS provides us with a standard, simple way to access devices through **system calls**. To make many common tasks more efficient, file systems utilize a wide range of data structures and access methods, ranging from basic lists to sophisticated b-trees.

We will explore devices, I/O, and file systems in much more detail in the certification course on **Persistence**.

Design Goals

Now, we should have an idea of what an OS does.

- Takes physical **resources** (CPU, memory, disk, etc) and **virtualizes** them.
- Handles tough issues related to **concurrency**.
- Stores files **persistently**, making it safe for the long-term.

Since we want to build OS's we want to focus our design goals and make necessary trade-offs for the most efficient system.

- Build up **abstractions** to make the system easy to use.
- Provide **high-performance**, or **minimize the overheads**
- Provide **protection** and prevent bad behavior between applications, as well as between the OS and applications
    - **Isolation** is the core of protection, and accounts for a lot of what the OS has to do.
- Our OS must be **reliable**, as all applications fail if the OS fails.
- **Energy-efficiency** is important, because our resources are limited.
- Our OS should be **secure**, and protect against harmful applications.
- Our OS should have **Mobility**, as OSs are running on smaller devices

The OS will have various goals depending on how the system is used and will likely be built in slightly different ways. It's okay though, because many of the ideas we will explore on creating an OS apply to a wide range of devices.