# Introduction to Operating Systems

Ryan Liu

July 5, 2025

# Chapter 2

When a program runs, it is just executing instructions. Millions of times a second, the processor fetches an instruction from memory, decodes it, and executes it. Then it moves on to the next instruction. It continues until the program terminates.

Under the hood, processors can execute multiple instructions at once, or execute them out of order and put them back in order, but this book abstracts the execution of instructions to a sequential stream.

The **Operating System (OS)** is in charge of making sure the system operates correctly and efficiently in an easy-to-use matter.

The primary way the OS achieves this is through **virtualization**. You can virtualize traditionally physical resources (processor, memory, disk) and transform it into a virtual counterpart. Sometimes the OS is referred to as a **virtual machine**.

The OS provides APIs programmers can call — **system calls**. If following some standard (POSIX), it is said to provide a **standard library**. Generally on the order of some hundreds of calls.

Since virtualization allows time multiplexing (sharing CPU time) and space multiplexing (separation of virtual addr spaces and sharing RAM), the OS is sometimes known as a **resource manager**. Resources include: CPU, memory, disk

## 2.1 Virtualizing the CPU

Deciding which process should run in the current timesplice is determined by a **policy** of the OS.

## 2.2 Virtualizing Memory

To read mewmory, one must specify an **address** to access the data stored there. To **write** memory, one must also specify the data to be written to the given address. Memory is also accessed on each instruction fetch (since instructions of the program are in memory too).

In the example of page 8, two programs are shown storing things in the same address. How? That is the virtual address. Each program has its own **virtual address space**, private to that process. It is the OS' job to map virtual addresses to physical ones.

## 2.3 Concurrency

Problems of concurrency extend to **multithreaded** programs. A process has threads, and threads share the process' private address space.

Concurrent threads running can lead to race conditions, when modifying shared data.

Concurrency is addressed in greater detail in the second part of the book.

## 2.4 Persistence

In system memory, data can be easily lost, since devices such as DRAM store values in a **volatile** manner (when power turns off, data in memory is lost.)

Persistence storage pardware comes in the form of an **I/O** device, typically an **HDD** or **SSD** in this day and age.

The SW in the OS that usually manages disk is called the **file system**.

Unlike the private address space or private CPU abstraction the OS provides to processes, the OS assumes users will want to share information in files and does not provide a private disk space.

The OS syscalls required are open(), write(), close().

Actually writing to disk is tedious. Interacting with a **device driver** is non-trivial.

Fun fact from the author: many file systems do batch-writes since disk requests are slow. To handle system crashes during writes, most file systems incorporate a write protocol such as **journaling** or **copy-on-write**, ensuring recoverability to a reasonable state in the case of a crash.

## 2.5 Design Goals

Build **abstractions** for ease of use. Emphasize **performance** by **minimizing overheads** of the OS. Consider trade-offs.

Provide **protection** between applications, as well as between the OS and applications. **isolation** between processes is key to protection.

Emphasize **reliability**; the OS must run always.

Other potential goals: **energy-efficiency**, **security**, **mobility (edge computing)**

## 2.6 Some History

Blah Blah

In the beginning, there was no multiplexing, there was a human operator who did scheduling, batch processing was used, computers were not interactive.

Then, kernel and user space was separated, and the idea of a system call was invented. Largely for security reasons.

Remember that syscalls have to jump to kernel mode via a **trap instruction**, or perhaps an **interrupt**. A prespecified **trap handler** (looked up in a **trap table** based on info passed to the OS) is executed.

In kernel mode, the OS has full access to the hardware of the system and can give processes more memory or perform I/O. When it's done, it passes control back to the user via a **return-from-trap** instruction, which reverts to user mode and passes control back to where the application made the system call.

### 2.6.1 The Era of Multiprograming

The OS began time multiplexing the CPU, which helps speed up I/O bound problems. Issues such as **memory protection** (isolation) and dealing with **concurrency** became important. UNIX was developed around this time.

### 2.6.2 The Modern Era

We have PCs woohoo Early OS such as **DOS** did not implement memory protection and shit and early PC OS missed some features. whatever blah blah

Everything is UNIX lmaoo Linux is UNIX based, MAC OS X is UNIX based, Android is a Linux distro and thus UNIX based. Basically UNIX is hella important