1. Introduction to Operating Systems

- Von Neumann model: a running program that executes instructions. Many millions of times every second, the processor fetches an instruction from memory, decodes it (i.e., figures out which instruction this is), and executes it (i.e.like add two numbers together, access memory, check a condition, jump to a function, and so forth). After it is done with this instruction, the processor moves on to the next instruction, and so on, and so on, until the program finally completes.
- There is a body of software that is responsible for making it easy to run programs, allowing programs to share memory, enabling programs to interact with devices, and other fun stuff like that. That body of software is the OS (operating system).
    - The primary way the OS does this is through a general technique that we call **virtualization**.
        - the OS takes a physical resource (such as the processor, or memory, or a disk) and transforms it into a more general, powerful, and easy-to-use virtual form of itself. Thus, we sometimes refer to the operating system as a virtual machine.
    - the OS also provides some interfaces (APIs) that you can call (in order to allow users to tell the OS what to do and thus make use of the features of the virtual machine).
        - A typical OS, in fact, exports a few hundred **system calls** that are available to applications
        - Because the OS provides these calls to run programs, access memory and devices, and other related actions, we also sometimes say that the OS provides a **standard library** to applications.
    - The OS is sometimes known as a **resource manager** as virtualization allows many programs to run (thus sharing the CPU), concurrently access their own instructions and data (thus sharing memory) and many programs to access devices (thus sharing disks and so forth).
    - Each of the CPU, memory, and disk is a resource of the system; it is thus the OS's role to manage those resources

Virtualizing the CPU:

- The OS gives us the illusion that the system has a large number of virtual CPU's; turning a single CPU (or small set of them) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**.
    - For example, if we create a function Spin() which repeatedly checks the time and returns once it has run for a second and then, it prints out the string that the user passed in on the command line, and repeats, forever.
        - If we pass in "A" as the string, once a second has passed, the code prints the input string by us.
        - However, we can also pass in many strings like so: prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
            - Even though we only have one processor, all four of these programs will run at the same time.

Virtualizing Memory:

- The model of physical memory presented by modern machines is very simple. **Memory** is just an array of bytes; to read memory, one must specify an address to be able to access the data stored there; to write (or update) memory, one must also specify the data to be written to the given address.
    - A program keeps all of its data structures in memory, and accesses them through various instructions
    - For example, if we create a program that allocates memory by calling malloc(), which
        - allocates some memory, prints out the address of the memory, and then puts the number zero into the first slot of the newly allocated memory. Then, it loops, delaying for a second and incrementing the value stored at the address held in p. With every print statement, it also prints out what is called the process identifier (the PID) of the running program. This PID is unique per running process.
        - We can again run multiple instances of this same program.
            - each running program has allocated memory at the same address, and yet each seems to be updating the value stored at the address held in p
            - It is as if each running program has its own private memory, instead of sharing the same physical memory with other running programs
- The OS is virtualizing memory in the example above; Each process accesses its own private virtual address space (sometimes just called its address space), which the OS somehow maps onto the physical memory of the machine.
    - A memory reference within one running program does not affect the address space of other processes (or the OS itself); as far as the running program is concerned, it has physical memory all to itself.
        - The reality, however, is that physical memory is a shared resource, managed by the operating system.

Concurrency:

- Concurrency is to refer to a host of problems that arise, and must be addressed, when working on many things at once (i.e., concurrently) in the same program.
- The problems of concurrency are no longer limited just to the OS itself. Modern multi-threaded programs exhibit the same problems.
- We can think of a thread as a function running within the same memory space as other functions, with more than one of them active at a time.

Persistence:

- In system memory, data can be easily lost, as devices such as DRAM store values in a volatile manner; when power goes away or the system crashes, any data in memory is lost. So, we need hardware and software to be able to store data persistently
- The hardware comes in the form of some kind of input/output or I/O device; in modern systems, a **hard drive** is a common repository for long lived information, although **solid-state drives (SSDs)** are making headway in this arena as well.

- The software in the operating system that usually manages the disk is called the **file system**; it is thus responsible for storing any files the user creates in a reliable and efficient manner on the disks of the system.
- In order to actually write to disk, the file system must first figure out where on disk this new data will reside, and then keep track of it in various structures the file system maintains. Doing so requires issuing I/O requests to the underlying storage device, to either read existing structures or update (write) them. The OS however provides a standard and simple way to access devices through its system calls. Thus, the OS is sometimes seen as a standard library.
- To handle the problems of system crashes during writes, most file systems incorporate some kind of intricate write protocol, such as journaling or copy-on-write, carefully ordering writes to disk to ensure that if a failure occurs during the write sequence, the system can recover to a reasonable state afterwards.

Designing an OS:

- One goal in designing and implementing an operating system is to provide high performance;We must strive to provide virtualization and other OS features without excessive overheads.
- The operating system must also run non-stop; when it fails, all applications running on the system fail as well. Must be reliable.

History:

- Usually, on old mainframe systems, one program ran at a time, as controlled by a human operator. Much of what a modern OS would do (e.g., deciding what order to run jobs in) was performed by this operator. This mode of computing was known as batch processing, as a number of jobs were set up and then run in a “batch” by the operator
- In kernel mode, the OS has full access to the hardware of the system and thus can do things like initiate an I/O request or make more memory available to a program.
- multiprogramming became commonplace; Instead of just running one job at a time, the OS would load a number of jobs into memory and switch rapidly between them, thus improving CPU utilization. Memory protection and concurrency became important.
- UNIX was the unifying principle of building small powerful programs that could be connected together to form larger workflows. The shell, where you type commands, provided primitives such as pipes to enable such meta-level programming, and thus it became easy to string together programs to accomplish a bigger task.
- Linux is UNIX based.