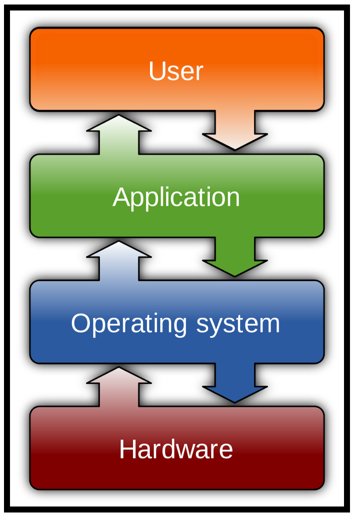
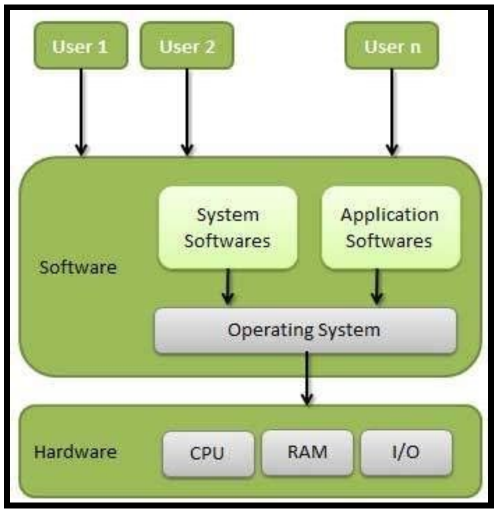
__Lecture Notes:__

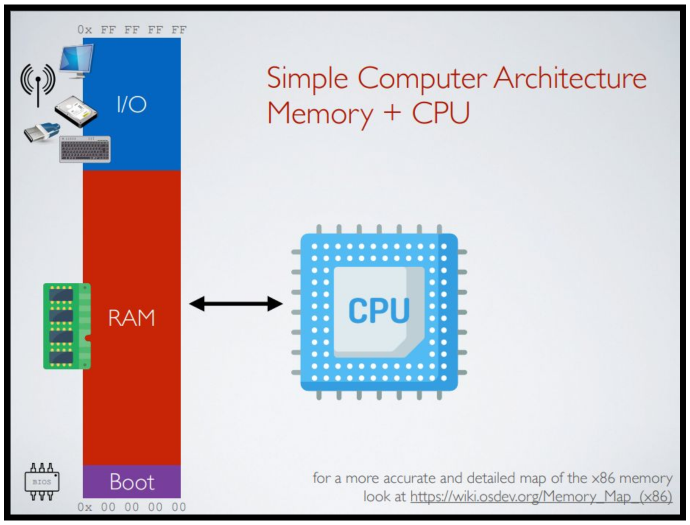
- **Introduction:**
- An **operating system (OS)** is the primary software that manages all the hardware and other software on a computer. The operating system is the one big piece of software running the show, and it's in charge of everything else.





- The operating system interfaces with the computer's hardware and provides services that applications can use.
- An operating system is the core set of software on a device that keeps everything together. Operating systems communicate with the device's hardware. They handle everything from your keyboard and mouse to the Wi-Fi radio, storage devices, and display. In other words, an operating system handles input and output devices. Operating systems use device drivers written by hardware creators to communicate with their devices.
- Operating systems also include a lot of software, things like common system services, libraries, and application programming interfaces (APIs) that developers can use to write programs that run on the operating system.
- The operating system sits in between the applications you run and the hardware, using the hardware drivers as the interface between the two. For example, when an application wants to print something, it hands that task off to the operating system. The operating system sends the instructions to the printer, using the printer's drivers to send the correct signals. The application that's printing doesn't have to care about what printer you have or understand how it works. The OS handles the details.
- The OS also handles multi-tasking, allocating hardware resources among multiple running programs. The operating system controls which processes run, and it allocates them between different CPUs if you have a computer with multiple CPUs or cores, letting multiple processes run in parallel. It also manages the system's internal memory, allocating memory between running applications.
- The **kernel** is the core computer program at the heart of your operating system. This single program is one of the first things loaded when your operating system starts up. It handles allocating memory, converting software functions to instructions for your computer's CPU, and dealing with input and output from hardware devices. The kernel is generally run in an isolated area to prevent it from being tampered with by other software on the computer.

- The following are some of the important functions of an operating System:
    1. Memory Management
    2. Processor Management
    3. Device Management
    4. File Management
    5. Security
    6. Control over system performance
    7. Job accounting
    8. Error detecting aids
    9. Coordination between other software and users
- **Simple Computer Architecture:**
- In a nutshell, a simple computer architecture consists of 2 things:
    1. CPU
    2. Memory


0x FF FF FF FF
I/O
RAM
Boot
0x 00 00 00 00
Simple Computer Architecture
Memory + CPU
CPU
for a more accurate and detailed map of the x86 memory
look at https://wiki.osdev.org/Memory_Map_(x86)

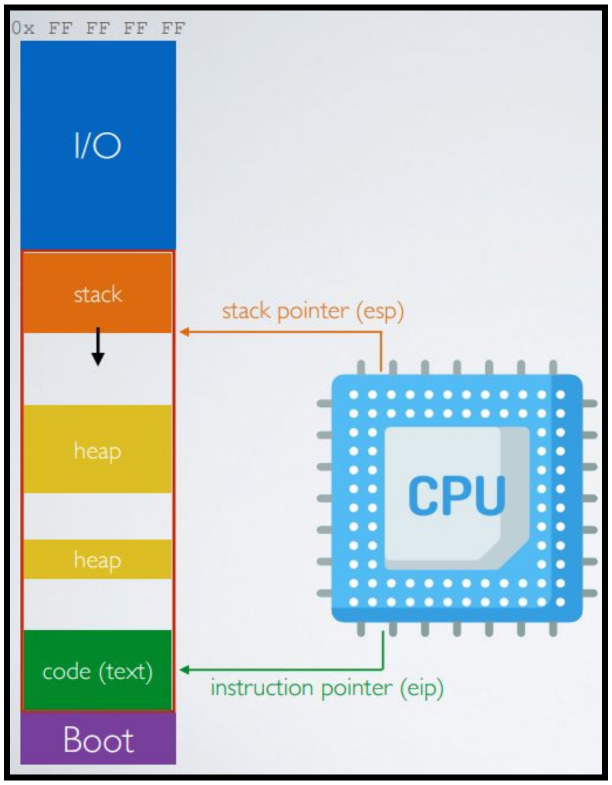
- **CPU:**
- The **central processing unit (CPU)** is where most calculations take place. It is an internal component of the computer that is often referred to as the "brains" of the computer. The CPU does operations/calculations in memory.
- The functions of the CPU are:
    1. The CPU is considered as the brain of the computer.
    2. The CPU performs all types of data processing operations.
    3. The CPU stores data, intermediate results, and instructions (program).
    4. The CPU controls the operation of all parts of the computer.
- The primary components of a CPU are the:
    1. **Control Unit (CU):**
        - The **control unit (CU)** controls and interprets the execution of instructions It extracts instructions from memory and decodes and executes them, calling on the ALU when necessary.
        - This unit controls the operations of all parts of the computer but does not carry out any actual data processing operations.
        - Functions of this unit are:
            1. It is responsible for controlling the transfer of data and instructions among other units of a computer.

2. It manages and coordinates all the units of the computer.
3. It obtains the instructions from the memory, interprets them, and directs the operation of the computer.
4. It communicates with Input/Output devices for transfer of data or results from storage.
5. It does not process or store data.

2. ALU (Arithmetic Logic Unit):

- The **arithmetic logic unit (ALU)** performs arithmetic and logical operations. It is where data is held temporarily and where calculations take place.

- In a nutshell, an OS manages hardware and runs programs. It also:
  1. Creates and manages processes.
  2. Manages access to the memory (including RAM and I/O).
  3. Manages files and directories of the filesystem on disk(s).
  4. Enforces protection mechanisms for reliability and security.
  5. Enables inter-process communication.
- **Processor:**
- A **processor** is the logic circuitry that responds to and processes the basic instructions that drive a computer.
- Each processor has its **Instruction Set Architecture (ISA)**. The ISA is a set of instructions/operations that can be written in assembly code. The ISA provides commands to the processor, to tell it what it needs to do. The ISA consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O.
- The processor executes instructions stored in memory. Each instruction is a bit string that the processor understands as an operation. Some instructions are:
  1. arithmetic
  2. read/write bit strings
  3. bit logic
  4. jumps
- There are approximately 2000 instructions on modern x86-64 processors.
- **Running one program:**
- This is what happens when you are running 1 program:
  1. The CPU has 2 pointers, an **instruction pointer (eip)** and a **stack pointer (esp)**.
  2. The EIP tells the computer where to go next to execute the next command and controls the flow of a program.
  3. The ESP stores the address of the last program request in a stack.

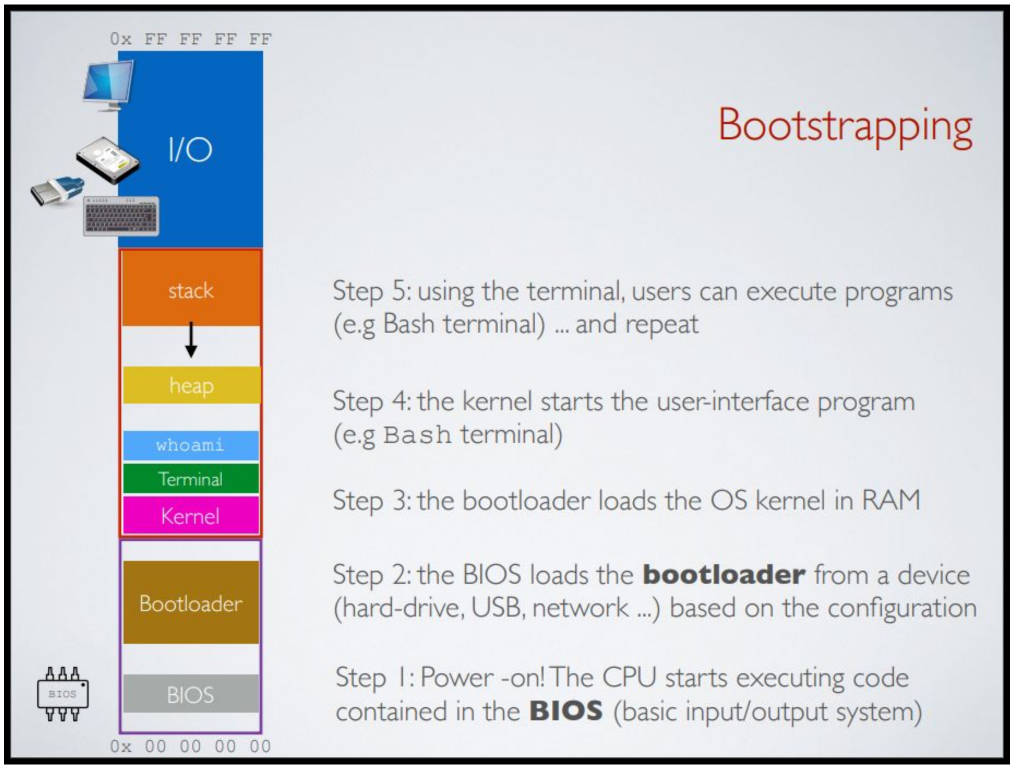


- **Bootstrapping and System Calls:**
- A **bootstrap** is the program that initializes the operating system during startup.
- **Bootstrapping** is the process of loading a set of instructions when a computer is first turned on or booted.
- Without bootstrapping, the computer user would have to download all the software components, including the ones not frequently required. With bootstrapping, only those software components need to be downloaded that are legitimately required and all extraneous components are not required. This process frees up a lot of space in the memory and consequently saves a lot of time.
- The bootstrapping process does not require any outside input to start. Any software can be loaded as required by the operating system rather than loading all the software automatically. The bootstrapping process is performed as a chain. At each stage, it is the responsibility of the simpler and smaller program to load and execute the much more complicated and larger program. This means that the computer system improves in increments by itself. The booting procedure starts with the hardware procedures and then continues onto the software procedures that are stored in the main memory.
- **Bootstrapping Process:**
    1. The CPU executes the code in the boot section of the memory. The boot section has **BIOS (Basic Input/Output System)**. BIOS is the program a personal computer's microprocessor uses to get the computer system started after you turn it on. It also manages data flow between the computer's operating system and attached devices such as the hard disk, video adapter, keyboard, mouse and printer.
    2. The BIOS then loads the **bootloader** from a device (hard-drive, USB, network, etc) based on the configuration. A **bootloader** is a special operating system software that loads into the working memory of a computer after start-up.

Bootloaders are used to boot other operating systems. For this purpose, immediately after a device starts, a bootloader is generally launched by a bootable medium like a hard drive, a CD/DVD or a USB stick. The boot medium receives information from the computer's firmware (BIOS) about where the bootloader is.

3. The bootloader loads the OS kernel in RAM.
4. The kernel starts the user-interface program, such as Bash terminal.
5. Using the terminal, users can execute programs and repeat.

A diagram showing the bootstrapping process:



Bootstrapping

Step 5: using the terminal, users can execute programs (e.g Bash terminal) ... and repeat

Step 4: the kernel starts the user-interface program (e.g `Bash` terminal)

Step 3: the bootloader loads the OS kernel in RAM

Step 2: the BIOS loads the **bootloader** from a device (hard-drive, USB, network ...) based on the configuration

Step 1: Power -on! The CPU starts executing code contained in the **BIOS** (basic input/output system)

- **Note:** User programs do not operate I/O devices directly. The OS abstracts those functionalities and provides them as **system calls**.
- **System Calls:**
- A **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
- A system call is a way for programs to interact with the operating system.
- A computer program makes a system call when it makes a request to the operating system's kernel. System call provides the services of the operating system to the user programs via Application Program Interface (API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.
  I.e. System calls provide user programs with an API to use the services of the operating system.
- Services Provided by System Calls:
  1. Process creation and management
  2. Main memory management

3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

- There are 6 different categories of system calls:
    1. Process control: end, abort, create, terminate, allocate and free memory.
    2. File management: create, open, close, delete, read file etc.
    3. Device management
    4. Information maintenance
    5. Communication
    6. Protection
- Table of Unix System Calls:

Categories of System Calls	**Unix**

Process Control	fork()
	exit()
	wait()

File Manipulation	open()
	read()
	write()
	close()

Device Manipulation	ioctl()
	read()
	write()

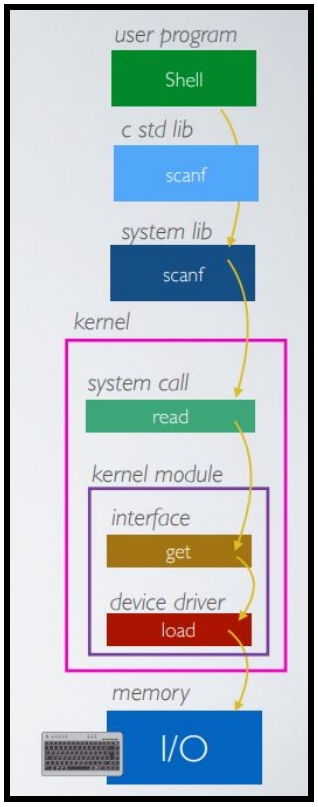
Information Maintenance	getpid()
	alarm()
	sleep()

Communication	pipe()
	shmget()
	mmap()

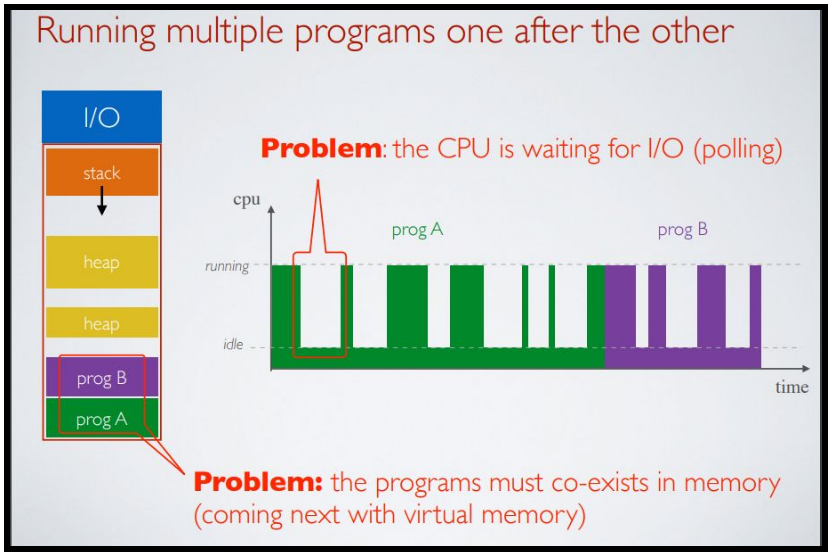
Protection	chmod()
	umask()
	chown()

- **Note:** There are 393 system calls on Linux 3.7.

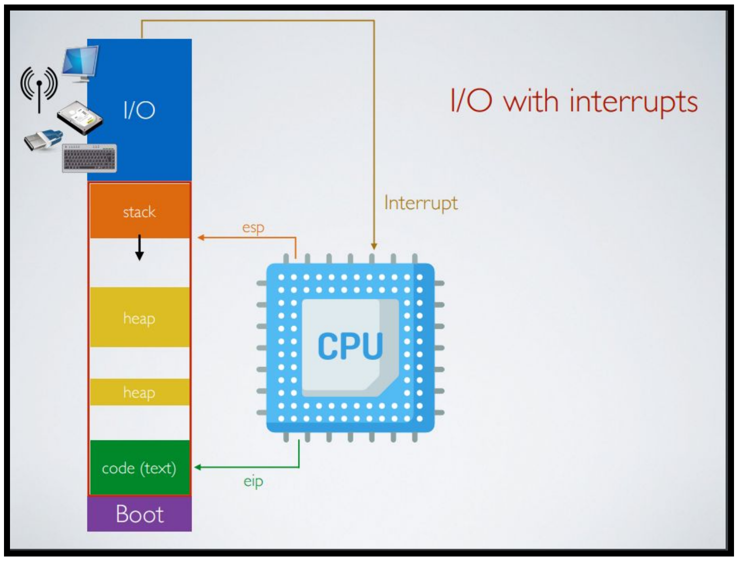
- Diagram of System Call Process:



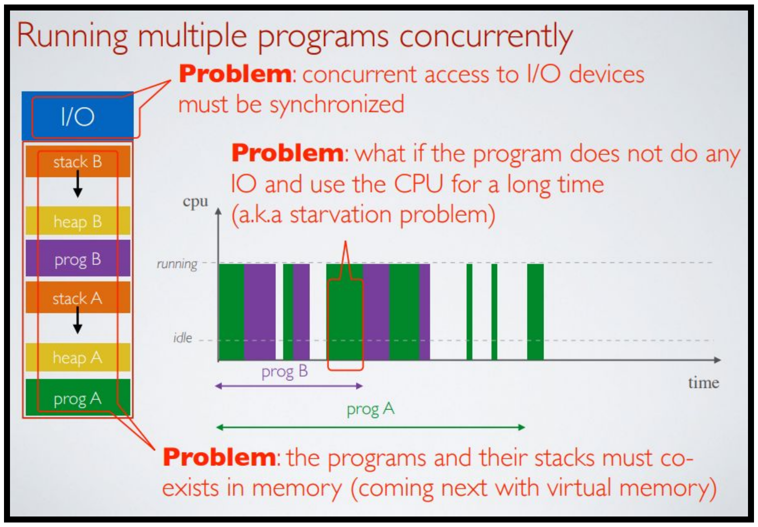
- **Concurrency:**
- **Concurrency** is a way to execute more than one thing at the same time.
  I.e. Concurrency means multiple computations are happening at the same time.
- Consider 2 programs, A and B. Suppose we run program A first, then program B. One problem is that the programs must co-exists in memory. Another problem is that the CPU is often just waiting for I/O. This is called **polling**. Polling is the process where the computer or controlling device waits for an external device to check for its readiness or state, often with low-level hardware. This is wasting time.


Running multiple programs one after the other
I/O
stack
heap
heap
prog B
prog A
Problem: the CPU is waiting for I/O (polling)
cpu
running
idle
prog A
prog B
time
Problem: the programs must co-exists in memory
(coming next with virtual memory)

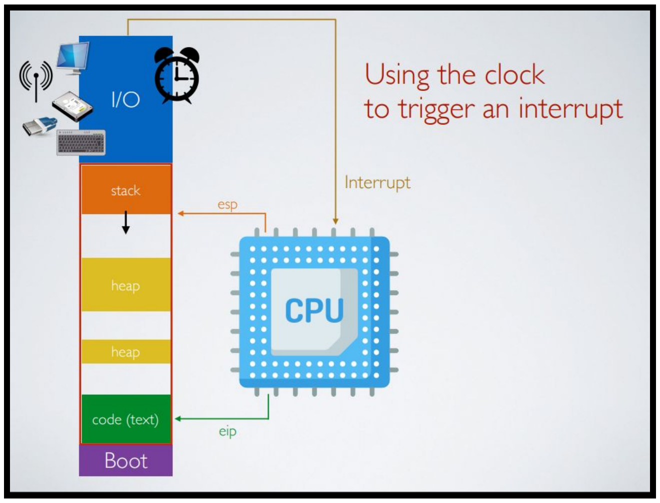
- One solution to problem 2 is to use **interrupts**. **Interrupt** is a hardware mechanism in which a device notifies the CPU that it requires its attention. Interrupt can take place at any time. When the CPU gets an interrupt signal through the indication interrupt-request line, the CPU stops the current process and responds to the interrupt by passing the control to the interrupt handler which services the device. If we use interrupt, we can do something like "Once the data from the I/O is ready, send the CPU a signal through the interrupt line. Once the CPU receives an interrupt, it will stop its current task and focus on what caused the interrupt." Now, the CPU doesn't have to wait and check the status of the I/O. Instead, the CPU can do something else, like executing another program, and only when it receives an interrupt signal will it focus on the original task.



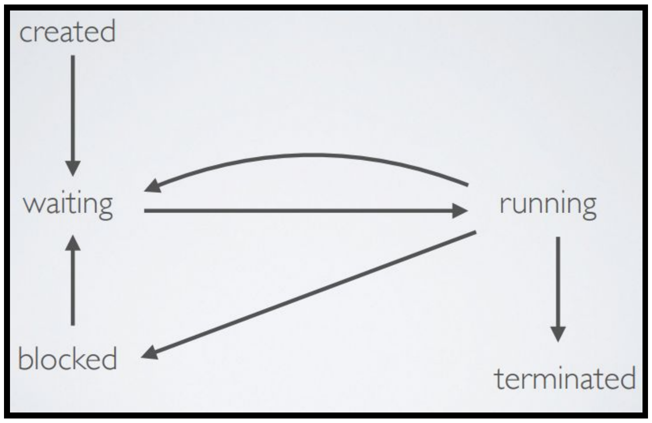
- Now, we can run 2 programs concurrently. However, there are a few problems with running multiple programs concurrently. The first problem is that concurrent access to I/O devices must be synchronized. The second problem is what happens if the program does not do any IO and uses the CPU for a long time. This is known as the **starvation problem**. **Starvation** is when a process ready to run has to wait indefinitely for the CPU because of low priority. The third problem is that the programs and their stacks must coexist in memory.


Running multiple programs concurrently
Problem: concurrent access to I/O devices must be synchronized
Problem: what if the program does not do any IO and use the CPU for a long time (a.k.a starvation problem)
Problem: the programs and their stacks must co-exists in memory (coming next with virtual memory)
I/O
stack B
heap B
prog B
stack A
heap A
prog A
cpu
running
idle
time
prog B
prog A

- One method that we can use to fix problem 2 is to use a clock to trigger an interrupt. Basically, we set the clock to some amount of time and after that amount of time has passed, it will trigger an interrupt. This will cause the CPU to stop executing the current program and switch to another one, usually the next one to be executed.
  E.g. We set the clock to 5 milliseconds, and once 5 milliseconds passed by, the clock will trigger an interrupt, causing the CPU to focus on another program, usually the next one to be executed.
  Note: If there are no other programs that need to be executed, the CPU will focus on the current one still. However, this is a way to allow the CPU to focus on other programs.
  Note: If process B takes a long time before calling an interrupt, it would delay process A.

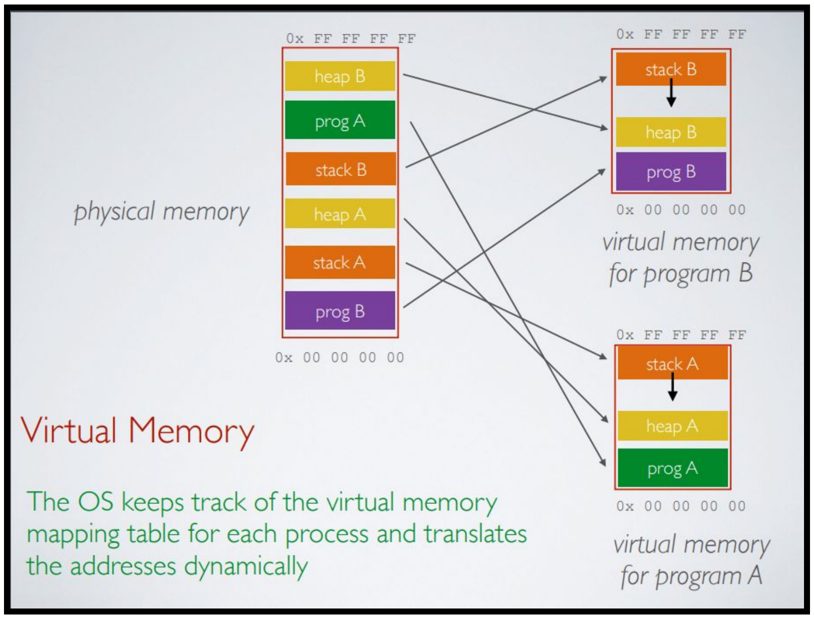


- This is a summary of the different process states:
  1. A process, say A, gets created.
  2. A is waiting for the processor to execute it. Usually, there are many processes waiting and only one process running.
  3. After some time, A is now running. There are a few different paths now:
     a. An interrupt, caused by the clock, could happen which causes the CPU to execute another process. In this case, A has to wait for the CPU again.
     b. A is running, but needs some kind of I/O. In this case, A is blocked until the processor receives an interrupt, which will unblock A. Now, A has to wait for the processor again.
     c. A is terminated.



- With concurrency:
    - From the system perspective, there is better CPU usage resulting in a faster execution overall, but not individually.
    - From the user perspective, programs are executed in parallel.
    - But it requires scheduling, synchronization and some protection mechanisms.
- A **multi-core processor** is a computer processor integrated circuit with two or more separate processing units, called cores, each of which reads and executes program instructions, as if the computer had several processors.
  For example, if your computer has 4 cores, it has 4 processors, each of them can run in parallel. This means that it can execute 4 processes at a time.
  However, there are many logistical problems. For example, if there's an interrupt, which core will receive that interrupt?
  Note: Concurrency is at the OS level while **parallelism** (multi-core) is at the hardware level.
- Some problems that we are going to address during the semester are:
    1. **Scheduling:** Deciding which process to execute when severals are ready to be run. Not all processes are equal. Some are more important than others.
    2. **Synchronization:** Managing concurrent access to resources using semaphores, locks, monitors.
    3. **Communication:** Exchanging messages/data between processes using IPC (sockets & signals).
    4. **Threads:** Lightweight concurrency within a process. It's like a mini process.
- **User Spaces:**
- The **user space** is system memory allocated to running applications.
- The user space is the portion of memory that contains unprivileged processes run by a user. It is strictly separated from kernel space, the portion of memory where privileged operating system kernel processes are executed.
- **Note:** The more RAM your computer has, the more user space is available.
- It is an old problem from older constraints. In the past, there was one computer with multiple users. Suppose there are 2 users, Alice and Bob, on the same computer. What prevents Alice from reading Bob's data, or starting/stopping any of Bob's programs, or accessing any file on the filesystem, or using any I/O device, or changing the system configuration, or rebooting the machine?
- Principle 1: Users have full privileges within their own user space.
- Principle 2: Every access to another user space must go through the kernel via system calls. This is called **complete mediation**. The principle of **complete mediation** requires that all accesses to objects be checked to ensure they are allowed.
- Principle 3: System calls can be allowed or denied based on the system security policy. This is called **access control**.
- Most servers, personal computers, mobile and embedded systems have a single physical user, but not all programs are reliable or trustworthy. It is still a good model to provide reliability and security. Hence, the **multi-user paradigm** is not obsolete.
- **Virtual memory:**
- **Virtual memory** is a memory management capability of an operating system which uses hardware and software to allow a computer to compensate for physical memory shortages, by temporarily transferring data from random access memory (RAM) to disk storage.

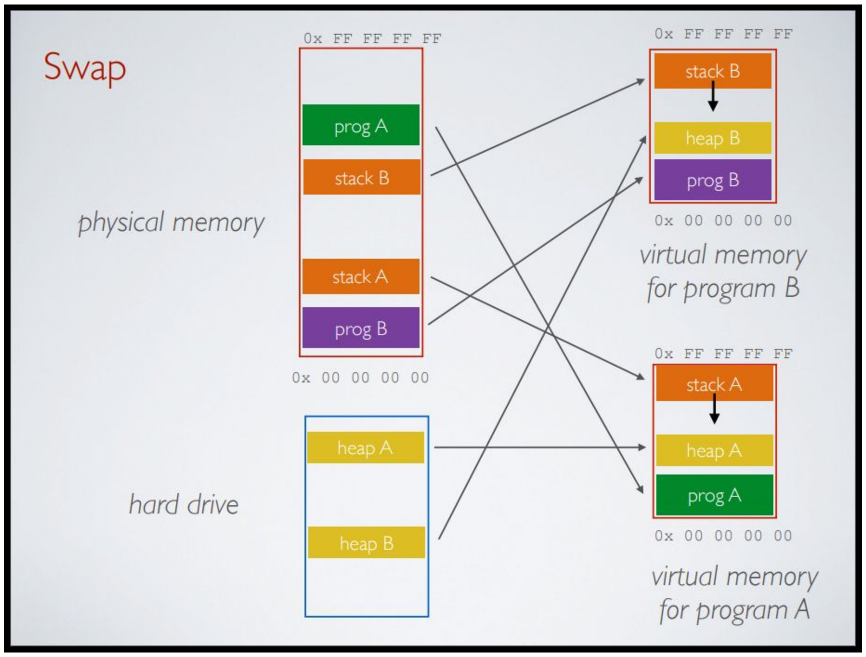
- A computer can address more memory than the amount physically installed on the system. This extra memory is actually called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM.
- How can we make programs and execution contexts coexist in memory? One solution is to place multiple execution contexts at random locations in memory, but having programs placed at random locations is problematic. With virtual memory, the OS keeps track of the virtual memory mapping table for each process and translates the addresses dynamically.



Virtual Memory

The OS keeps track of the virtual memory mapping table for each process and translates the addresses dynamically

- Another program arises if we run out of memory because of too many concurrent programs. One solution is to swap memory by moving some data to the disk. Managing memory becomes very complex but necessary.



- **File System:**
- A **file system** is a process that manages how and where data on a storage disk, typically a hard disk drive, is stored, accessed and managed. It is a logical disk component that manages a disk's internal operations as it relates to a computer and is abstract to a human user.