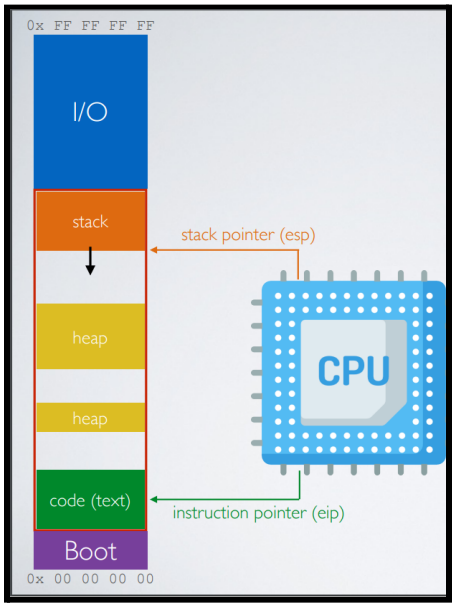
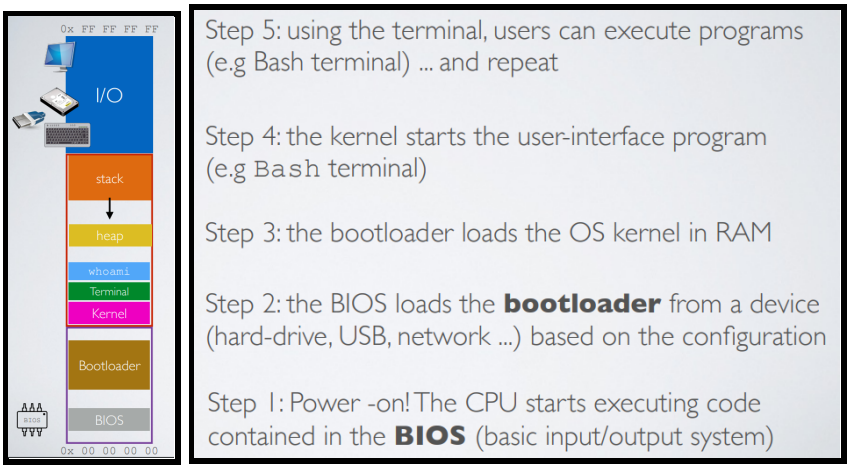
Lecture Notes:

- **Computer Architecture:**
- A simple computer architecture consists of Memory + CPU.
- A **CPU/Processor** is the logic circuitry that responds to and processes the basic instructions that drive a computer. The CPU is seen as the main and most crucial integrated circuit chip in a computer, as it is responsible for interpreting most computer commands. CPUs will perform most basic arithmetic, logic and I/O operations, as well as allocate commands for other chips and components running in a computer.
- Each processor has its **Instruction Set Architecture (ISA)**.
- The ISA provides commands to the processor, to tell it what it needs to do. It consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, and exception handling, and external I/O.
- Each instruction is a bit string that the processor understands as an operation:
    - arithmetic
    - read/write bit strings
    - bit logic
    - jumps
- There are about 2000 instructions on modern x86-64 processors.
- This is what happens when you run 1 program:

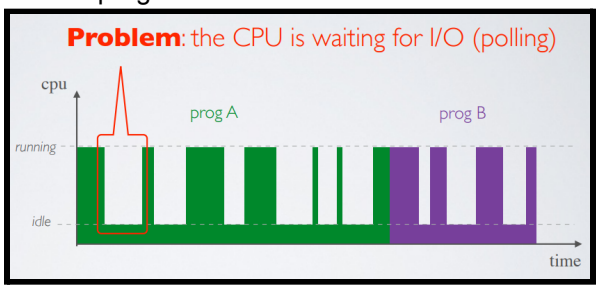


- A **stack** is a reserved area of memory used to keep track of a program's internal operations, including functions, return addresses, passed parameters, etc. It is the memory set aside as scratch space for a thread of execution. A stack is usually maintained as a "last in, first out" (LIFO) data structure, so that the last item added to the structure is the first item used.
- The **stack pointer** is used to indicate the location of the last item put onto the stack.
- The **heap** is memory set aside for dynamic allocation. I.e. When you use malloc, it goes on the heap. Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time.
- **Bootstrapping:**
- A **bootstrap program** is the first code that is executed when the computer system is started. It is the program that initializes the operating system during startup. The entire operating system depends on the bootstrap program to work correctly as it loads the operating system.
- 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.

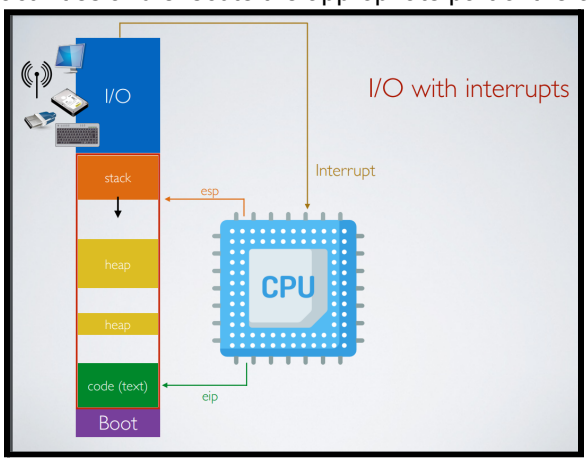


- Step 1: Power-on. The CPU starts executing code contained in the BIOS (basic input/output system).
  Step 2: The BIOS loads the bootloader from a device (hard-drive, USB, network, etc) based on the configuration.
  Step 3: The bootloader loads the OS kernel in RAM.
  Step 4: The kernel starts the user-interface program (e.g Bash terminal).
  Step 5: Using the terminal, users can execute programs (e.g Bash terminal) and repeat.
- **Concurrency:**
- **Concurrency** is the execution of the multiple instruction sequences at the same time. It happens in the operating system when there are several process threads running in parallel. The running process threads always communicate with each other through shared memory or message passing.
- Concurrency helps in techniques like coordinating execution of processes, memory allocation and execution scheduling for maximizing throughput.
- Problems in concurrency:
  - **Sharing global resources:** If two processes both make use of a global variable and both perform read and write on that variable, then the order in which various read and write are executed is critical.
  - **Optimal allocation of resources:** It is difficult for the operating system to manage the allocation of resources optimally.
  - **Locking the channel:** It may be inefficient for the operating system to simply lock the channel and prevent its use by other processes.
- Advantages of concurrency:
  - **Running of multiple applications:** Concurrency means that the OS can run multiple applications at the same time.
  - **Better resource utilization:** Concurrency enables that resources that are unused by one application can be used for other applications.
  - **Better average response time:** Without concurrency, each application has to be run to completion before the next one can be run.
  - **Better performance:** Concurrency enables better performance by the operating system. If one application only uses the processor and another application only uses the disk drive then the time to run both applications concurrently will be shorter than the time to run each application consecutively.
- Drawbacks of concurrency:
  - Coordinating multiple applications is difficult and complex.
  - Additional performance overheads and complexities in operating systems are required for switching among applications.
  - Sometimes running too many applications concurrently leads to severely degraded performance.

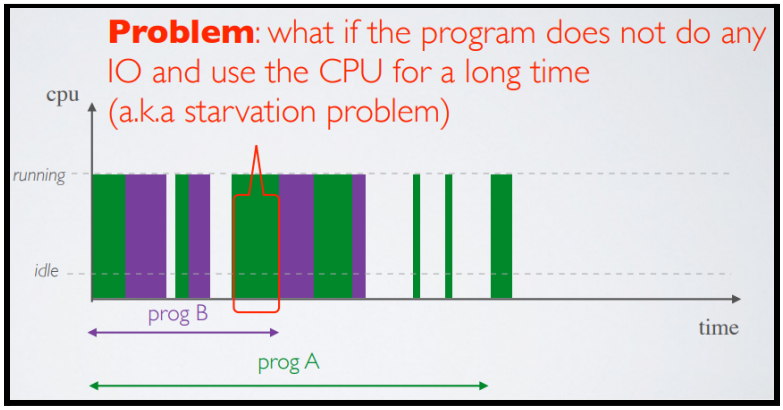
- Consider the following example: We are running 2 programs, A and B. If we run B after finishing running A, we're wasting a lot of time. This is because most programs have to deal with I/O, which is slow. So, we'll have a lot of time where we'll just be waiting for A to communicate with another program.



One way to solve this issue is to use **interrupts**. An interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention. It alerts the processor to a high-priority process requiring interruption of the current working process. Interrupts are signals sent to the CPU by external devices, normally I/O devices. They tell the CPU to stop its current activities and execute the appropriate part of the operating system.



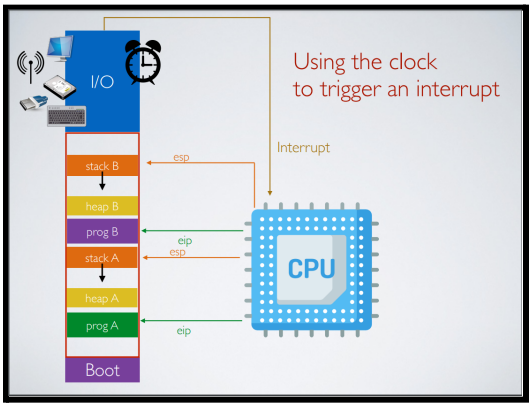
What happens is that while the CPU is waiting for one program, say A, to finish its communication, it will move on to another program. However, once A finishes its communication, it will generate an interrupt to let the CPU know it has finished its communication and to stop what it's currently doing and continue executing it. This way, while programs are in the middle of communicating (I/O), the CPU can work on other programs and won't have to waste time.



From the picture above, you can see that while A is in the middle of I/O, the CPU works on B.

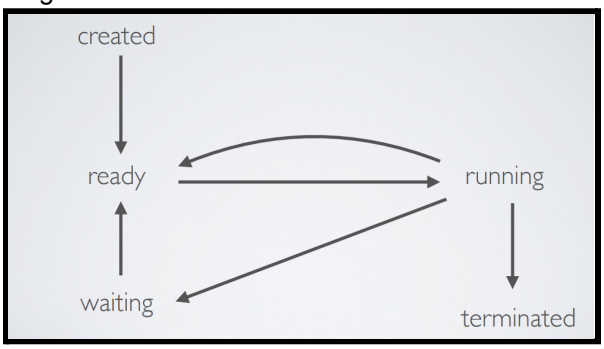
However, there are a few problems with interrupts:

1. Concurrent access to I/O devices must be synchronized. Suppose that both A and B need access to the same I/O device at the same time, what happens then?
2. What if the program does not do any IO and use the CPU for a long time?
   This is called the **starvation problem**.
   To fix this issue, we will use a system clock that triggers an interrupt at a specific time interval. We do this to avoid having the CPU stuck on one program for too long. Each time the system clock triggers an interrupt, the CPU will switch to another program.



Note: The CPU is only executing (running) 1 program at a time.

- Program states:



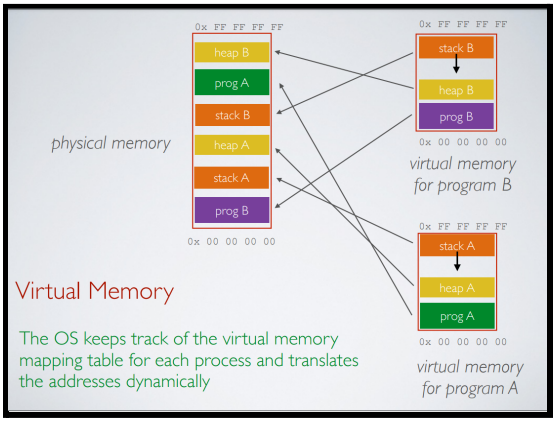
After a program is created, it will go to the **ready queue** to let the CPU know that it's ready. Then, at some point, the CPU will run that program. From there, the program has 3 options:

1. It is completely done running, so it is terminated.
2. It needs to wait for I/O, so it goes to waiting. After the I/O is complete, it will go back into the ready queue.
3. It is done running a section, generates an interrupt, and goes back to the ready queue.

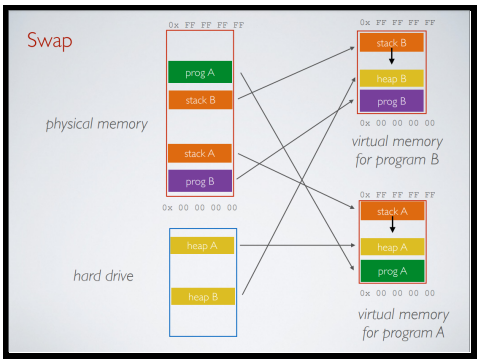
- **Note:** Even though most modern computers/laptops have multiple cores, the same issues still apply since our computers/laptops are running hundreds of programs at a time. However, with multiple cores, interrupts become more difficult cause we have to decide which core will handle it.
- With concurrency:
  - From the system perspective, there is better CPU usage resulting in a faster execution overall but not individually.
  - From the user perspective programs seem to be executed in parallel.
  - But it requires scheduling, synchronization and some protection mechanisms.

- **User Programs:**
- Consider this example: You are writing a Bash shell that reads keyboard inputs from the user. However, there are many ways the user can send the keyboard inputs:
    1. The one connected to the USB (USB keyboard).
    2. The one connected to the bluetooth (Bluetooth keyboard).
    3. The remote one connected to the network (SSH).

  How do you know which I/O device to listen to?
- User programs do not operate I/O devices directly.
- The OS abstracts those functionalities and provides them as **system calls**.
  I.e. The OS creates APIs for user programs to use so they don't interact with the kernel directly.
- A system call is a mechanism that provides the interface between a process and the operating system. It is a programmatic method in which a computer program requests a service from the kernel of the OS.
- System call offers the services of the operating system to the user programs via an API. System calls are the only entry points for the kernel system.
- System calls provide user programs with an API to use the services of the operating system.
- There are 6 categories of system calls:
    1. Process control
    2. File management
    3. Device management
    4. Information/maintenance (system configuration)
    5. Communication (IPC)
    6. Protection
- There are 393 system calls on Linux 3.7.
- **Virtual Memory:**
- A computer can address more memory than the amount physically installed on the system. This extra memory is called **virtual memory** and it is a section of a hard disk that's set up to emulate the computer's RAM. It is done by treating a part of secondary memory as the main memory. In virtual memory, the user can store processes with a bigger size than the available main memory.
- Therefore, instead of loading one long process in the main memory, the OS loads the various parts of more than one process in the main memory.
- The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory serves two purposes. First, it allows us to extend the use of physical memory by using a disk. Second, it allows us to have memory protection, because each virtual address is translated to a physical address.
- Consider this example: We want to make programs and execution contexts coexist in memory. Placing multiple execution contexts (stack and heap) at random locations in memory is not a problem as long as you have enough memory. However having programs placed at random locations is problematic. We can use virtual memory to solve this problem.



- Another problem arises if we run out of memory because of too many concurrent programs. We can use virtual memory to solve this too. We can swap memory by moving some data to the disk. Managing memory becomes very complex but necessary.



- **File System:**
- A **file** is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.
- **Operating System:**
- In a nutshell, an OS:
    - Manages hardware and runs programs.
    - Creates and manages processes.
    - Manages access to the memory (including RAM and I/O).
    - Manages files and directories of the filesystem on disk(s).
    - Enforces protection mechanisms for reliability and security.
    - Enables inter-process communication.

Textbook Notes:

- **Introduction to Operating Systems:**
    - Introduction to Operating Systems:
    - The primary way the operating system (OS) makes it easy to seemingly run many at the same time, allows programs to share memory, and enables programs to interact with devices is through a general technique that we call **virtualization**. That is, 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.
    - In order to allow users to tell the OS what to do and thus make use of the features of the virtual machine, the OS also provides some APIs that you can call. 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.
    - Finally, because virtualization allows many programs to run, and many programs to **concurrently** access their own instructions and data, and many programs to access devices, the OS is sometimes known as a resource manager. Each of the CPU, memory, and disk is a resource of the system; it is thus the operating system's role to manage those resources, doing so efficiently or fairly or indeed with many other possible goals in mind.
    - Virtualizing The CPU:
    - It turns out that the operating system, with some help from the hardware, creates the illusion that the system has a very large number of virtual CPUs. Turning a 1 or more CPU(s) into a seemingly infinite number of CPUs and thus allowing many programs to seemingly run at once is what we call **virtualizing the CPU**.
    - However, the ability to run multiple programs at once raises all sorts of new questions. For example, if two programs want to run at a particular time, which should run? This question is answered by a **policy** of the OS. Policies are used in many different places within an OS to answer these types of questions.

- 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.
- Memory is accessed all the time when a program is running. A program keeps all of its data structures in memory, and accesses them through various instructions, like loads and stores or other explicit instructions that access memory in doing their work. Don't forget that each instruction of the program is in memory too; thus memory is accessed on each instruction fetch.
- When the OS is virtualizing memory, each process has and accesses its own private **virtual address space**/**address space**, which the OS 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.
- Persistence:
- In system memory, data can be easily lost, such as when power goes away or the system crashes, any data in memory is lost. Thus, we need hardware and software to be able to store data persistently; such storage is thus critical to any system as users care a great deal about their data.
- 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 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.
- Unlike the abstractions provided by the OS for the CPU and memory, the OS does not create a private, virtualized disk for each application. Rather, it is assumed that oftentimes, users will want to share information that is in files.
- For performance reasons, most file systems first delay such writes for a while, hoping to batch them into larger groups.
- 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 reasonable state afterwards.
- Design Goals:
- An OS takes physical resources, such as a CPU, memory, or disk, and virtualizes them, handles tough and tricky issues related to concurrency and stores files persistently, thus making them safe over the long-term.
- One of the most basic goals is to build up some abstractions in order to make the system convenient and easy to use.
- Another goal in designing and implementing an operating system is to provide high performance; another way to say this is our goal is to minimize the overheads of the OS. Virtualization and making the system easy to use are well worth it, but not at any cost; thus, we must strive to provide virtualization and other OS features without excessive overheads.
- Another goal will be to provide protection between applications, as well as between the OS and applications. Because we wish to allow many programs to run at the same time, we want to make sure that the malicious or accidental bad behavior of one does not harm others; we certainly don't want an application to be able to harm the OS itself. Protection is at the heart of one of the main principles underlying an operating system, which is that of isolation; isolating processes from one another is the key to protection and thus underlies much of what an OS must do.

    - The operating system must also run non-stop. When it fails, all applications running on the system fail as well. Because of this dependence, operating systems often strive to provide a high degree of reliability.
- **The Abstraction - The Process:**
    - Introduction:
    - A **process** is a running program.
    - The program itself is a lifeless thing: it just sits there on the disk, a bunch of instructions and maybe some static data, waiting to spring into action. It is the operating system that takes these bytes and gets them running, transforming the program into something useful.
    - The OS creates an illusion of having endless CPUs by virtualizing the CPU. By running one process, then stopping it and running another, and so forth, the OS can promote the illusion that many virtual CPUs exist when in fact only 1 or a few CPUs exist. This basic technique, known as **time sharing** of the CPU, allows users to run as many concurrent processes as they would like; the potential cost is performance, as each will run more slowly if the CPU(s) must be shared.
    - **Time sharing** is a basic technique used by an OS to share a resource. By allowing the resource to be used for a little while by one entity, and then a little while by another, and so forth, the resource in question can be shared by many. The counterpart of time sharing is **space sharing**, where a resource is divided in space among those who wish to use it. For example, disk space is naturally a spaceshared resource; once a block is assigned to a file, it is normally not assigned to another file until the user deletes the original file.
    - To implement virtualization of the CPU, and to implement it well, the OS will need both some low-level machinery and some high-level intelligence. We call the low-level machinery **mechanisms**. Mechanisms are low-level methods or protocols that implement a needed piece of functionality.
    - On top of these mechanisms resides some of the intelligence in the OS, in the form of **policies**. Policies are algorithms for making some kind of decision within the OS. For example, given a number of possible programs to run on a CPU, which program should the OS run? A scheduling policy in the OS will make this decision, likely using historical information, workload knowledge, and performance metrics to make its decision.
    - The Abstraction - A Process:
    - The abstraction provided by the OS of a running program is something we will call a **process**.
    - To understand what constitutes a process, we thus have to understand its **machine state**: what a program can read or update when it is running. At any given time, what parts of the machine are important to the execution of this program?
    - One obvious component of a machine state that comprises a process is its memory. Instructions lie in memory; the data that the running program reads and writes sits in memory as well. Thus the memory that the process can address (called its **address space**) is part of the process.
    - Also part of the process's machine state are **registers**; many instructions explicitly read or update registers and thus clearly they are important to the execution of the process. Note that there are some particularly special registers that form part of this machine state. For example, the program counter tells us which instruction of the program will execute next. Similarly a stack pointer and associated frame pointer are used to manage the stack for function parameters, local variables, and return addresses.
    - Finally, programs often access persistent storage devices too. Such I/O information might include a list of the files the process currently has open.
    - Process Creation - A Little More Detail:
    - The first thing that the OS must do to run a program is to load its code and any static data into memory, into the address space of the process. Programs initially reside on disk in some kind of executable format; thus, the process of loading a program and static data into memory requires the OS to read those bytes from disk and place them in memory somewhere.

- In early operating systems, the loading process is done **eagerly** (All at once before running the program). Modern OSes perform the process **lazily** (by loading pieces of code or data only as they are needed during program execution).
- Once the code and static data are loaded into memory, there are a few other things the OS needs to do before running the process.
- Some memory must be allocated for the program's run-time stack.
- The OS may also allocate some memory for the program's heap. In C programs, the heap is used for explicitly requested dynamically-allocated data; programs request such space by calling malloc() and free it explicitly by calling free(). The heap will be small at first; as the program runs, and requests more memory via the malloc() library API, the OS may get involved and allocate more memory to the process to help satisfy such calls.
- The OS will also do some other initialization tasks, particularly as related to I/O. For example, in UNIX systems, each process by default has three open file descriptors, for standard input, output, and error. These descriptors let programs easily read input from the terminal and print output to the screen.
- By loading the code and static data into memory, by creating and initializing a stack, and by doing other work as related to I/O setup, the OS has now finally set the stage for program execution. It thus has one last task: to start the program running at the entry point, namely main(). By jumping to the main() routine, the OS transfers control of the CPU to the newly-created process, and thus the program begins its execution.
- Process States:
- There are 3 **states** a process can be in:
    1. **Running:** In the running state, a process is running on a processor. This means it is executing instructions.
    2. **Ready:** In the ready state, a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
    3. **Blocked/Waiting:** In the blocked state, a process has performed some kind of operation that makes it not ready to run until some other event takes place. A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.
- A process can be moved between the ready and running states at the discretion of the OS. Being moved from ready to running means the process has been **scheduled**. Being moved from running to ready means the process has been **descheduled**. Once a process has become blocked, the OS will keep it as such until some event occurs. At that point, the process moves to the ready state again.
- Data Structures:
- The OS is a program, and like any program, it has some key data structures that track various relevant pieces of information. To track the state of each process, for example, the OS likely will keep some kind of process list for all processes that are ready and some additional information to track which process is currently running. The OS must also track, in some way, blocked processes; when an I/O event completes, the OS should make sure to wake the correct process and ready it to run again.
- The **register context** will hold, for a stopped process, the contents of its registers. When a process is stopped, its registers will be saved to this memory location. By restoring these registers, the OS can resume running the process.
- Sometimes a system will have an **initial state** that the process is in when it is being created. Also, a process could be placed in a **final state** where it has exited but has not yet been cleaned up, sometimes called a **zombie state**.

- **The Abstraction - Address Spaces:**
  - The Address Space:
  - An **address space** is the running program's view of memory in the system. The address space of a process contains all of the memory state of the running program.
  - **Note:** When we describe the address space, what we are describing is the abstraction that the OS is providing to the running program.
  - One question/problem is: "How can the OS build this abstraction of a private, potentially large address space for multiple running processes all sharing memory on top of a single, physical memory?"

    The OS uses **virtual memory** to solve this problem. Virtual memory is a section of a hard disk that's set up to emulate the computer's RAM. It is done by treating a part of secondary memory as the main memory. In virtual memory, the user can store processes with a bigger size than the available main memory.
  - Goals:
  - One major goal of a virtual memory system is **transparency**. The OS should implement virtual memory in a way that is invisible to the running program. Thus, the program shouldn't be aware of the fact that memory is virtualized; rather, the program behaves as if it has its own private physical memory. Behind the scenes, the OS and hardware does all the work to multiplex memory among many different jobs, and hence implements the illusion.
  - Another goal of VM is **efficiency**. The OS should strive to make the virtualization as efficient as possible, both in terms of time and space.
  - Finally, a third VM goal is **protection**. The OS should make sure to protect processes from one another as well as the OS itself from processes. When one process performs a load, a store, or an instruction fetch, it should not be able to access or affect in any way the memory contents of any other process or the OS itself. Protection thus enables us to deliver the property of isolation among processes; each process should be running in its own isolated cocoon, safe from the ravages of other faulty or even malicious processes. **Isolation** is a key principle in building reliable systems. If two entities are properly isolated from one another, this implies that one can fail without affecting the other. Operating systems strive to isolate processes from each other and in this way prevent one from harming the other. By using memory isolation, the OS further ensures that running programs cannot affect the operation of the underlying OS.