**Operating System**

**Unit 1. Overview of Operating System**

1. **Introduction to Operating System**

Operating System (or shortly OS) primarily provides services for running applications on a computer system.

Operating systems are there from the very first computer generation and they keep evolving with time. In this chapter, we will discuss some of the important types of operating systems which are most commonly used.

## Batch operating system

The users of a batch operating system do not interact with the computer directly. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator. To speed up processing, jobs with similar needs are batched together and run as a group. The programmers leave their programs with the operator and the operator then sorts the programs with similar requirements into batches.

The problems with Batch Systems are as follows –

* Lack of interaction between the user and the job.
* CPU is often idle, because the speed of the mechanical I/O devices is slower than the CPU.
* Difficult to provide the desired priority.

## Time-sharing operating systems

Time-sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of Multiprogrammed batch systems, the objective is to maximize processor use, whereas in Time-Sharing Systems, the objective is to minimize response time.

Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response. For example, in a transaction processing, the processor executes each user program in a short burst or quantum of computation. That is, if n users are present, then each user can get a time quantum. When the user submits the command, the response time is in few seconds at most.

The operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems are as follows −

* Provides the advantage of quick response.
* Avoids duplication of software.
* Reduces CPU idle time.

Disadvantages of Time-sharing operating systems are as follows −

* Problem of reliability.
* Question of security and integrity of user programs and data.
* Problem of data communication.

## Distributed operating System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as **loosely coupled systems** or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

The advantages of distributed systems are as follows −

* With resource sharing facility, a user at one site may be able to use the resources available at another.
* Speedup the exchange of data with one another via electronic mail.
* If one site fails in a distributed system, the remaining sites can potentially continue operating.
* Better service to the customers.
* Reduction of the load on the host computer.
* Reduction of delays in data processing.

## Network operating System

A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks.

Examples of network operating systems include Microsoft Windows Server 2003, Microsoft Windows Server 2008, UNIX, Linux, Mac OS X, Novell NetWare, and BSD.

The advantages of network operating systems are as follows −

* Centralized servers are highly stable.
* Security is server managed.
* Upgrades to new technologies and hardware can be easily integrated into the system.
* Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems are as follows −

* High cost of buying and running a server.
* Dependency on a central location for most operations.
* Regular maintenance and updates are required.

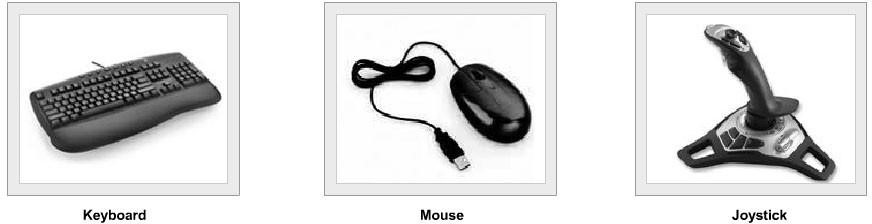
## Real Time operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the **response time**. So in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

* 1. **Need for an OS:**

The primary need for the OS arises from the fact that user needs to be provided with services and OS ought to facilitate the provisioning of these services. The central part of a computer system is a processing engine called CPU. A system should make it possible for a user’s application to use the processing unit. A user application would need to store information. The OS makes memory available to an application when required. Similarly, user applications need use of input facility to communicate with the application. This is often in the form of a key board, or a mouse or even a joy stick (if the application is a game for instance).



The output usually provided by a video monitor or a printer as some times the user may wish to generate an output in the form of a printed document. Output may be available in some other forms. For example it may be a video or an audio file.

Let us consider few applications.

* Document Design
* Accounting
* E-mail
* Image processing
* Games

We notice that each of the above application requires resources for

* Processing information
* Storage of Information
* Mechanism to inputting information
* Provision for outputting information
* These service facilities are provided by an operating system regardless of the nature of application.

The OS offers generic services to support all the above operations. These operations in turn facilitate the applications mentioned earlier. To that extent an OS operation is application neutral and service specific.

* 1. **User and System View:**

From the user point of view the primary consideration is always the convenience. It should be easy to use an application. In launching an application, it helps to have an icon which gives a clue which application it is. We have seen some helpful clues for launching a browser, e-mail or even a document preparation application. In other words, the human computer interface which helps to identify an application and its launch is very useful. This hides a lot of details of the more elementary instructions that help in selecting the application. Similarly, if we examine the programs that help us in using input devices like a key board – all the complex details of character reading program are hidden from the user. The same is true when we write a program. For instance, when we use a programming language like C, a printf command helps to generate the desired form of output. The following figure essentially depicts the basic schema of the use of OS from a user stand point. However, when it comes to the view point of a system, the OS needs to ensure that all the system users and applications get to use the facilities that they need.

USER 1

USER 2

**………**

USER n

SYSTEM AND APPLICATION PROGRAMS

OPERATING SYSTEM

COMPUTER HARDWARE

Also, OS needs to ensure that system resources are utilized efficiently. For instance, there may be many service requests on a Web server. Each user request need to be serviced. Similarly, there may be many programs residing in the main memory. The system need to determine which programs are active and which need to await some form of input or output. Those that need to wait can be suspended temporarily from engaging the processor. This strategy alone enhances the processor throughput. In other words, it is important for an operating system to have a control policy and algorithm to allocate the system resources.

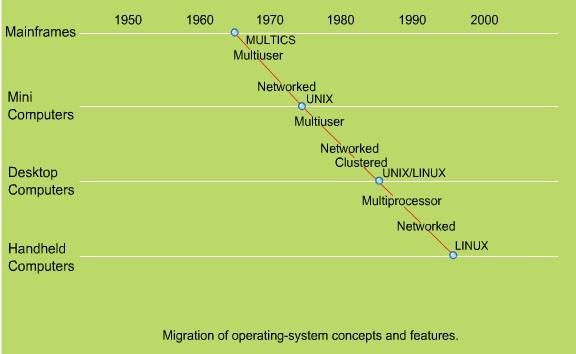
* 1. **The Evolution:**

It would be worthwhile to trace some developments that have happened in the last four to five decades. In the 1960s, the common form of computing facility was a mainframe computer system. The mainframe computer system would be normally housed in a computer center with a controlled environment which was usually an air conditioned area with a clean room like facility. The users used to bring in a deck of punched cards which encoded the list of program instructions.

The mode of operation was as follows:

* User would prepare a program as a deck of punched cards.
* The header cards in the deck were the “job control” cards which would indicate which compiler was to be used (like Fortran / Cobol compilers).
* The deck of cards would be handed in to an operator who would collect such jobs from various users.
* The operators would invariably group the submitted jobs as Fortran jobs, Cobol jobs etc. In addition, these were classified as “long jobs” that required considerable processing time or short jobs which required a short and limited computational time.

Each set of jobs was considered as a batch and the processing would be done for a batch. Like for instance there may be a batch of short Fortran jobs. The output for each job would be separated and turned over to users in a collection area. This scenario clearly shows that there was no interactivity. Users had no direct control. Also, at any one time only one program would engage the processor. This meant that if there was any input or output in between processing then the processor would wait idling till such time that the I/O is completed. This meant that processor would idling most of the time as processor speeds were orders of magnitude higher than the input or output or even memory units. Clearly, this led to poor utilization of the processor. The systems that utilized the CPU and memory better and with multiple users connected to the systems evolved over a period of time as shown in Table.

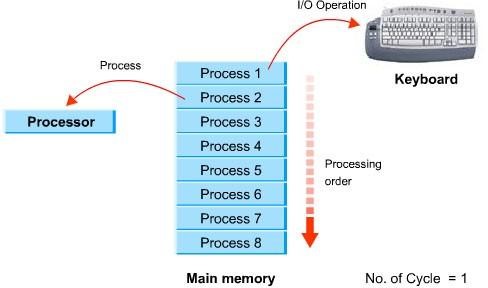


At this time we would like to invoke Von - Neumann principle of stored program operation. For a program to be executed it ought to be stored in the memory. In the scheme of things discussed in the previous paragraph, we notice that at any time only one program was kept in the memory and executed. In the decade of 70s this basic mode of operation was altered and system designers contemplated having more than one program resident in the memory. This clearly meant that when one program is awaiting

completion of an input or output, another program could, in fact, engage the CPU..

# *Late 60’s and early 70’s*

Storing multiple executables (at the same time) in the main memory is called multiprogramming. With multiple excutables residing in the main memory, the immediate consideration is: we now need a policy to allocate memory and processor time to the resident programs. It is obvious that by utilizing the processor for another process when a process is engaged in input or output the processor utilization and, therefore, its output are higher. Overall, the multiprogramming leads to higher throughput for this reason.



# Multiprogramming

While multiprogramming did lead to enhanced throughput of a system, the systems still essentially operated in batch processing mode.

## 1980’s

In late 70s and early part of the decade of 80s the system designers offered some interactivity with each user having a capability to access system. This is the period when the timeshared systems came on the scene.

Basically, the idea is to give every user an illusion that all the system resources were available to him as his program executed. To strengthen this illusion a clever way was devised by which each user was allocated a slice of time to engage the processor. During the allocated time slice a users’ program would be executed. Now imagine if the next turn for the same program comes quickly enough, the user would have an illusion that the system was continuously available to his task. This is what precisely time sharing systems attempted – giving each user a small time slice and returning back quickly enough so that he never feels lack of continuity. In fact, he carries an impression that the system is entirely available to him alone.

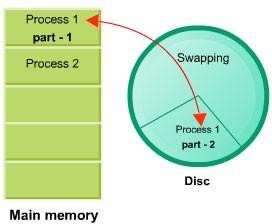
Timeshared systems clearly require several design considerations. These include the following: How many programs may reside in the main memory to allow, and also sustain timesharing? What should be the time slice allocated to process each program?

How would one protect a users’ program and data from being overwritten by another users’ program? Basically, the design trends that were clearly evident during the decade of 1970-80 were: Achieve as much overlapping as may be feasible between I/O and processing. Bulk storage on disks clearly witnessed a phenomenal growth. This also helped to implement the concept to offer an illusion of extended storage. The concept of “virtual storage” came into the vogue. The virtual storage essentially utilizes these disks to offer enhanced addressable space. The fact that only that part of a program that is currently active need be in the main memory also meant that multi-programming could support many more programs. In fact this could be further enhanced as follows:

1. Only required active parts of the programs could be swapped in from disks.
2. Suspended programs could be swapped out.

This means that a large number of users can access the system. This was to satisfy the notion that “computing” facility be brought to a user as opposed to the notion that the

“user go to compute”. The fact that a facility is brought to a user gives the notion of a utility or a service in its true sense. In fact, the PC truly reflects the notion of “computing utility” - it is regarded now as a personal productivity tool.



# Swapping of program parts main memory - disc, vice-versa

It was in early 1970s Bell Laboratory scientists came up with the now well known OS: Unix. Also, as the microcomputers came on scene in 1980s a forerunner to current DOS was a system called CP/M. The decade of 1980s saw many advances with the promise of networked systems. One notable project amongst these was the project Athena at MIT in USA. The project forms the basis to several modern developments. The client-server paradigm was indeed a major fall out. The users could have a common server to the so called X-terminals.

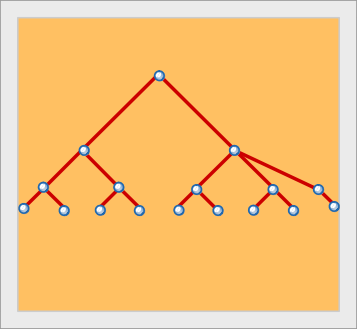
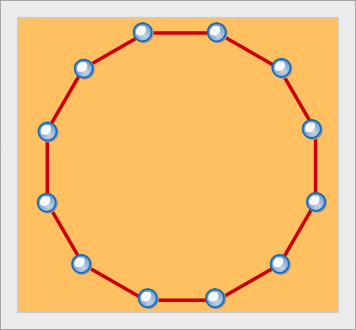
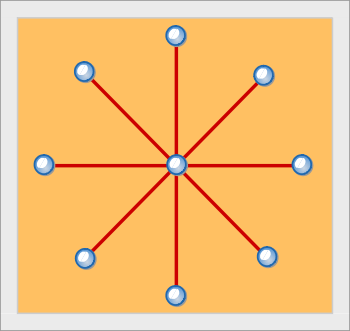
The X windows also provided many widgets to support convenient human computer interfaces. Using X windows it is possible to create multiple windows. In fact each of the windows offers a virtual terminal. In other words it is possible to think about each of these windows as a front-end terminal connection. So it is possible to launch different applications from each of the windows. This is what you experience on modern day PC which also supports such an operating environment.

In our discussions, we shall discuss many of the above issues in greater detail as we move to later chapters. On the micro-computer front the development was aimed at relieving the processor from handling input output responsibilities. The I/O processing was primarily handled by two mechanisms: one was BIOS and the other was the graphics cards to drive the display. The processor now was relieved from regulating the I/O. This made it possible to utilize the processor more effectively for other processing tasks. With the advent as shown in the figure, were

being experimented with protocols for communication amongst computers evolved. In particular, the TCP/IP suite of network protocols were implemented. The growth in the networking area also resulted in giving users a capability to establish communication

between computers. It was now possible to connect to a remote computer using a telnet

protocol. It was also possible to get a file stored in a remote location using a file transfer (FTP) protocol. All such services are broadly called network services.



**x**

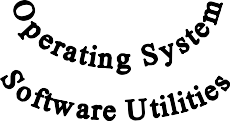
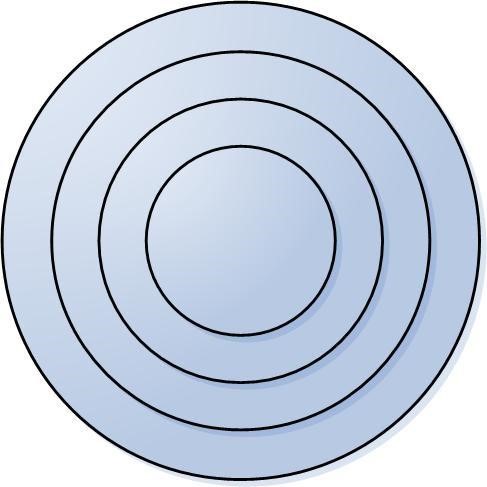
**x**

**x**

(a)Star (b) Ring (c) Tree

Let’s now briefly explore where the OS appears in the context of the software and application.

* 1. Operating System Architecture

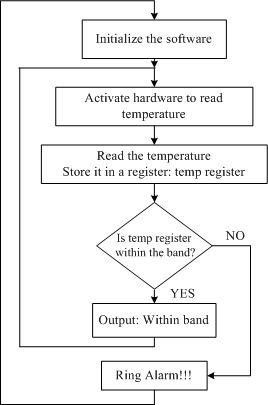


Let’s consider a scenario where we need to embed the computer system in an industrial application. This may be regulating the temperature of a vessel in a process control. In a typical process control scenario

* Monitoring – initializes and activates the hardware.
* Input – Reads the values from sensors and stores it in register.
* Decision – checks whether the readings are within the range.
* Output – responds to the situation.
* Scenario: A temperature monitoring chemical process.

What we need: A supervisory program to raise an alarm when temperature goes beyond a certain band.

* The desired sequence of operational events: Measure input temperature, process the most recent measurement, perform an output task.



The computer system may be employed in a variety of operational scenarios like a bank, airlines reservation system, university admissions and several others. In each of these we need to provide the resources for

* Processing
* User access to the system
* Storage and management of information
* Protection of information against accidental and intentional misuse
* Support for data processing activities
* Communication with I/O devices
* Management of all activities in a transparent manner.

Let’s now review What Does an OS Do?

* Power On Self Test (POST)
* Resource management

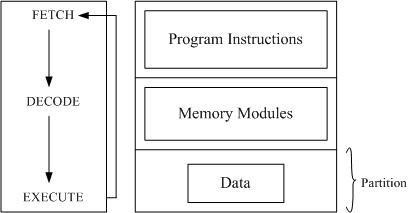
Support for multi-user

* Error Handling
* Communication support over Network
* (Optional) Deadline support so that safety critical application run and fail gracefully

**Operational View:**

Let’s briefly look at the underlying principle of operation of a computer system. Current systems are based on The Von-Neumann principle. The principle states that a program is initially stored in memory and executed by fetching an instruction at a time. The basic cycle of operation is

* Fetch an instruction (Fetch)
* Interpret the instruction (Decode)
* Execute the instruction (Execute)



Operating Cycle Memory Map

Modern systems allow multiple users to use a computer system. Even on a stand alone PC there may be multiple application which are running simultaneously. For instance, we have a mail program receiving mails, a clock to display time while we may be engaged in browsing a word process.

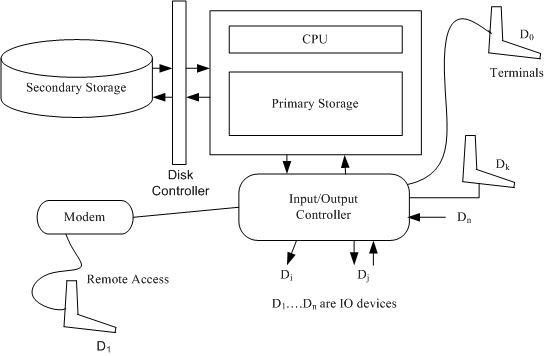
In other words OS need to schedule the processor amongst all the application simultaneously without giving an impression that the processor time is being divided and scheduled per an application.

An Operational Overview:

* Processor – schedule and allocate processor time.

Memory – executes program and access data

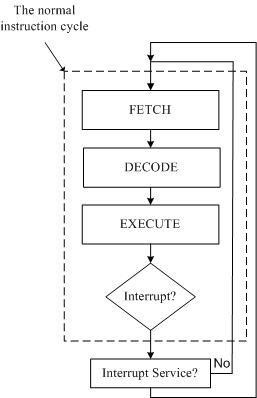
* Input output devices
* Communication with devices
* Mutual exclusion – schedule the usage of shared device and fair access
* Shell of an OS
* Human computer interface (HCI/CHI)



# A Modern Computer System

The peripheral devices communicate in a mode known as interrupt mode .Typically human input is considered important and often uses this mode. This is so because human desire to guide the operation. For instance, we use a mouse click or a key board input.

These require immediate attention by needing an appropriate interruption service.



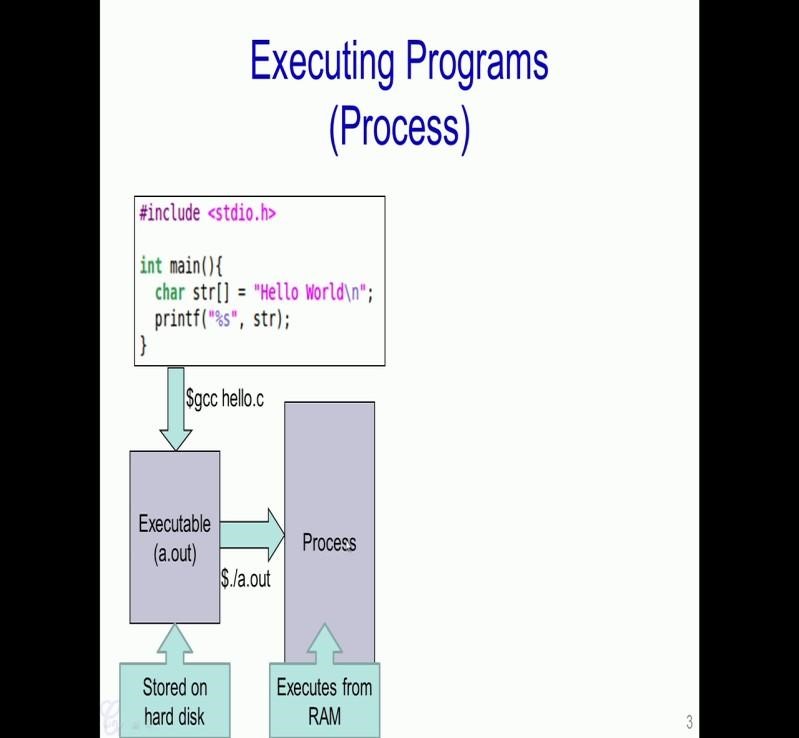
**Processes and Tools:**

**Processes:**

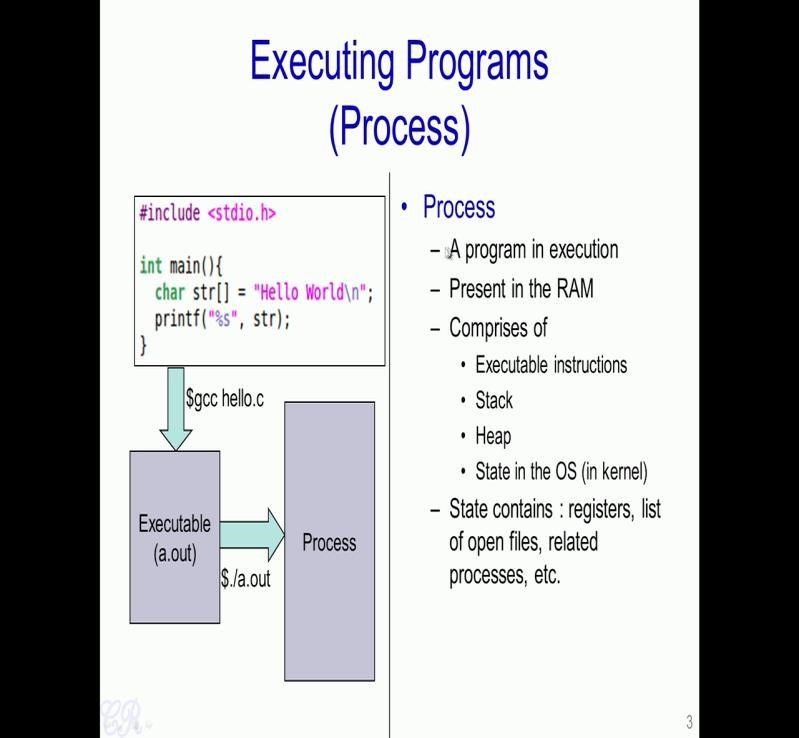
Most OSs support a notion that a program in execution may be regarded as a process. Clearly, with multiple applications active at the same time there are many processes that are active. The OS needs to manage all these processes. Often applications may spawn processes that need to communicate with each other. Such inter process communication forms the primary basis of distributed computing.

With the coming together of communications network and computers it is possible to create processes on one machine and seek services from another machine. The machine seeking the services is called ***client machine*** and the machine offering services is called ***server machine***. When you use a web browser, your computer is the client and the machine which provides the content for browsing is the web server. All modern OSs support client-server operations for service provisioning.

System Calls, Files and essentially the structure of the Operating System.



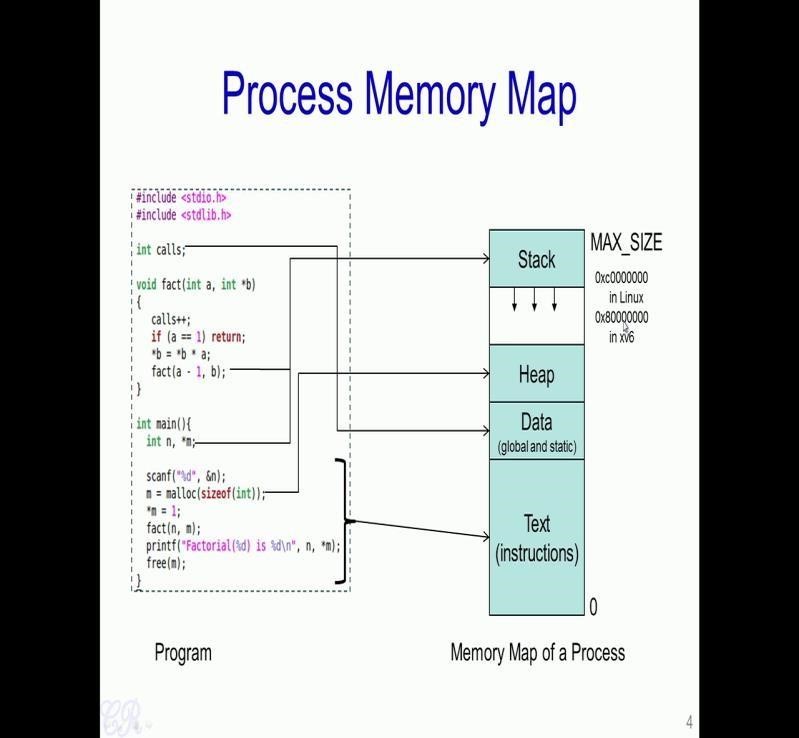
Consider this particular program written in C. So, this program prints "Hello World" on to the screen. In order to compile and run this program, we first need to use a compiler such as gcc and specify the c code name such as hello.c in this case and what we will get is an executable, in this case it is called about. This executable is stored on the hard disk. In order to run this particular program we specify a command like ./about and it results in a process being created in the RAM. So, this process is essentially the – about program under execution, which is present in the RAM.



To define it formally, a process is a program under execution which is executed from

RAM and essentially comprises of various sections, such as the Executable instructions, Stack, Heap and also a hidden section known as the State. So, this state is actually maintained by the operating system and contains various things like the registers, list of open files, process, list of related processes etc..

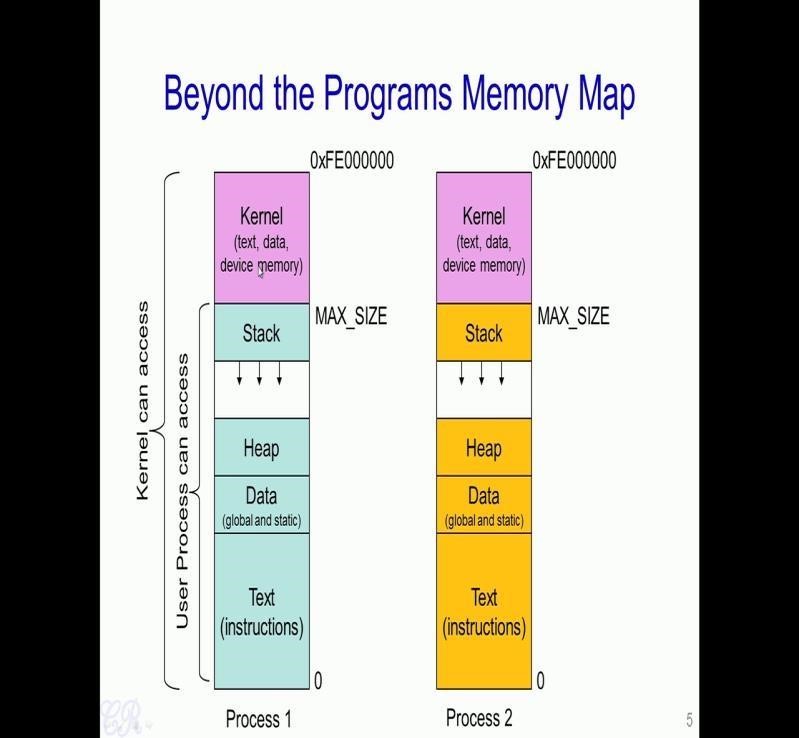
we will look more into detail about what this particular process contains and how it is managed.



So, Let us take a very simple example. So, this is a program and this is a process that is created when this program is executed. Now the process has various sections for example, it has the Text, Data, Heap and the Stack. Now various parts of this program when executed get mapped into the various sections of the process. For instance, all the instruction such as the instructions involved in the function main will get mapped into the text section of the process. Similarly, other functions such as the fact() function the instructions involved in this will also get mapped into the text section.

Now the global data and also static data gets mapped into the data section of the process. So, this section is actually divided into two parts where called as initialized and non- initialized sections. Third section is the heap, now any dynamically allocated memory such as m which is dynamically allocated using malloc gets created in the heap. Now the final section is called the stack, which contains all the local variables such as n and m and also information about function invocation. For example, in this case we have a recursive function which is getting invoked. So, all this information is present in the stack.

Now, the memory map of a process comprising all of the sections has a maximum limit called the MAX SIZE. So, typically at least in processes which are used in typical operating systems these days, this MAX SIZE is going to be fixed by the OS. For instance in a 32 bit Linux operating system, the MAX SIZE of every process is fixed at 0xc0000000. In the xv6 operating system which we are looking at for this course, the MAX SIZE of a process is fixed at the address 0x80000000.



So, what we had seen is that if every process, a program that is a program under execution gets mapped into an area which starts at 0 and ends at MAX SIZE. So, what is present beyond this MAX SIZE of the process? So, typically the Kernel or the operating system gets mapped to the memory region from the MAX SIZE to the maximum limit. The entire thing like Text i.e the instructions of the operating system, operating system data, the OS heap and also device memory gets mapped into this upper region of the OS.

So, typically any user program could access any part of this lower region i.e the green region. So, there would not be any problem to actually read data from any of these user space regions or even write data to parts of these user space regions. But, however, the process cannot access any data present in the Kernel memory that is beyond the MAX SIZE limit. However, the Kernel or the operating system which is executing from this upper region can access data from any part of the region that is it could execute or access data from in this kernel region as well as in the user space region.

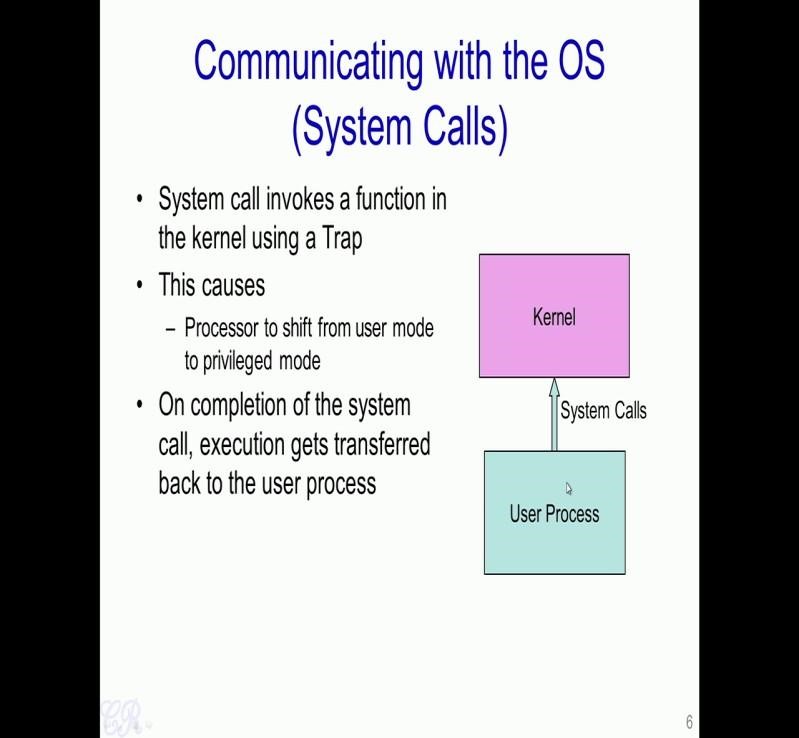
Now what happens when we actually have multiple processes running in the same system? So, each process would have its own memory map, we having its own instructions, data, heap and stack, and also the kernel component is also present beyond the MAX SIZE. So, what you see is that every process in the system would have the kernel starting at MAX SIZE and extending beyond. Only the lower parts and this kernel part is going to be same for every process that is executing in the system. Below this MAX SIZE is going to vary from one process to another.

So, what does this mean? So, what it means is that when you execute one process and then executing another process, the regions above the MAX SIZE is going to be similar, while the regions below the MAX SIZE is going to change from one process to another.

So, we mention that user programs will not be able to access any data in the kernel space.

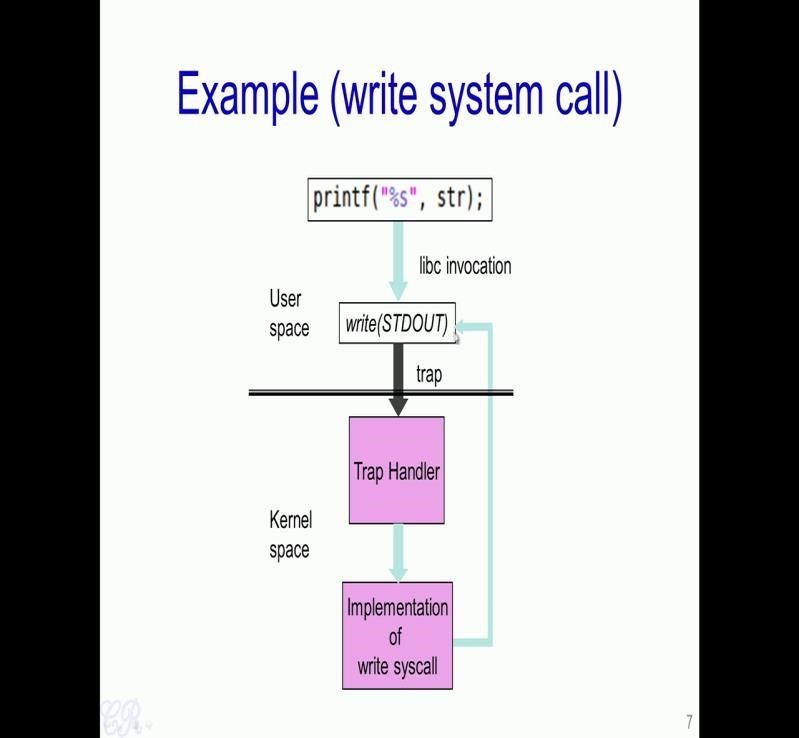
So, in that case how does the user program actually invoke the operating system?

* 1. System Calls



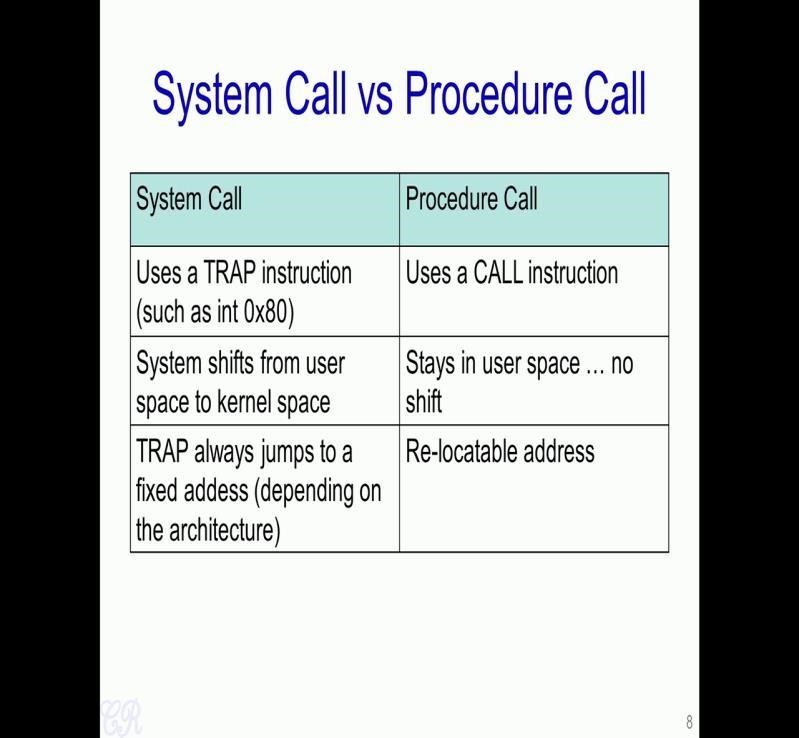
So, there are special invocation functions which the operating system support these are known as System Calls. So a System Calls are a set of functions which the OS support and a User Process could invoke any of the system calls to get information or to access hardware for other resources within the Kernel.

So, what happens when a system call is invoked; is that a process which is generally running in a user mode gets shifted to something known as a kernel mode or a privilege mode which will allow the kernel or the operating system to actually execute. When the system call completes execution then the user process will resume its execution from where it actually stopped.



So Let us take an example of the printf statement. So, printf in fact is a library call. So, it is present in this libc and it results in particular function in the User space known as write() to be invoked and printf() function will then invoke a system call called write, with the parameter call STDOUT. So, STDOUT is a special parameter which essentially tells the operating system that this string provided by printf should be displayed on to the standard output that is the screen.

So, the write is the system call which causes a trap to be triggered, and this trap will result in something known as a Trap Handler in the Kernel space to be executed and the Trap Handler would then invoke a function which will correspond to the write system call. So, this write system call will then be responsible for actually printing the string provided by star on to the screen. After the write system call completes, then the execution is transferred back to the user space and the process continues to execute.

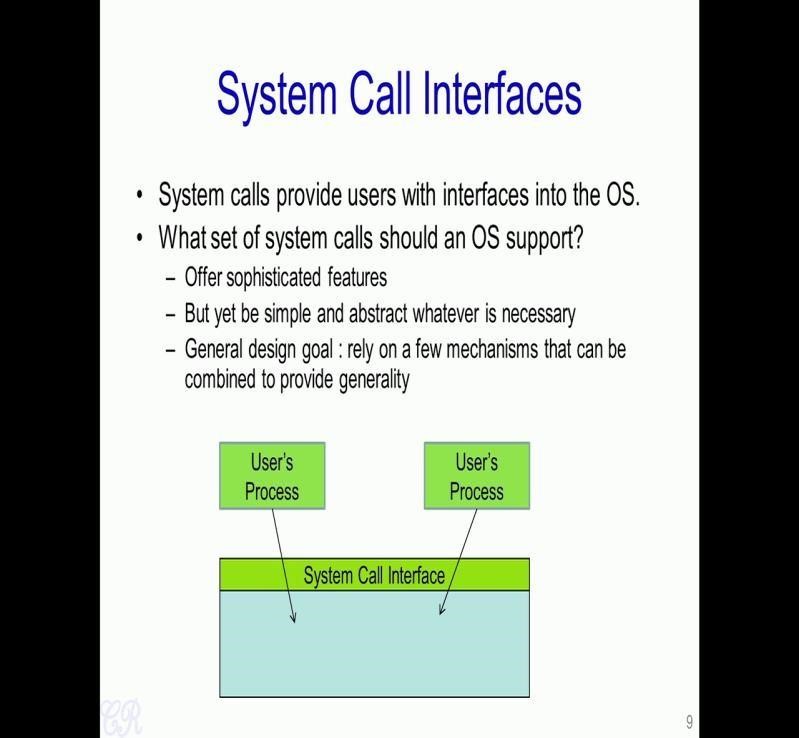


What is the difference between a System Call verses a Standard Function Call or

Procedure Call. So, one important difference is that, when we want to invoke a function in a program or in a process we use an instruction such as a CALL instruction this is a standard x68 assembly instruction, and this will result in the function getting called and after that function gets invoked it returns back to the calling function.

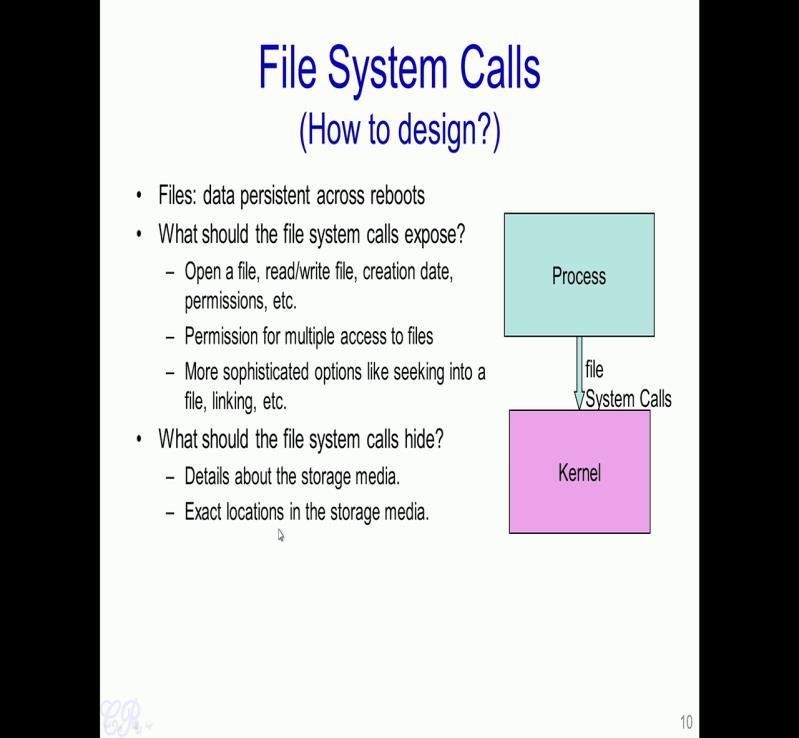
In order to invoke a system call however, we use a TRAP instruction such as the int 0x80. So, int here stands for interrupt or software interrupt and it results in the system shifting from the user space or the user space mode of operation to the kernel space. So, the trap instruction causes the kernel to be invoked and it causes instructions in the kernel to then be executed. However, when we use the standard function call or the procedure call using the CALL instruction there is no change or shift from user space to kernel space and so on. So, the execution continues to remain in the user space as it was before.

Another very settle difference between the system call and a standard procedure call or a function call is that the destination address or the destination function which is invoked can be at a relocatable address. So, it could change every time the program is compiled and so on. However with the system call when a trap instruction is used the hardware actually or the processor actually decides where the next instruction in the kernel space should get invoked. So, this is going to be fixed irrespective of what program is running, what operating system is running, and so on.



So, one crucial aspect when actually designing operating systems is what system call should the operating system supports? We had seen that the only way a user process could invoke a particular functionality in the operating system is through the system call interface. So, the question now comes that if a person is actually designing an operating system, so what are the interface that the system call should support.

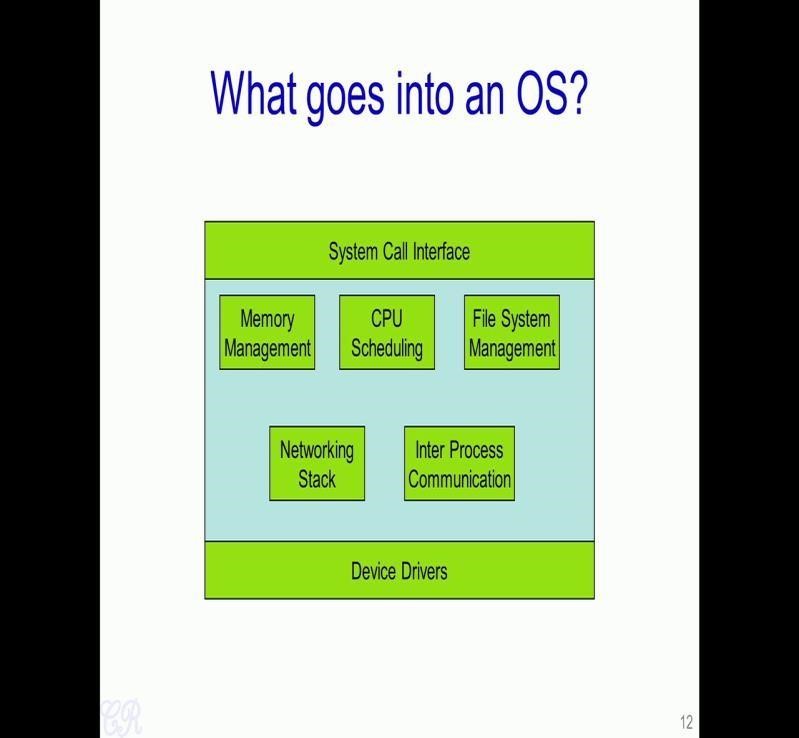
So, one obvious requirement is that the system call interface should have several sophisticated features so that a user process could actually very easily be able to interface or invoke several important functionality in the operating system. However a different approach is to have a very simple system call interface and abstract whatever is necessary from the operating system.



Let us take an example of system calls that an OS supports for accessing files. So, as we know files are a data which is persistent across reboots, so these are data which is stored in the hard disk and could be read, written or accessed using function such as fopen, close, read, write and so on. Every time we do a file open, it would require that the hard disk be accessed, so a process would need to invoke a system call into the kernel and the kernel should actually take care of accessing the hard disk or a hard disk buffers or any other storage medium to open the file and return back a pointer to the process.

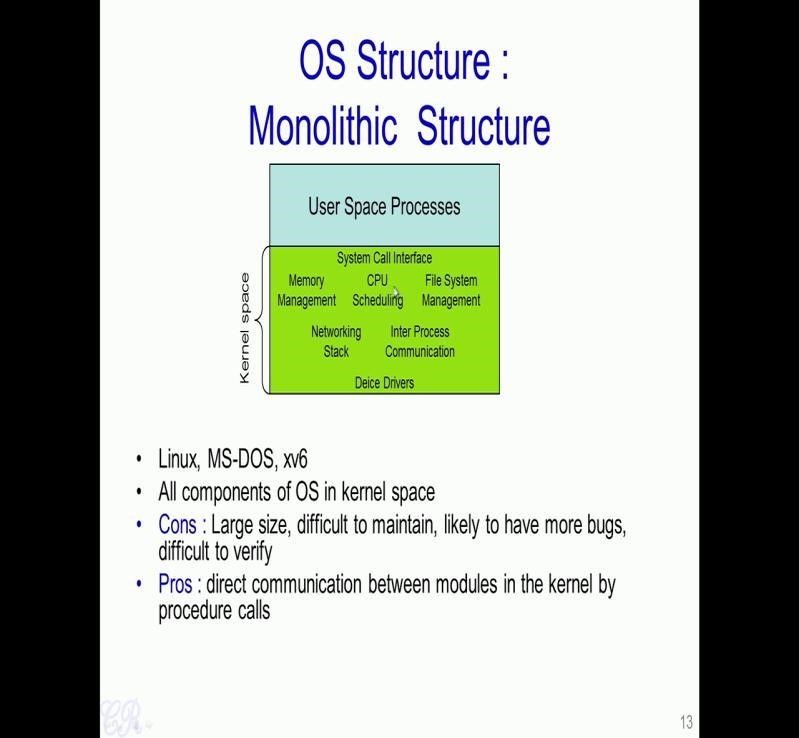
Now the question comes is how does a operating system designer decide what system call should be provided or supported in order to access files? So, some of the obvious things are like there should be system calls to open a file, read or write to a file, there should be system calls to actually modify the creation date, set permissions and so on. The operating system could also support more complicated or more sophisticated operations such as being able to seek into a particular offset within a file, be able to link between files and so on. So, these are the essential requirements that a system call for handling files should support.

On the other hand, operating system should be able to hide some details about the file. For instance, details which should not be supported by system calls is like things such as like details about the storage media where the file is stored, for instance whether the file is stored on a USB drive or a Hard disk or a CD-ROM. This is actually abstracted out by the operating system, and the user process would not be easily able to know where the file is stored. Another aspect which is abstracted out by the operating system is the exact locations in the storage media.



We will now look at how a typical Operating System structure looks like. So, the Operating System, suppose we consider this as a big green block have a several modules built internally. For example, it would have a memory management module which manages all the memory in the system, it would have a CPU scheduling block that is also the file system management module which will control how the file system such as the those present in hard disk or a CD-ROMs are managed. So, you have a networking stack which manages the TCP/IP network and you have something known as the inter process communication module which would take care of processes communicating with each other.

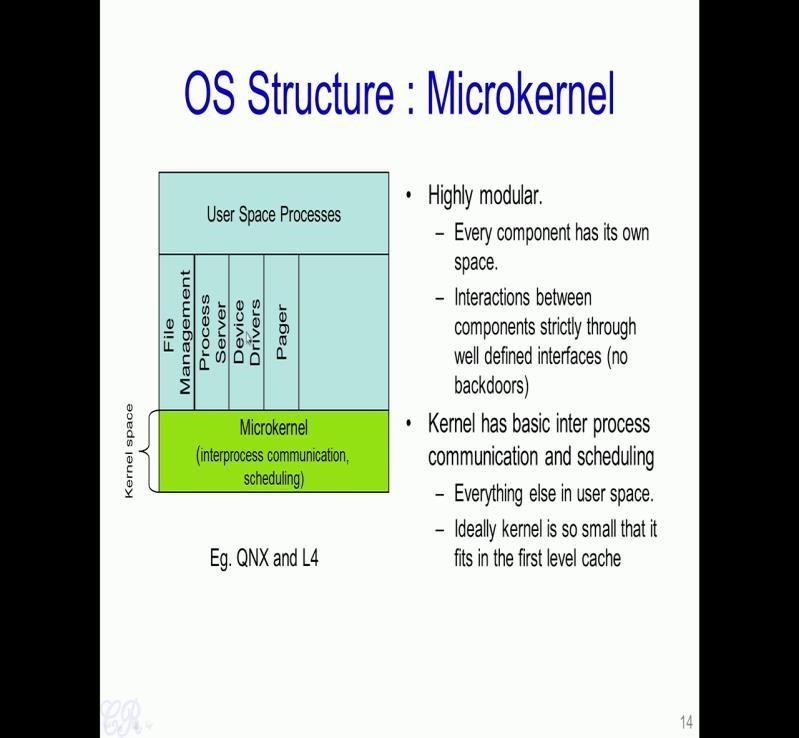
So, two important things which have not been mentioned as yet is the System Call Interface, which allows user processes to actually access features or functionalities within each of these modules. Another aspect is the Device Drivers which would take care of communicating with the hardware devices and other hardware resources within the system. So, this essentially is all the different modules that an operating system supports.



So, In a Monolithic Structure of an operating system, all these various modules in the OS are present in a single addressable kernel space, so what this means is that this is just one large chunk of code and all of them are you could think of as one large program where all these modules are present in. So Therefore, calling any function from the memory management to say the networking stack would just mean a simple function call.

Similarly from the networking stack to the device driver would be another function call.

This is essentially the advantage of having such a monolithic structure, is that you could there is a direct communication between one module and another. On the other hand the Kernel space becomes very large, and therefore, difficult to maintain and is likely to have more bugs. So, typical operating system such as Linux and xv6 and MS-DOS uses a monolithic structure. So, to take an example the Linux operating system or the Linux Kernel has around 10 million lines of code. So, all these 10 million lines of code comprises of the entire kernel Linux Kernel which is actually present in this area.



Another common structure of the operating system is known as the Microkernel structure, where the kernel is actually highly modular and every component in the kernel has its own addressable space. So, it is like having each of these as independent processes and you have a very small microkernel which actually runs in the kernel space, which is in charge of managing communication between each of these processes, and also communication between user process and the operating system processes.

So, the advantage here is that this microkernel is extremely small. So, ideally it is small enough to actually fit into the L1 cache of the system itself, so typically it would be quite fast. However, the drawback is that you now cannot have direct calls from say the file management to a device driver or rather like unlike the monolithic kernel where you could make direct function invocation from a file management module to a device driver function. Here every invocation of that form should be through a communication channel known as an IPC or Inter Process Communication channel.

## Virtual Machine

A Virtual Machine (VM) is an emulated environment of a physical computer system that runs an operating system and applications. A VM can be implement through software, firmware, hardware or a combination. An organization can have multiple VMs running different operating systems (OS) all stored on one host machine that is monitored by a hypervisor. There are two types of hypervisors, Type 1 and Type 2. We will learn about them further in the lesson.

There are different types of virtual machines and each offer different functions.

### System Virtual Machines

A system virtual machine is an environment that allows multiple instances of the operating system(VMs) to run on a host system, sharing the physical resources.

### Process Virtual Machine

A process virtual machine, also known as an application VM, is used to execute computer programs in a platform-independent environment. It is designed to run applications in the same way irrespective of the platform.

A VM is very useful in organizations for many reasons, from application development and testing in different environments to data backup and storage, etc. as it proves to be cost effective. One example would be a programmer is developing an application that needs to be launched to the entire workforce for use. The application will pull data from other sources and the users will be able to run reports for specific information. The programmer has to ensure the application will work on the organizations computers without any bugs. Before the programmer launches the application to the workforce, they have a VM loaded onto the server that has a copy of the OS that the organization uses and their standard for all their computers. The programmer will launch their application in the VM OS to see how it works and will fix any bugs they run into before fully launching a bug free application to the workforce. They would do the same if they have to perform any updates to the system or the application, they would test the application in the VM environment before releasing it to the workforce.

**Unit 2. Process Management**

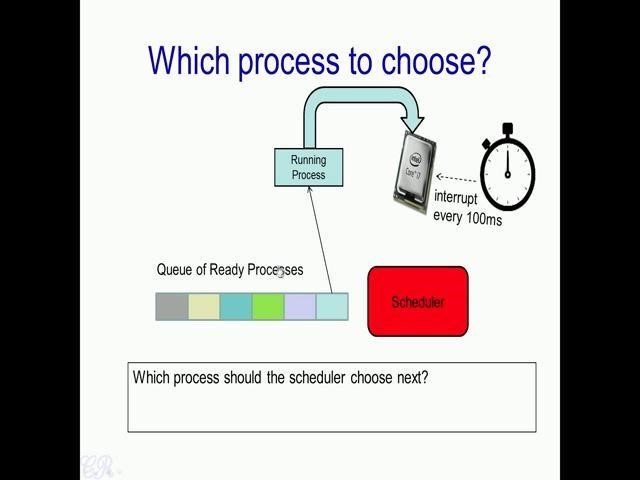
2.1. Introduction. A process is a program in execution. Process is not as same as program code but a lot more than it. A process is an 'active' entity as opposed to program which is considered to be a 'passive' entity. Attributes held by process include hardware state, memory, CPU etc.A process is the [instance](https://en.wikipedia.org/wiki/Instance_(computer_science)) of a [computer program](https://en.wikipedia.org/wiki/Computer_program) that is being executed by one or many threads. It contains the program code and its activity. Depending on the [operating system](https://en.wikipedia.org/wiki/Operating_system) (OS), a process may be made up of multiple [threads of execution](https://en.wikipedia.org/wiki/Thread_(computing)) that execute instructions [concurrently](https://en.wikipedia.org/wiki/Concurrency_(computer_science)).

While a computer program is a passive collection of [instructions](https://en.wikipedia.org/wiki/Instruction_set), a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often results in more than one process being executed.

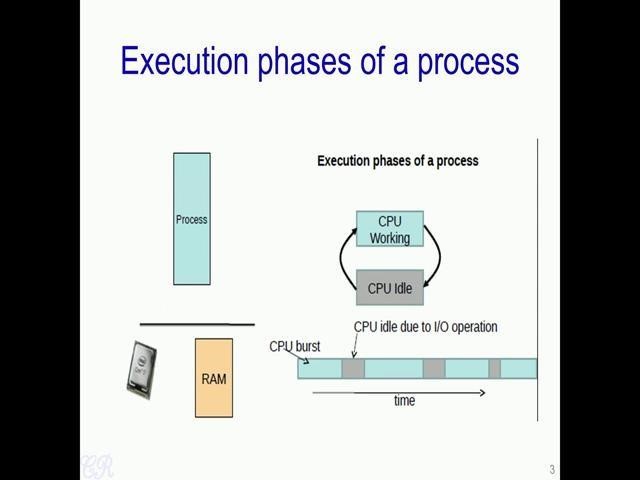
[Multitasking](https://en.wikipedia.org/wiki/Computer_multitasking) is a method to allow multiple processes to share [processors](https://en.wikipedia.org/wiki/Central_processing_unit) (CPUs) and other system resources. Each CPU (core) executes a single [task](https://en.wikipedia.org/wiki/Task_(computing)) at a time. However, multitasking allows each processor to [switch](https://en.wikipedia.org/wiki/Context_switch) between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform [input/output](https://en.wikipedia.org/wiki/Input/output) operations, when a task indicates that it can be switched, or on hardware [interrupts](https://en.wikipedia.org/wiki/Interrupt).

A common form of multitasking is [time-sharing](https://en.wikipedia.org/wiki/Time-sharing). Time-sharing is a method to allow high responsiveness for interactive user applications. In time-sharing systems, [context switches](https://en.wikipedia.org/wiki/Context_switch) are performed rapidly, which makes it seem like multiple processes are being executed simultaneously on the same processor. This seeming execution of multiple processes simultaneously is called [concurrency](https://en.wikipedia.org/wiki/Concurrency_(computer_science)).

For security and reliability, most modern [operating systems](https://en.wikipedia.org/wiki/Operating_system) prevent direct [communication](https://en.wikipedia.org/wiki/Inter-process_communication) between independent processes, providing strictly mediated and controlled inter-process communication functionality.

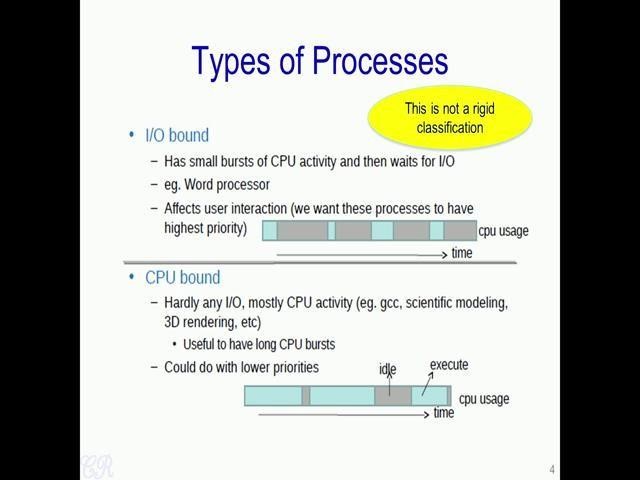


We had seen in operating systems that a scheduler present would choose a particular process from the ready queue and that process is assigned to run in the processor. Now the question that we were going to analyze now is how should this scheduler (mentioned in above image) choose the next process? Essentially, how should the scheduler choose a process to run on the CPU from the existing queue of ready processes?



Now to analyze this, we first look at execution phases of a process. Now any program has been known to have two phases of execution; one is when it is actually executing instructions which is known as a CPU burst (mentioned in above image as blue boxes), while the other is when it is blocked on an I/O or not doing any operations, so in this particular case, the CPU is idle (mentioned in above image as grey boxes).

Thus, as we look with over time, a particular process would have some amount of CPU burst in which it would execute instructions on the processor, then it would have an some idle time in which it is waiting for a I/O operation. Then there would be a burst of CPU time and so on. Thus, there is always this inter-leaving between CPU burst and idle time that is a waiting for an operation.



So, based on this cases of execution (CPU working and CPU idle case) one could classify processes into two types. One is the I/O bound processes, while the other is the CPU bound process. Why do you make this distinction between I/O bound and CPU bound processes? Essentially, this is from a scheduling perspective, we would like to give I/O bound processes a higher priority with which they are allocated the CPU. Essentially, we want that I/O bound processes wait lesser time for the CPU compared to CPU bound processes. So, why is this required, we can take with an example.

So suppose we are using a word processor such as a notepad or Vim and we are giving this I/O bound process a very low priority. Now, suppose a user presses a key, because it has a low priority it does not get the CPU very often and therefore, it would take some time before that key pressed by the user appears onto the screen. So, this may be quite uncomfortable for the user, therefore we would like to give the I/O bound process such as the word processor higher priority with which it will get the CPU, so that the user interaction with the CPU becomes more comfortable.

On the other hand, if you look at CPU bound processes, we could give it a lower priority. Now for instance, if you take one of these applications, the CPU bound applications like for instance say let us say ‘gcc’ that is compiling a program and let us say you are compiling a large program which takes 5 minutes. Now it will not effect this user much if the time taken to compile that particular program increases from 5 minutes to say 5.5 minutes. Thus, a CPU bound processes could work with a lower priority. This classification between I/O bound and CPU bound is not a rigid classification, that is a process could be an I/O bound process at one time, and after some time it could behave like a CPU bound process.

So, to take an example of a process which behaves both like an I/O bound as well as CPU bound, you could take for instance Microsoft excel. When we are actually entering data into the various cells in excel, it acts as I/O bound process. So, it behaves like an I/O bound process with small CPU burst and large times of I/O cycles. While on the other hand when you are actually computing some statistic on the data entered, Excel will behave like a CPU bound process, where there is a large portion of CPU activity or the time taken to actually operate on that particular data.

# 2.2. States of a Process

* New (Create) – In this step, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.
* Ready – New -> Ready to run. After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue for ready processes.
* Run – The process is chosen by CPU for execution and the instructions within the process are executed by any one of the available CPU cores.
* Blocked or wait – Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or wait state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.
* Terminated or completed – Process is killed as well as PCB is deleted.
* Suspend ready – Process that was initially in the ready state but were swapped out of main memory(refer Virtual Memory topic) and placed onto external storage by scheduler are said to be in suspend ready state. The process will transition back to ready state whenever the process is again brought onto the main memory.
* Suspend wait or suspend blocked – Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory.  
  When work is finished it may go to suspend ready.

2.3. Types of schedulers:

1. Long term – performance – Makes a decision about how many processes should be made to stay in the ready state, this decides the degree of multiprogramming. Once a decision is taken it lasts for a long time hence called long term scheduler.
2. Short term – Context switching time – Short term scheduler will decide which process to be executed next and then it will call dispatcher. A dispatcher is a software that moves process from ready to run and vice versa. In other words, it is context switching.
3. Medium term – Swapping time – Suspension decision is taken by medium term scheduler. Medium term scheduler is used for swapping that is moving the process from main memory to secondary and vice versa.

## 2.4. Operations on the Process

### 1. Creation

Once the process is created, it will be ready and come into the ready queue (main memory) and will be ready for the execution.

### 2. Scheduling

Out of the many processes present in the ready queue, the Operating system chooses one process and start executing it. Selecting the process which is to be executed next, is known as scheduling.

### 3. Execution

Once the process is scheduled for the execution, the processor starts executing it. Process may come to the blocked or wait state during the execution then in that case the processor starts executing the other processes.

### 4. Deletion/killing

Once the purpose of the process gets over then the OS will kill the process. The Context of the process (PCB) will be deleted and the process gets terminated by the Operating system.

Multiprogramming – We have many processes ready to run. There are two types of multiprogramming:

1. Pre-emption – Process is forcefully removed from CPU. Pre-emption is also called as time sharing or multitasking.
2. Non pre-emption – Processes are not removed until they complete the execution.

**2.5. CPU Scheduling**

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the short-term scheduler (or CPU scheduler). The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

## 2.6. Scheduling Criteria

There are many different criterias to check when considering the "best" scheduling algorithm, they are:

#### CPU Utilization

To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

#### Throughput

It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

#### Turnaround Time

It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

#### Waiting Time

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

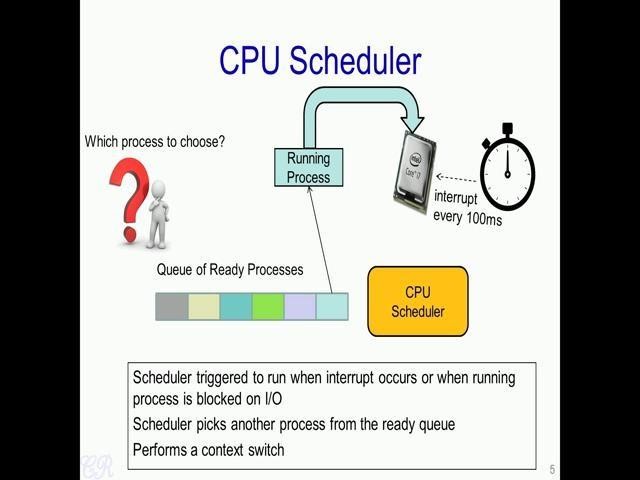
#### Load Average

It is the average number of processes residing in the ready queue waiting for their turn to get into the CPU.

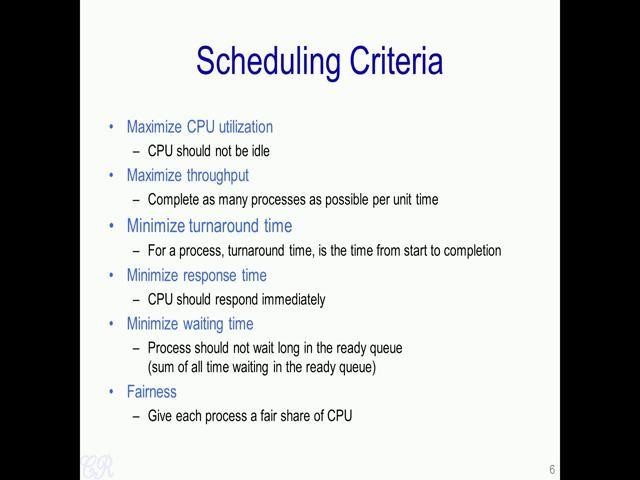
#### Response Time

Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution(final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.



Now, let us come back to the question about how the CPU scheduler should choose from the queue of ready processes, the next process to execute in the CPU. There could be several ways in which the scheduler could makes this choice; essentially, there could be several CPU scheduling algorithms which would look into the queue and make a particular decision.



So in order to compare these various scheduling algorithms, operating systems text books or operating systems research defines several scheduling criteria. So, these criteria could be used to actually to compare various scheduling algorithms to see the advantages and disadvantages of each of them. So let us go through each of these scheduling criteria one by one. The first scheduling criteria is the CPU utilization. The scheduling algorithm should be designed in such a way so as to maximize CPU utilization. In other words, the CPU should be idle as minimum time as possible.

The next criteria, we will look at is the throughput. Essentially, scheduling algorithms would try to complete as many processes as possible per unit time. A third criteria is the turnaround time and this criteria is looked at from a single process perspective. So, turnaround time is defined as the time taken for a single process from start to completion.

The fourth criteria is response time. So, this is defined as the time taken from the point that when the process enters into the ready queue to the point when the process goes into the running state, that is the time taken from the instant the process enters the ready queue to the time the CPU begins to execute instructions corresponding to that process. Another criteria, is the waiting time. Now this criteria, is based on the time taken by a process in the ready queue. Now as we know processes ready to run are present in the ready queue and it is required that they do not wait too long in the ready queue. So, scheduling algorithms could be designed in such a way that the waiting time or the average waiting time in the ready queue is minimized.

The final criteria we will see now is fairness. The scheduler should ensure that each process is given a fair share of the CPU based on some particular policy. So, it should not be the case that some process, for instance, takes say 90 percent of the CPU while all other processes just get round 10 percent of the CPU. So, all these criteria need to be considered while designing a scheduling algorithm for an operating system.

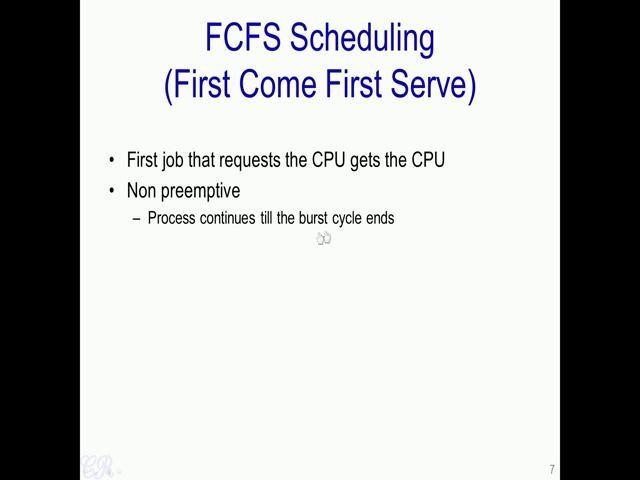
A single scheduling algorithm will not efficiently be able to cater to all these criteria simultaneously. So, therefore, scheduling algorithms are therefore designed for to meet a subset of these criteria. For instance, if you consider real time operating system, the scheduling algorithm for that system would for instance be designed to have minimum response time; other factors such as CPU utilization and throughput may be of secondary importance.

On the other hand, desktop operating system like a Linux will be designed for fairness, so that all applications running in the CPU or in the system are given a fair share of the CPU. Criteria such as response time may be less important from that perspective. So, we will now look at several scheduling algorithms starting from the simplest one that is the first come first serve scheduling algorithm and go to more complex scheduling algorithms as we proceed.

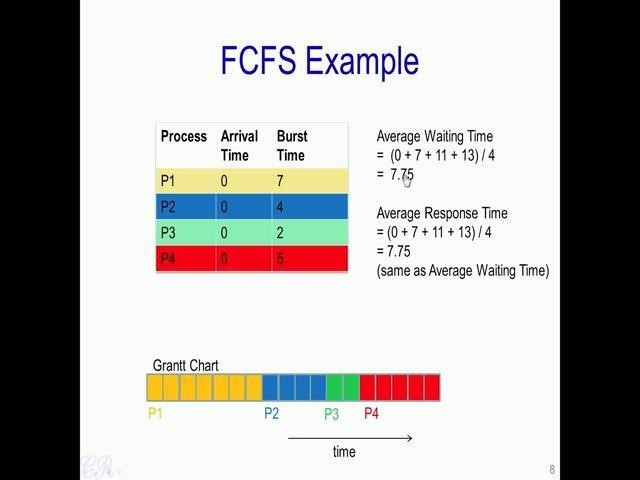
## 2.7. Scheduling Algorithms

To decide which process to execute first and which process to execute last to achieve maximum CPU utilisation, computer scientists have defined some algorithms, they are:

1. [First Come First Serve(FCFS) Scheduling](https://www.studytonight.com/operating-system/first-come-first-serve)
2. [Shortest-Job-First(SJF) Scheduling](https://www.studytonight.com/operating-system/shortest-job-first)
3. [Round Robin(RR) Scheduling](https://www.studytonight.com/operating-system/round-robin-scheduling)



So let us look at the First Come First Serve of the FCFS scheduling algorithm. The basic scheme in this case is that the first process that requests the CPU would be allocated the CPU or in other words, the first process which enters into the ready queue would be allocated the CPU. So, this is a non preemptive scheduling algorithm which means that the process once allocated the CPU will continue to execute in the CPU until its burst cycle completes.



Let us see this with an example. Let us say we have a system with 4 processes running (1st column); the processes are label P 1, P 2, P 3, P 4 and they have an arrival time, so the arrival time is present in the 2nd column (mentioned in above image). So the arrival time is defined as the time when these processes enter into the ready queue. So, for this particular very simple example, so we will consider that all processes enter the ready queue at the same time that is at the 0th time instant. The 3rd column is the CPU burst time, so it gives the amount of CPU burst for each process.

For instance, P 1 has a CPU burst of 7 cycles; P 2 has a burst of 4 cycles. Thus, this particular table (mentioned in above image) we have like 4 processes, which all enter simultaneously into the ready queue at the time instants 0 and they have different CPU burst time, for instance P 1 has 7 cycles, P 2 - 4 cycles, P 3 - 2 cycles and P 4 - 5 cycles.

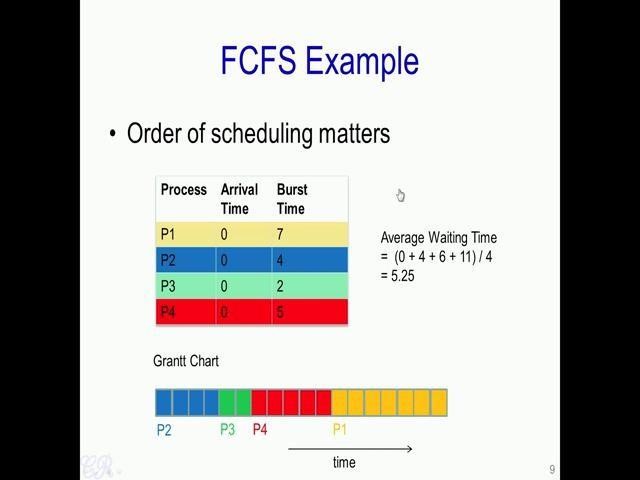
Now, we will see how these four processes get scheduled into the CPU or how these four processes get allocated the CPU. Since, all of these processes arrive at the same time, the scheduler does not actually have a choice to make. So, he would pick randomly a particular ordering. For instance, let us say the scheduler picks P 1 to run, so P 1 runs for 7 cycles and when it completes, the scheduler picks P 2 and P 2 runs for 4 cycles. After P 2 completes its burst, and P 3 executes in the CPU for 2 cycles; and then finally, P 4 is scheduled in to execute for 5 CPU cycles (mentioned in above image). So, this is represented by a Gantt chart.

So, a Gantt chart is a horizontal bar chart developed as a production tool in 1917 by Henry L Gantt who was an American engineer and a Social scientist. So, essentially in a Gantt chart we have like several blocks over here and each block represents a cycles of execution (mentioned in above image). So, for instance, P 1 executes for 7 cycles, so it has likes 7 blocks – 1, 2, 3, 4, 5, 6, 7 (yellow blocks), P 2 then executes for 4 cycles, so it’s given like 4 blocks (blue blocks). Then P 3 executes for 2 cycles, so it is given 2 blocks (green blocks) and finally, P 4 executes for 5 cycles, so it is given 5 blocks (red blocks).

So, we could compute the average waiting time for this particular case (mentioned in above FCFS example image). We see that process P 1 enters into the ready queue at the instant 0 and immediately gets to execute in the CPU. Thus, it does not have to wait at all (waiting time is 0). The second process P 2 arrives also at 0 that is also arrives at this point (instant 0) but gets to execute only after P 1 executes that is only after 7 cycles; thus, its wait time is for process P 2 is 7. Similarly, process P 3 which also enters in the 0th cycle gets to execute only after process P 1 and P 2 completes. In other words, it has to wait 11 cycles and the fourth process in a similar way needs to wait for 13 cycles. Therefore, the average waiting time in this particular case is 7.75 cycles ((0+7+11+13)/4

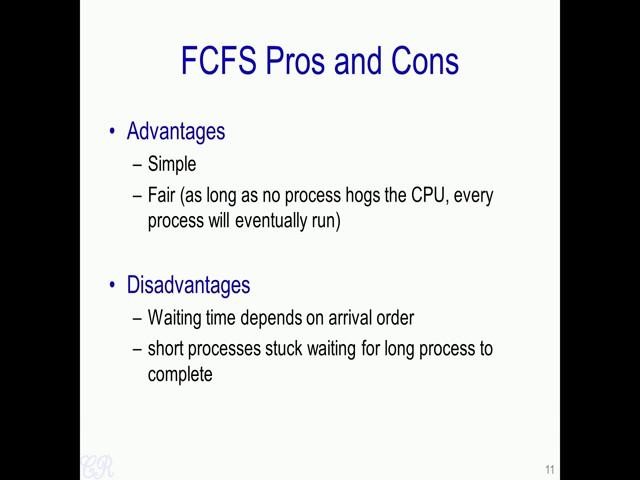
= 7.75).

Now, we can look at an other scheduling criteria, which is the average response time. So, average response time in this case is the same as the average waiting time. So the average response time is the time taken for a particular process to begin executing in the CPU minus the time it actually enters into the ready queue. So, for instance, P 2 enters into the ready queue at this instant, but begins to execute in the CPU only after 7 cycles. So, therefore, the response time for process P 2 is 7. Similarly, process P 3 has a response time of 11 because it has waited for 11 cycles to actually begin executing in the CPU (mentioned in above image). So, on average, the average response time is 7.75 just like the average waiting time.



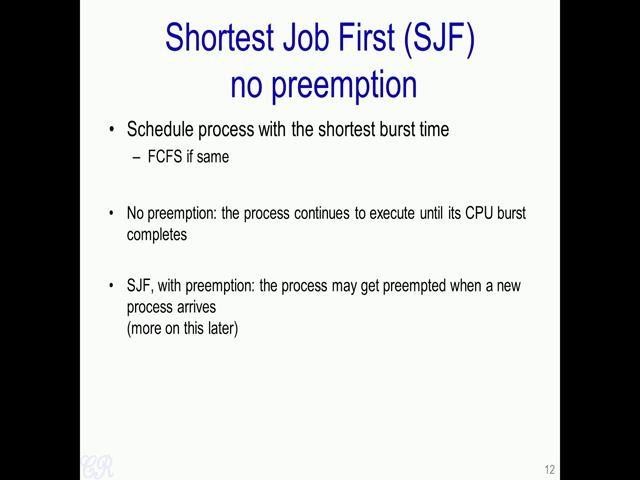
Now, one characteristic of the FCFS scheduling algorithm is that the order of scheduling matters. we assume that P 1 executes then P 2 executes then P 3 and then P 4, and we have got average waiting time and average response time of 7.75 cycles. Now, suppose we just change the ordering and let us say the ordering is now as follows that P 2 executes then P 3 executes then P 4 and finally P 1.

In such a case, if we compute the average waiting time we see that it gets reduce from 7.75 cycles to 5.25 cycles. Similarly, if you compute the average response time, you will see that the response time is also 5.25 cycles. In a similar way, you could compute the other criteria, and you will be able to actually see the difference with the two ordering schemes.



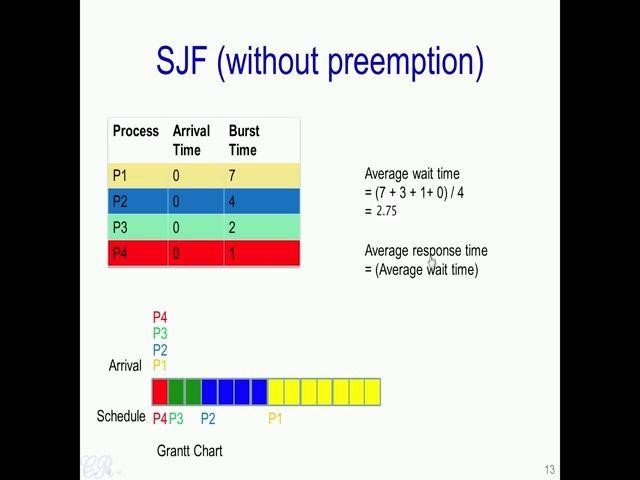
However, FCFS scheduling algorithm have several advantages. For the first thing is it is extremely simple. So the scheduling algorithm could complete very quickly and therefore, the time taken by the scheduling algorithm will be very less, and you would end up with very less context delays while changing the contexts. Another advantage of the FCFS scheduling algorithm is that it is fair. As long as no process hogs the CPU, every process will eventually run; or in other words, as long as every process terminates at some point, every other process in the ready queue will eventually get to execute in the CPU.

Now the drawback or the disadvantage of the FCFS schedulers as we have seen is that the waiting time depends on the arrival order. Another disadvantage is that short processes are stuck in the ready queue waiting for long processes to complete.



Now, let us look at another scheduling algorithm know as the Shortest Job First scheduling algorithm. In this particular scheduling algorithm, the job or the process with the shortest CPU burst time is scheduled before the others. Now if you have more than one process with the same CPU burst time then standard FCFS scheduling is used. There are two variants of the shortest job first scheduling algorithm.

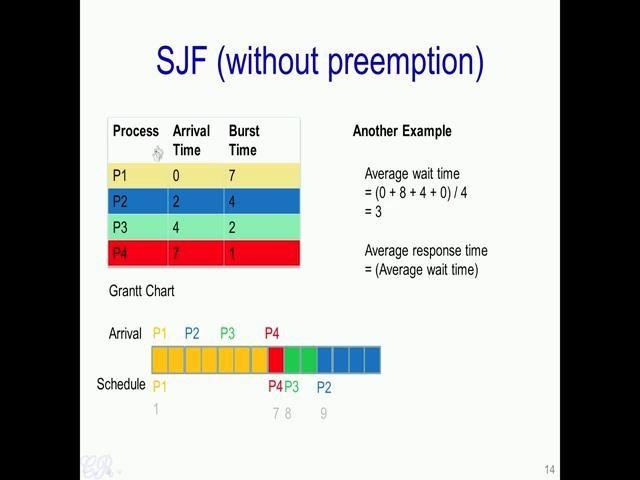
The first is the no pre-emption variant, while the second one is the shortest job first with pre-emption. Now in the SJF with no pre-emption, the process will continue to execute in the CPU until its CPU burst completes. In the second variant with pre-emption, it may be possible that the process which may get pre-empted, when a new process arrives into the ready queue. We will see more on this later, but first we will start with the shortest job first variant with no pre-emption.



So let us take the same example that we have seen in the previous case, we had their

Four processes P 1 to P 4 and all of them are arriving at the instant 0, that is at instant 0 these four processes P 1 to P 4 get into the ready queue. And each of these processes have a different CPU burst time that is 7, 4, 2 and 1 respectively (mentioned in above image). Now in the first instant, the CPU scheduler will look at the various burst times and find the one which is minimum.

So, in this case (mentioned in above image) we see that P 4 has the minimum CPU burst time (burst time 1), so that were scheduled first. So, first the process P 4 gets scheduled until it completes, in this particular case it completes in 1 cycle. Then among the remaining three, we see that P 3 has the lowest CPU burst time. So, process P 3 gets scheduled and executes till it completes its burst (burst time 2). Then P 2 gets scheduled because it has a burst time of 4, while P 1 has a higher burst time of 7. And finally, P 1 gets to execute till completion. Now, the average wait time, if we compute this is 2.75, while the average response time is also 2.75 as in the wait time case.



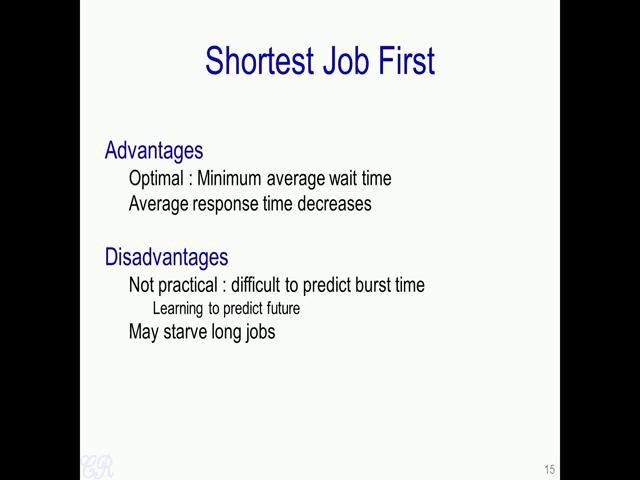
Now let us look at another example of shortest job first without pre-emption (mentioned in above image). So, we will take the same four processes P 1 to P 4 and each of these processes have the same burst time as before that is 7, 4, 2 and 1 respectively. However, they arrive at different instants that is the moment the instant in which they enter into the ready queue would be different. So, P 1 enters at the 0th instant, P 2 in the 2nd instant, P 3 at the 4th instant and P 4 in the 7th instant. Now, this is a slight modification in the Gantt chart, where in addition to showing which process is executing in the CPU, it also shows the order in which processes arrive. It shows that the P 1 arrives first, then P 2 in the 2nd instant, then P 3 and finally P 4 in the 7th instant.

Now, when the scheduler begins to execute at this particular instant (starting point in Gantt chart), the only process that has arrived at this particular point is P 1 therefore, it schedules P 1 to execute. So, P 1 executes for its entire burst that is of 7 cycles, and then at this particular cycle (after 7 blocks), the scheduler enters again or a scheduler executes again and this time it has got three processes to choose from all P 2, P 3 as well as P 4 have arrived in the ready queue.

And out of them P 4 has the shortest burst time therefore, it is chosen for execution.

Therefore, P 4 executes in the CPU, and then P 3 because P 3 has a shorter burst time

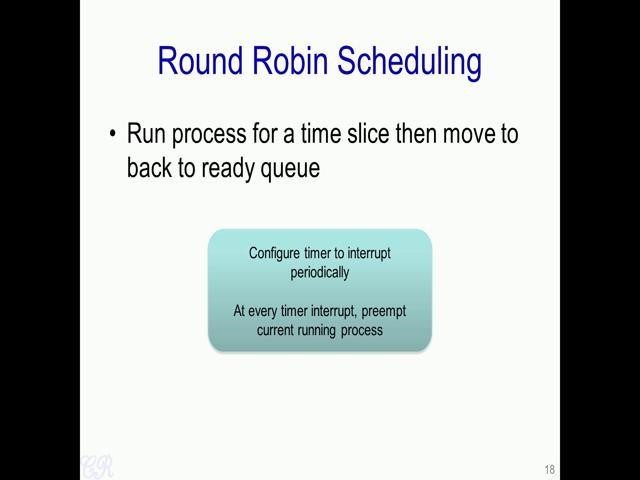
than P 2 and finally, P 2 gets executed. So, if we compute the average wait time, we see that it is 3 cycles (mentioned in above image).



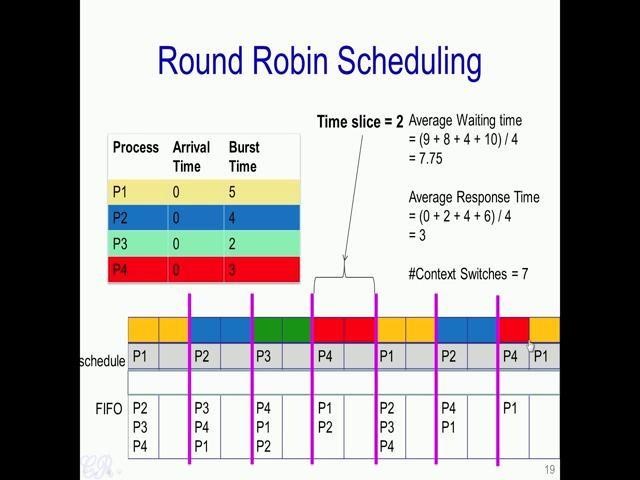
So the advantages of the shortest job first scheduling algorithm is that is Optimal. It will always give you the minimum average waiting time. And as a result of this, the average response time also decreases. The main disadvantages of the SJF scheduling algorithm is that it is not practical; essentially, it is very difficult to predict what the burst time would be. So, another drawback of the SJF scheduling algorithm is that some jobs may get starved; essentially if you have a process which has an extremely long CPU burst time, then it may never get a chance to execute in the CPU.



Now, we will look at the shortest job first scheduling algorithm (SJF) with pre-emption. So, this is also the shortest remaining time first. So the basic idea in this algorithm is that if a new process arrives into the ready queue, and this process has a shorter burst time than the remaining of the current process, then there is a context switch and the new process gets scheduled into the CPU. This further reduces the average waiting time as well as the average response time. However, as in the previous case that is the shortest job first with no pre-emption here also it is not practical.



Now we will look at another scheduling algorithm known as the Round Robin Scheduling algorithm. So, essentially with the round robin scheduling algorithm, a process runs for a time slice that is a process executes for a time slice and when the time slice completes it is moved on to the ready queue. So, in order to achieve this round robin scheduling algorithm which is also a pre-emptive scheduling algorithm; now in order to achieve the round robin scheduling, we need to configure the timer in the system to interrupt the CPU periodically. At every timer interrupt, the kernel would pre-empt the current process and choose another process to execute in the CPU.



Let us discuss the round robin scheduling algorithm with an example. So, one difference with respect to the other scheduling algorithms that we have seen so far is the notion of time slice. So, this is the Gantt chart (refer above image). So, especially see that periodically in this case, with a period equal to 2 that is we have keeping a time slice equal to 2, there is a timer interrupt that occurs and the timer interrupt would result in the scheduler being run and potentially another process being scheduled into the CPU. So, a data structure which is very useful in implementing the round robin scheduling algorithm is the FIFO.

This particular FIFO store the processes that need to be executed next into the CPU. For example, in this particular case P 2 is at the top of the FIFO, so it is a next process which gets executed in the FIFO. So, P 2 gets executed over here (next to P1 in Gantt chart). So, for example, we will still consider the 4 processes as we have done before that is P 1 to P 4 and we will assume that all of them arrive at the instant 0 and go into the FIFO of the ready queue and they have burst times of 5, 4, 2 and 3 respectively.

So let us say for discussion that the scheduler starts off with this particular order P 1, P 2, P 3 and P 4 and it first chooses to execute P 1. So, P 1 executes for 2 cycles and then there is a interrupt which occurs leading to a context switch and then the top of the FIFO in this particular case (mentioned in above image) P 2 get scheduled into the CPU, while P 1 gets pushed into the FIFO. So, P 2 then executes for 2 cycles until the next timer interrupt in which case the time slice of 2 cycles completes and then it gets pushed into the FIFO. So, P 2 now is at the bottom of the FIFO, while P 3 which is at the top of the FIFO get scheduled to run. So, in this way, every two cycles a new process may get scheduled into the CPU and execute.

So, if we compute the average waiting time. So, in this particular case, we will see that the average waiting time and the average response time is different. So, what is the average waiting time for P 1? So, the P 1 executes 2 cycles here (check first P1 block in Gantt chart), 2 cycles here (second P1 block) and completes execution over here (last P1 block). So, it waits in the ready queue in the remaining of the cycles. So the number of cycles it waits is 1, 2, 3, 4, 5, 6, 7, 8, 9 (P2, P3, P4 block then P2,P4 block). So, P 1 waits for 9 cycles; now P 2 waits for 8 cycles – 1, 2, 3, 4, 5, 6, 7, 8; P 3 for 4 cycles and P 4 for 10 cycles. So the average waiting time is 7.75 cycles.

Now, to compute the average response time as we have defined it before, the response time is the time the process enters into the ready queue to the time it begins to execute in the CPU, that latency would be the response time. So, P 1 for instance has a response time of 0 because it enters into the ready queue or enters into the FIFO and gets executed immediately. P 2 on the other hand enters in the 0th cycle, but gets to execute only after 2 cycles, so it has a response time of 2

Similarly, P 3 enters at 0 but executes only at this point. So, at rather at this instant (After P1 and P2 in Gantt chart) therefore, it has a response time of 4 and P 4 has a response time of 6 therefore, the average response time is 3. Now the number of context switches that occur is 7. So, in this case, 1, 2, 3, 4, 5, 6 and a context switch occurs over here (last block P4 and P1) because the process P 4 is existing out and P 1 gets continues to execute. So the numbers of context switches over here are 7 that is 1, 2, 3, 4, 5, 6 and the 7th context switch occurs over here (last block) then P 4 exits and P 1 gets switched into the CPU.

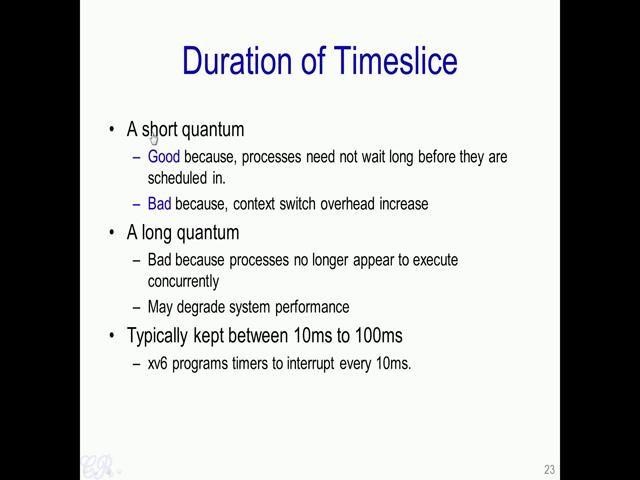


Now let us take another more complex example of round robin scheduling, where we also have arrival times which are not the same. So, P 1 is arriving at the 0th instant; P 2 at the 2nd, P 3 and P 4 at the 3rd and 9th respectively. So, we can similarly draw the Gantt chart and the states of the FIFO for this case. So to start with, in the 0 instant, the only process which has arrived is P 1 and therefore, P 1 executes for 2 cycles. And at this particular point, when the timer interrupt occurs, no other process is present as yet, therefore P 1 will continue to execute for another 2 cycles for another time slice.

However, in this time slice, we have two processes which have entered into the ready queue; these are the process P 2 and P 3. So, P 2 arrives at this interval (3rd cycle) while P 3 arrives at this interval (4th cycle) and they get added into the FIFO. So, at the second time slice completion, there is a context switch and P 2 gets scheduled into the CPU to execute while P 1 which was executing will then go into the FIFO. So, P 2 executes for 2 cycles, then P 3 executes for 2 cycles, then P 1 executes for 2 cycles and at that time P 4 has arrived and gets added into the FIFO.

Now, we have three processes P 1, P 2 and P 4 and these get schedule to run for a period of time. So the average waiting time in this case is 4.75, while the average response time is 2. So, how is the average waiting time 4.75, it means that process P 1 has waited for 7

cycles so, before it completes. So that is 1, 2, 3, 4, 5, 6 and 7. While process P 2 has waited for 6 cycles, so it is 1, 2, 3, 4, 5, 6; process 3 has waited for 3 cycles, and process 4 has waited for 3 cycles .The average response time can be verified to be equal to 2 and the number of context switches was as before 7.



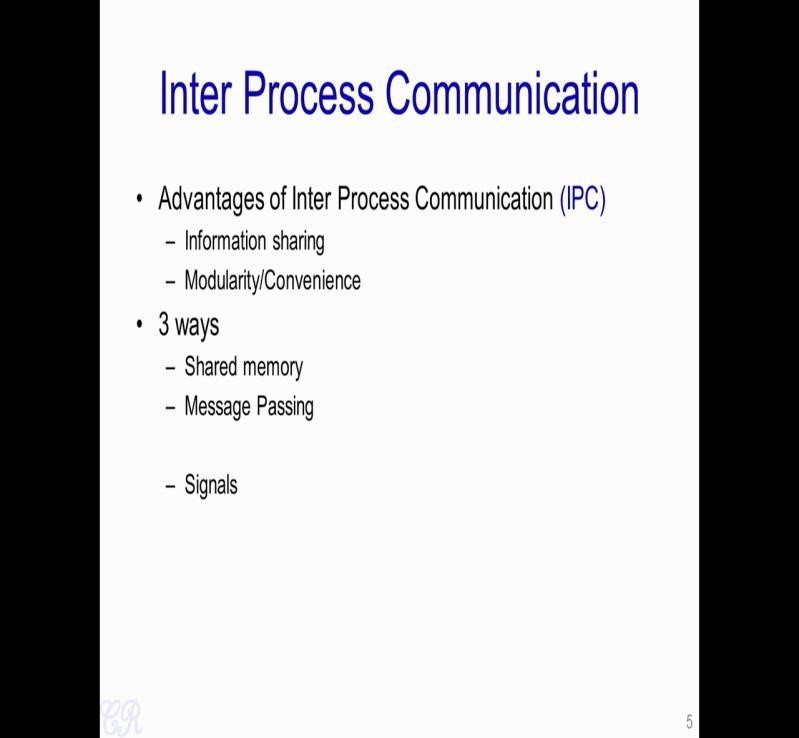
So, essentially, if we have a time slice which is of a very short quantum, the advantage is that processes need not wait too long in the ready queue before they get schedule into the CPU. Essentially this means that the response time of the process would be very good or it would have a less response time.

On the other hand, having a short time slice is bad, because we would have very frequent context switches. And as we seen before context switches could have considerable over heads. Therefore, it degrades the performance of the system. A long time slice or a long quantum has a drawback that processes no longer appear to execute concurrently, it appears more like a first come first serve type of scheduling algorithm and so this again in turn may degrade system performance. So, typically in a modern day operating systems the time slice duration is kept anything from 10 milliseconds to 100 milliseconds. So, xv6 programs, programs timers to interrupt every 10 milliseconds.

So the advantage of the round robin scheduling algorithm is as follows. The algorithm is fair because each process gets a fair chance to run on the CPU. The average wait time is low especially when the burst times vary and the response time is very good. On the other hand, the drawbacks of the round robin scheduling algorithm are as follows; there is an increase number of context switching that occurs and as we have seen before context switching has considerable over heads. And the second drawback is that the average wait time is high especially when the burst times have equal lengths.

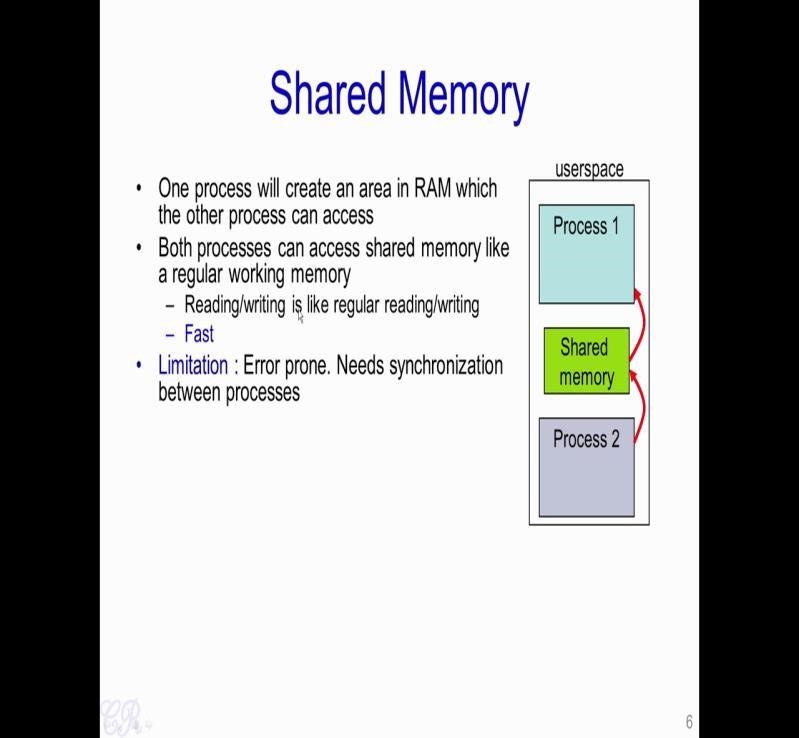
**Inter Process Communication**

we will look at Inter Process Communication. Essentially, when we write large applications it is often quite useful to write them as separate processes. So in order to have an efficient communication between these processes, in order to make things happen efficiently we use inter process communication.



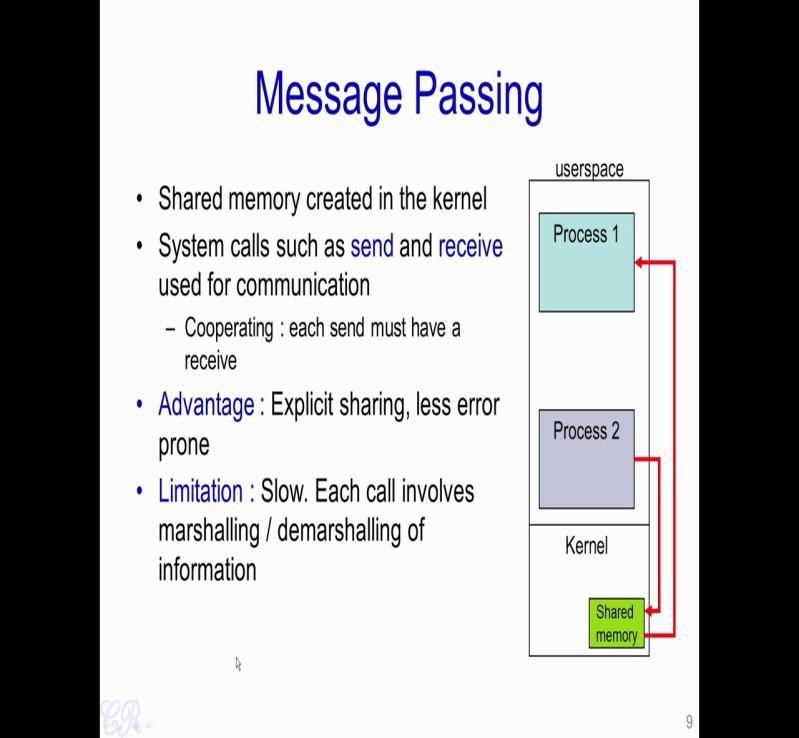
So in order to do this, there is a mechanism known as Inter Process Communication. Essentially with IPC's or inter process communication, two processes will be able to send and receive data between themselves. The advantage of IPC is that the processes could be return to be modular. So essentially each process is meant to do a single job and processes could then communicate with each other through IPC's.

Each processes job is to only focus on a single aspect. The communication between the processes is achieved by inter process communication. In typical operating systems there are 3 common ways in which IPC's are implemented. These are through Shared memory, Message passing and Signals. So let us look at each of these things.

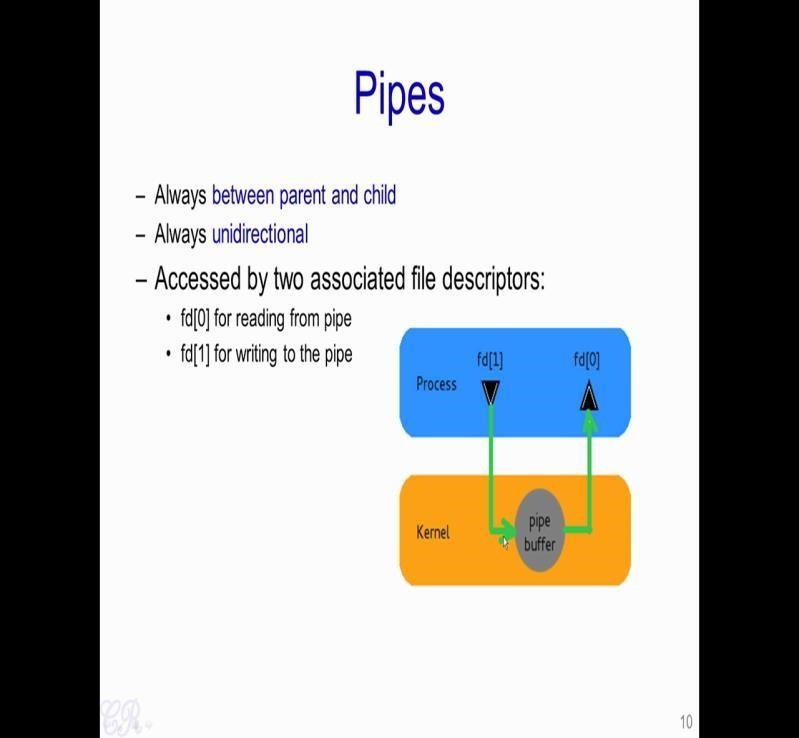


So, with a shared memory we have one process which creates an area in RAM which is then used by another process. So essentially, the communication between process 1 and process 2 is happening through this particular shared memory. Both processes can access the shared memory like a regular working memory, so they could either read or write to this particular shared memory independent of each other. The advantage with this is that the communication is extremely fast, there are no system calls which are involved. And, the only requirement is that you could define an array over here (in shared memory) and then fill the array in the shared memory which can then be read by the other process.

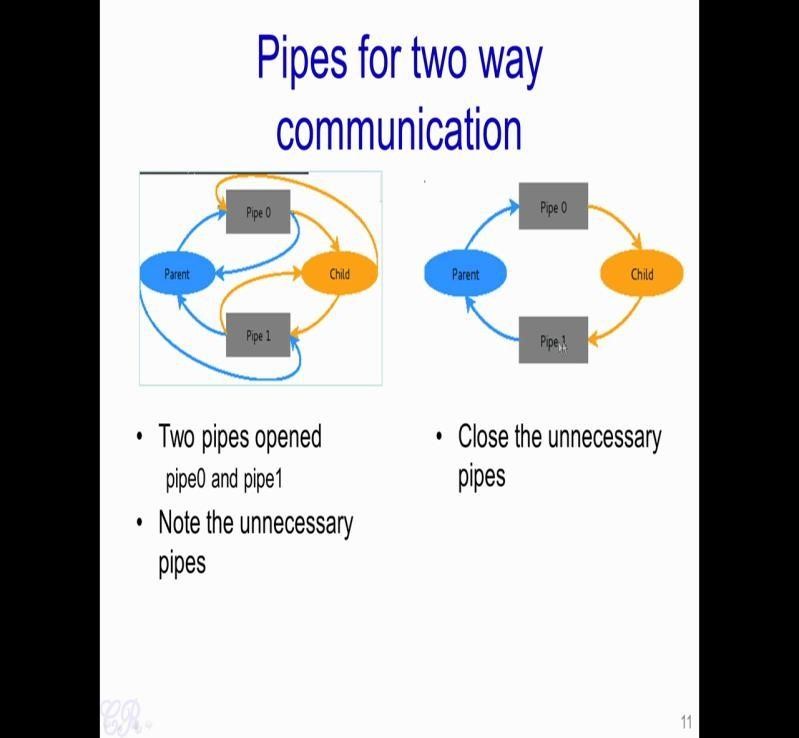
The limitation of the shared memory approach is that it is highly prone to error; it requires the two processes to be synchronized.

  
Next, we will look at the Message Passing. Unlike the shared memory that we just seen where the shared memory is created as part of the user space, in message passing the shared memory is created in the kernel. Essentially we would then require system call such as send and receive in order to communicate between the two processes. If a process 2 wants to send data to process 1, it will invoke the send system call. So this would then cause the kernel to execute and it would result in the data return into the shared memory. While, when process 1 invokes receive, data from the shared memory would be read by process 1.

The advantage of this particular message passing is that the sharing is explicit. Essentially both process 1 and process 2 would require support for the kernel to transfer data between each other. The limitations is that it is slow, each call for the send or receive involves the marshalling or demarshalling of information. And as we know in general a system call has significant overheads. Therefore, message passing is quite slow compared to shared memory. However, it is less error prone then shared memory because the kernel manages the sharing, and therefore would be able to do the synchronization between the process 1 and process 2.



Another very common application or message passing is the use of Pipes. Now pipes are used between parent and child processes only. Essentially, you can only communicate data from a parent process to a child process and vice verse. Another aspect of pipes is that it is unidirectional. Now when a pipe is created generally there are two file descriptors which are associated with it that is fd [0] which is used to read from the pipe, while fd [1]is used to write to the pipe. So we know the unidirectional nature of this particular IPC. So, fd [1] is exclusively used to write to the pipe buffer, while fd [0] is used to read from the pipe buffer.



So in order to obtain two way communication between the parent and the child,we would require two pipes to be obtained; pipe 0 and pipe 1 .So, when pipe 0 is opened it would create two file descriptors; one to write into pipe 0 and the other one to read from pipe 0. Similarly, there are two file descriptors to write to pipe 1 and read from pipe 1. Similarly there are two pair of file descriptors for pipes in the child process.

Now you note that these additional descriptors are not required, therefore we could actually close the extra file descriptors to obtain something like this. Now in order that the parents send some information to the child, the parent will write to pipe 0 while the child will read from pipe 0. In order to that the child sends some information to the parent, the child will write to pipe 1 while the parent will read from the pipe 1.

**Unit 3 (Deadlocks )**

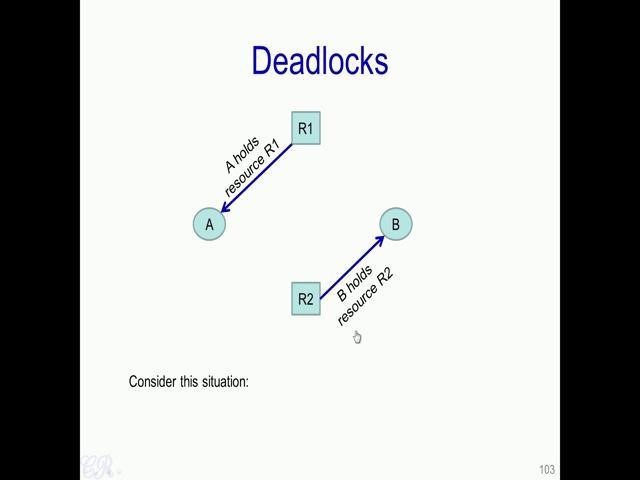
**3.1. Introduction : Deadlock**is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.  
Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s)

**3.2. Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)**  
**Mutual Exclusion:** One or more than one resource are non-sharable (Only one process can use at a time)  
**Hold and Wait:**A process is holding at least one resource and waiting for resources.  
**No Preemption:** A resource cannot be taken from a process unless the process releases the resource.  
**Circular Wait:** A set of processes are waiting for each other in circular form.

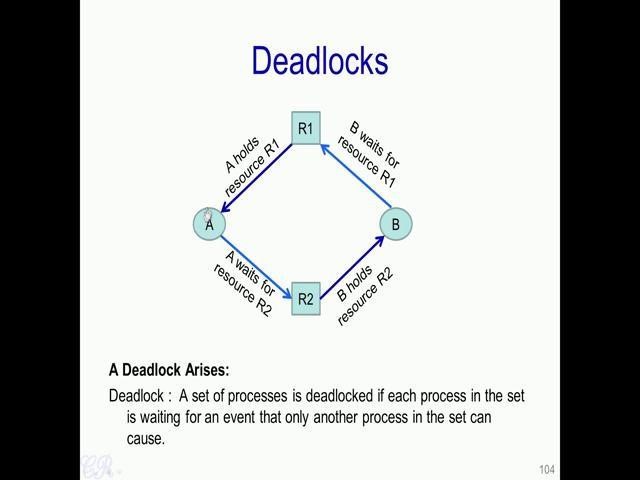
**3.3. Methods for handling deadlock**  
There are three ways to handle deadlock  
1) Deadlock prevention or avoidance: The idea is to not let the system into deadlock state.  
One can zoom into each category individually, Prevention is done by negating one of above mentioned necessary conditions for deadlock.  
Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

2) Deadlock detection and recovery: Let deadlock occur, then do preemption to handle it once occurred.

3) Ignore the problem all together: If deadlock is very rare, then let it happen and reboot.



So let us say we have two processes A and B, and we have two resources R 1 and R 2 . So, these resources could be anything in the system which have a limited quantity. For example, the resources could be as something as small as a The arrow from R 1 to A indicates that A is currently holding the resource. For instance, if R 1 is the file; that means, A has currently open the file exclusively and is doing some operations onto the file. In a similar way, the resource R 2 is held by B. So if it is a printer, for instance, if R 2 is a printer, it means that B is currently using the printer to print some particular document.

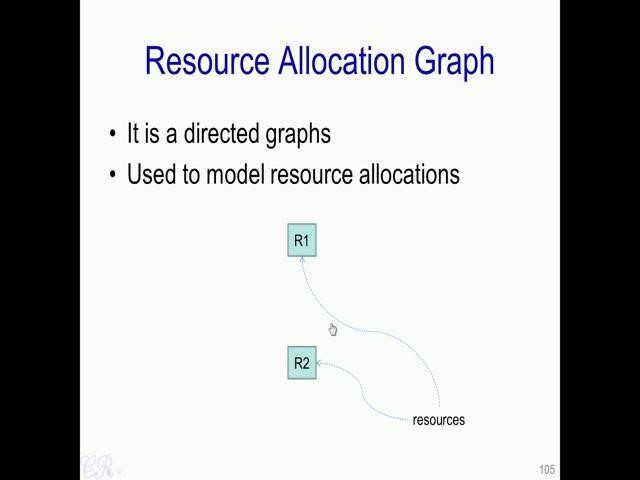


Now, consider this particular scenario where the process A holds the resource R 1, and process B holds the resource R 2; but at the same time, process A is requesting to use R 2. So, essentially the process A is waiting for R 2 to be obtained; and process B is waiting for the resource R 1 to be obtained. So, to take an example, process A is opened the file and is using a particular file which is stored in the disk.

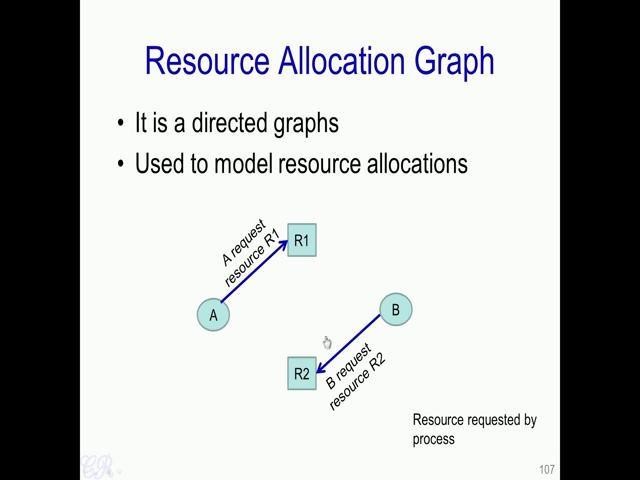
And at the same time, for instance, it wants to print the file to the resource R 2 which we assumed was a printer. Now in a similar way, process B is currently holding this resource (i.e printer) that is using this particular resource (R2) and it wants to open and utilize this particular resource R 1 (i.e file).

So, what we see that over here (as we discussed above), we have a scenario called a Deadlock. Essentially, a deadlock is a state in the system where each process in the deadlock is waiting an event that an other process in that set can cause. For instance, over here , the process A is waiting for the resource R 2 which is held by B; B in turn is waiting for R 1 which is held by A. We have a set of two processes A and B; and each process in the set is waiting for the other process to do something and each process in the set is waiting for the other process to do a particular thing. For example, A is waiting for B to release this particular resource R 2, while B is also waiting for A to release the resource R 1.

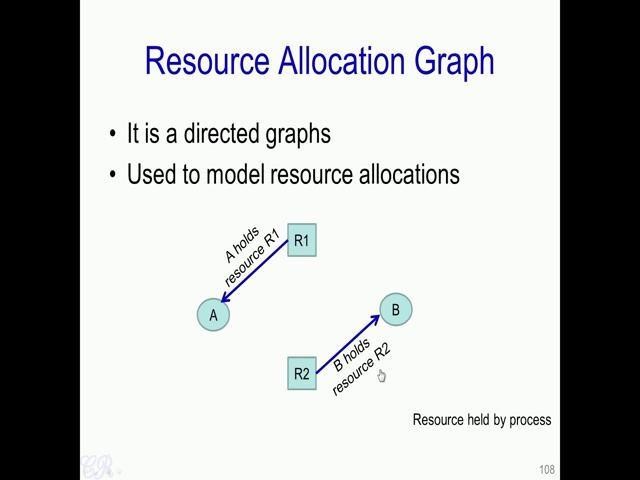
So, deadlock like this is a very critical situation that would occur in systems. And when this deadlocks occur, it could lead to process A and B in this case waiting for an infinite time continuously waiting without doing any useful work. So, such deadlocks should be analyzed thoroughly. So, in this particular video, we will see how such deadlocks are handled in systems. Now, in order to study deadlocks, we use graphs like this, these are known as Resource Allocation Graphs.



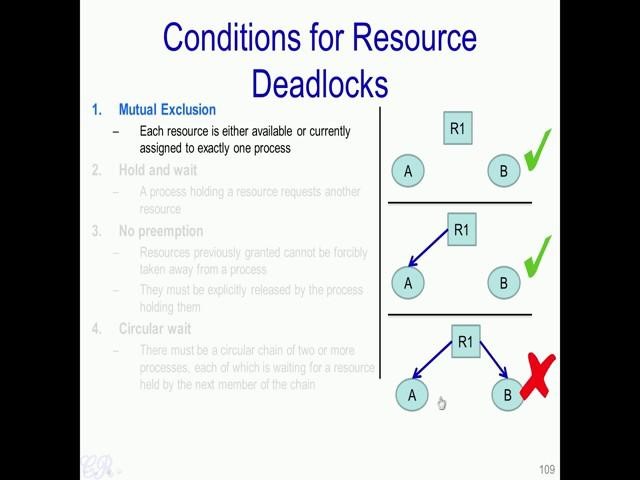
Now resource allocation graphs or directed graphs used to model the various resource allocations in the system. And there by determine whether a deadlock has occurred or a deadlock is potentially going to occur and so on. So, in this directed graph, we represent resources by a square. So, instance R 1 and R 2 are resources and they are represented by the square as shown over. In a similar way, circles as shown over here are used to represent processes.



Arrows from the process to the resource that is directed from the process to the resource would indicate that a request is made for that resource. For example, over here the arrow from A to R 1 indicates that A is requesting for resource R 1; similarly B, in this case is requesting for resource R 2 . So, these requests are made to the operating system and if possible the operating system will then allocate that resource to the corresponding process.



So, when that happens the graph will look like this essentially the direction of the arrow has changed. Now the arrow moves from R 1 to A, indicating that A holds resource R 1. Similarly, the arrow from R 2 to B indicates that B holds resource R 2. There are four conditions in order that a deadlock occurs. So, we will now look at each of these conditions for a deadlock.



So, the first is Mutual Exclusion. So, what we mean by this is that each resource in the system is either available or currently assigned to exactly one process. So, for instance over here we have resource R 1 which is free (1st figure). So, it is not assigned to any particular process, so this is fine. While this is also fine (2nd figure) where the resource is allocated to exactly one process, but in order that deadlocks happen this kind of scenario (3rd figure) should not be present that is the resource cannot be shared between two processes A and B.

The next condition for a deadlock is Hold and wait that is a process holding a resource can request another resource that is for example, in this case (figure mentioned above) the resource R 1 is held by process A. and while having R 1, A is also requesting for another resource R 2, so it essentially holding R 1 and waiting for R 2.

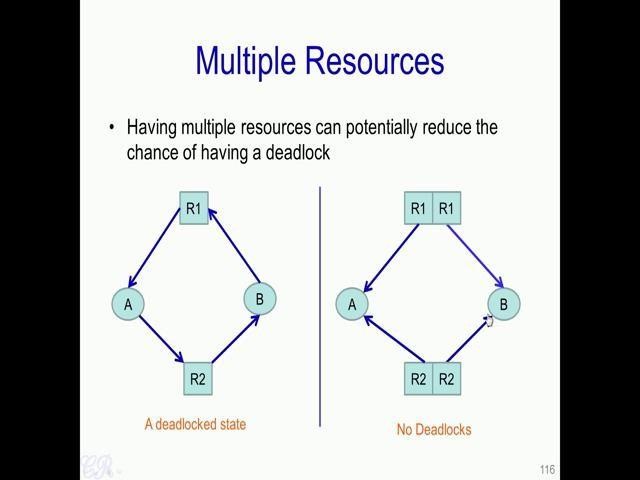
The third condition for a deadlock to happen is No preemption .

Essentially, it should not be the case that resources which an operating system previously granted for a particular process is forcibly taken away from that process. That is the OS or another entity in the system cannot forcibly remove a resource which has been allocated to a particular process. Instead processes should explicitly release the resource by themselves that is whenever the process wants it should release the resource by itself.

So, a fourth requirement is the Circular wait . What this means is that there is a circular chain of 2 or more processes, each of which is waiting for a resource held by the next member of the chain. So, we see over here (mentioned in above circular figure) with that we have a circular chain and there is a wait over here because process A is waiting for process B to release resource R 2; and process B is in turn waiting for process A to release the resource R 1. So, we have a circular wait condition over here.

If for instance, we were able to build a system where one of these conditions, were not present. For example, suppose we build a system where processes cannot hold a particular resource and wait for another resource at the same time. So, such system would never have any deadlocks.

On the other hand, suppose a system has been developed where all of these things are possible that is there is a mutual exclusion when using resources, a process could hold a resource and wait for another one, once allocated they cannot be forcefully preempted from the resource and circular wait mechanisms are allowed then deadlocks could potentially occur. So, having all these conditions does not imply that a deadlock has occurred. It only implies that there is a probability of deadlock occurring in the future.

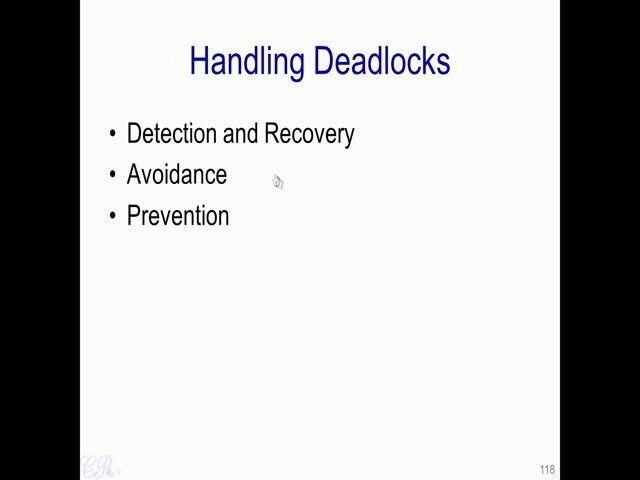


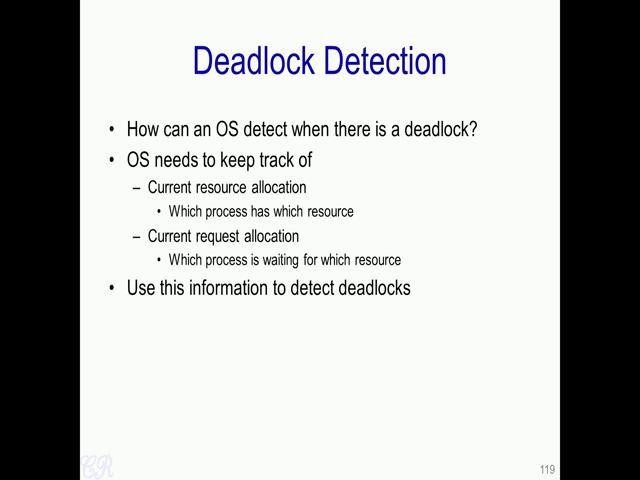
Now, we have seen that deadlock is indeed a probabilistic event and could occur with some probability. Now one way to reduce the probability is by having multiple resources present in the system. So, we had seen that this was a deadlock state , essentially because A is waiting for resource R 2 to be released by B, and B in turn is waiting for resource R 1. Now one way this can be solved is by having multiple resources, while this particular solution will not always work, and essentially depends on the type of resources, it may help to some extent.

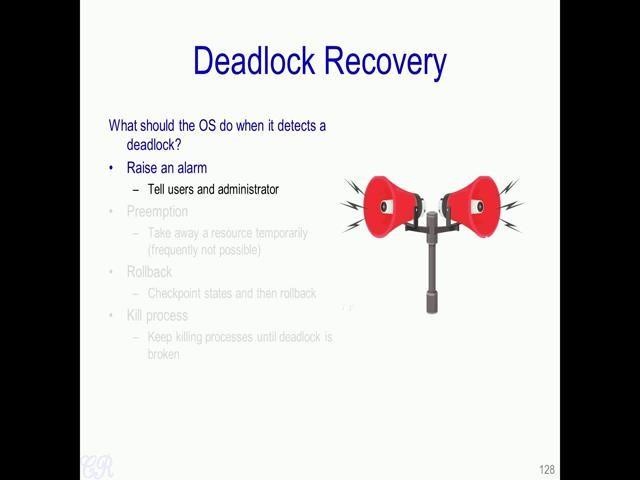
For instance, if we have 2 resources of exactly the same type then both A’s request as well as B’s request could be managed that is the resource if we have 2 types of R 1 or in other words if you have a duplicate of the resource R 1 then that can be given to A as well as B. Similarly, a duplicate of the resource R 2 can be given to A and B simultaneously (mentioned in above image right side). So, what this means is that for example, we could have 2 printers present and A can be allocated one printer while B could be allocated the other printer while this does not completely eradicate deadlocks, it may reduce the likelihood the deadlocks may occur.

**3.4. Dealing with Deadlocks**

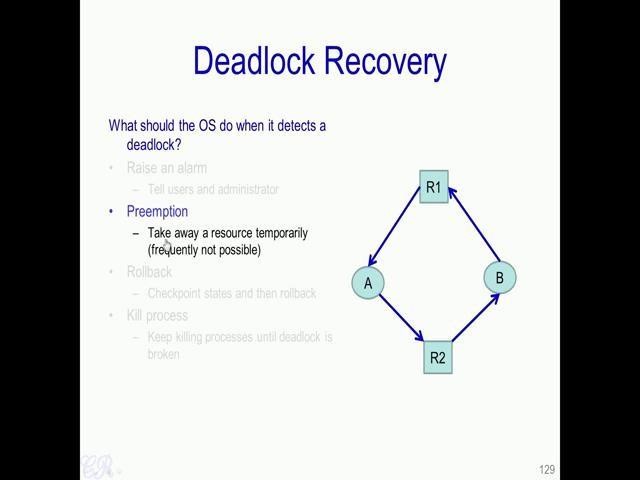
So there are 3 ways to handle Deadlocks. That is by Deduction and Recovery from Deadlocks, Deadlock Avoidance and Deadlock Prevention.



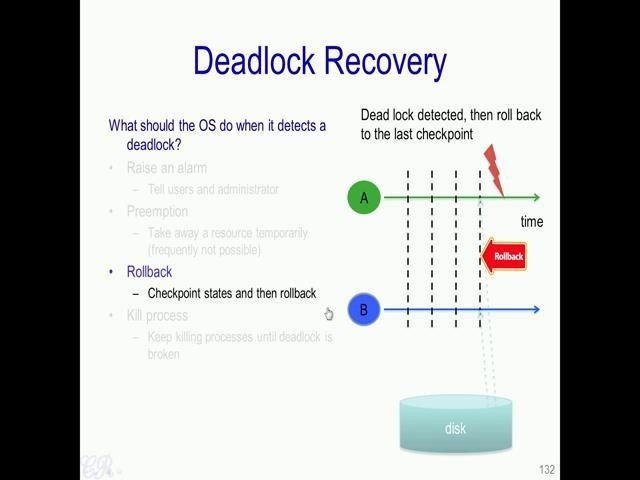




So, once the deadlocks is detected, what next should the operating system do? So, there could be various things that the OS could do. So, one thing is that it could raise an alarm, that is tell users and administrator that indeed deadlock has been detected.



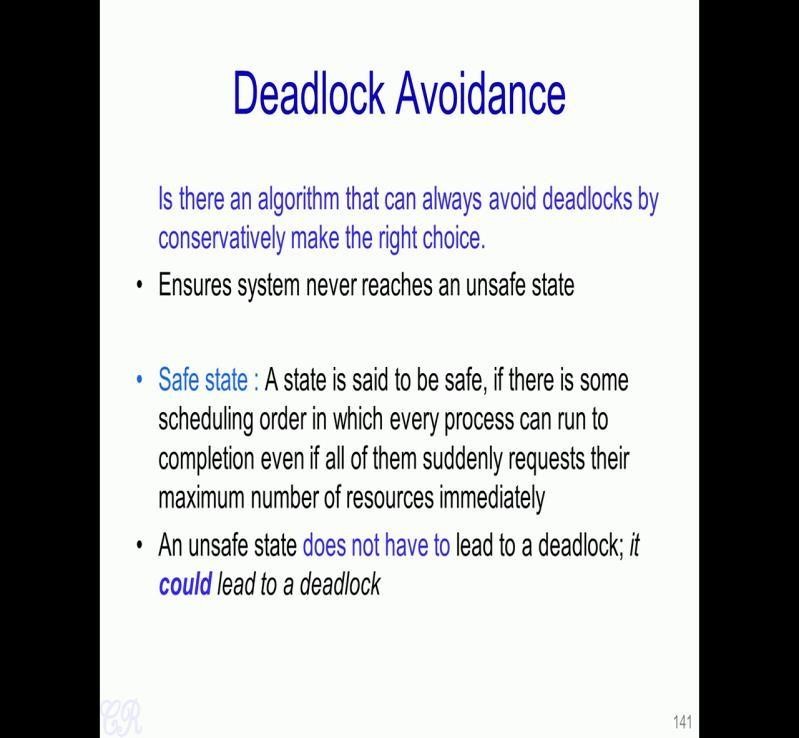
Then a second way is to force a preemption that is you force a particular resource to be taken away from a process and given to another process. For instance, it could be like R 2 was a printer and it is currently held by B. So, what could be done was that the printer could be forced to be taken away from B, while it is allocated to A for some time and thus the deadlock will be broken, this is as shown over here. So, B no longer has other resource R 2, but R 2 is given to A.

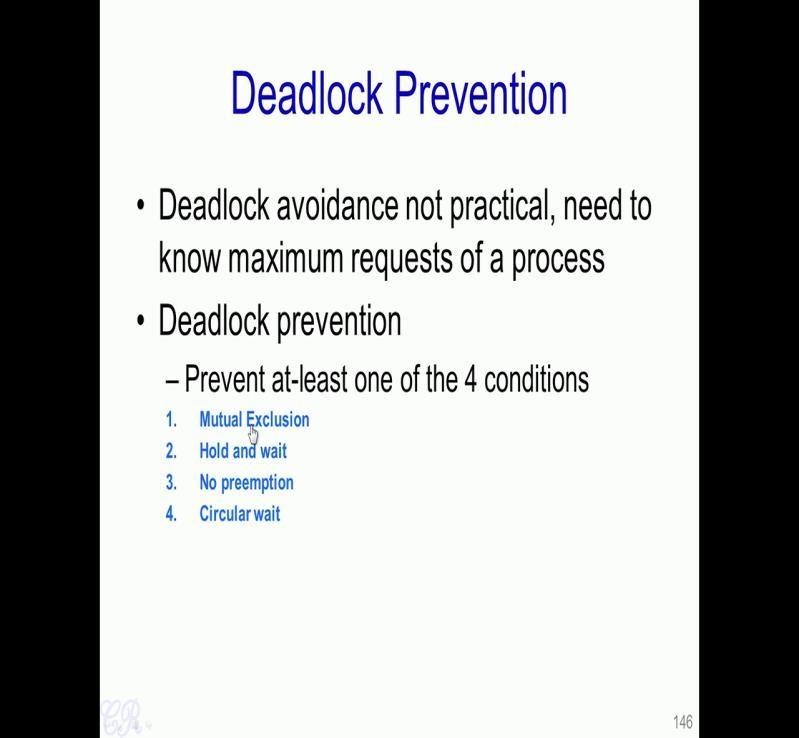


Then a third method is by using a technique known as rollback. So, with rollback, both processes A and B as they execute will be check pointed . So, by check point we mean that the state of the process gets stored on to the disk. The process executes for some time and then the entire state of the process gets stored on to the disk. Now storing the state of the process on the disk, will allow the processes to execute from the point where it has been check pointed, now the checkpoint state. So, as time progresses more check points are taken periodically as shown over here (dotted lines in above image).

Now process A and B will continue to execute from this state (last dotted lines in above image) and the deadlock may not occur again. Essentially we have seen that since deadlock is a probabilistic event by modifying the ordering in which the allocations are made we could prevent this particular deadlock.

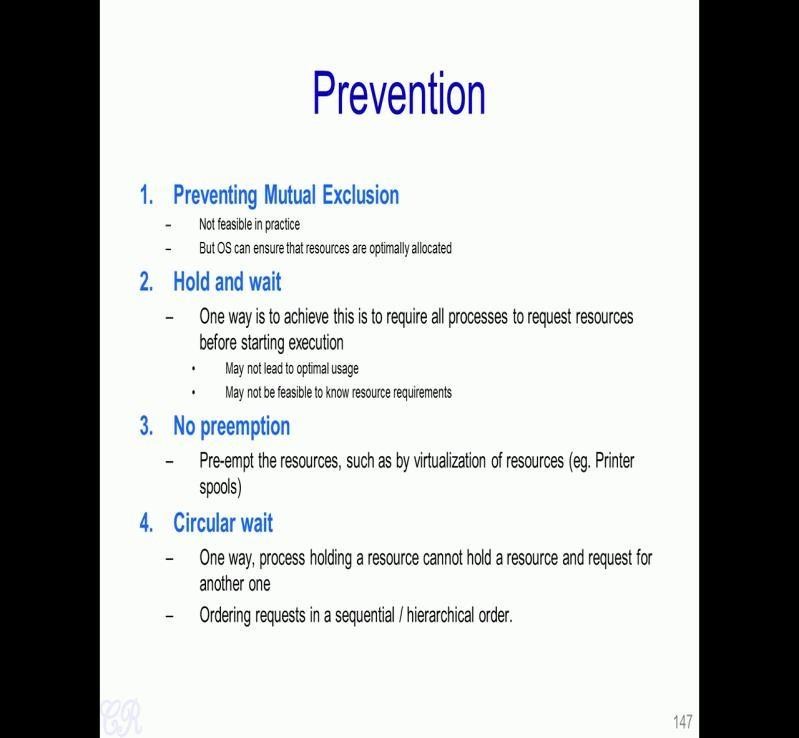
A fourth way is to kill processes. Essentially, if process A and B are in a deadlock state, killing one of these processes would break the deadlock. So, typically the lower important or the less priority process would be killed.





The third way to prevent deadlocks is what is known as Deadlock Prevention. Essentially we prevent deadlocks by designing systems where one of the 4 conditions is not satisfied . So, we have seen these 4 conditions and these are the conditions which are essential for a deadlock to occur. By preventing one of these conditions from holding in the system, we can therefore prevent deadlocks. For example, if we design a system where hold and wait condition cannot be satisfied, that is a process cannot hold a resource while waiting for another resource then such a system will not have a deadlock.

So, let us see various ways in which we can actually prevent one of these conditions from happening .



So Preventing Mutual Exclusion, so in practice this is not feasible. We will not be able to always prevent mutual exclusion. For example, take the case of a printer resource so; it always needs to work or print corresponding to a particular process. So, it cannot be simultaneously shared with two processes unless it has a spool present in it. However, this being said, the operating system can ensure that the resources are optimally allocated.

So let us see the Hold and wait. So, one way to achieve this is to require all processes to request their resources before they start executing. So, this obviously is not an optimal usage of the resources and also may not be feasible to know the resource requirements during the start of execution.

Now, let us look at the third way there is no preemption. So, Pre-empt the resources such as by virtualization of resources (example by printer spools). So, in a printer spool we could have several processes sending documents to be printed at exactly the same time. So, all these documents are spooled or buffered in the printer and eventually each document is then printed one after the other.

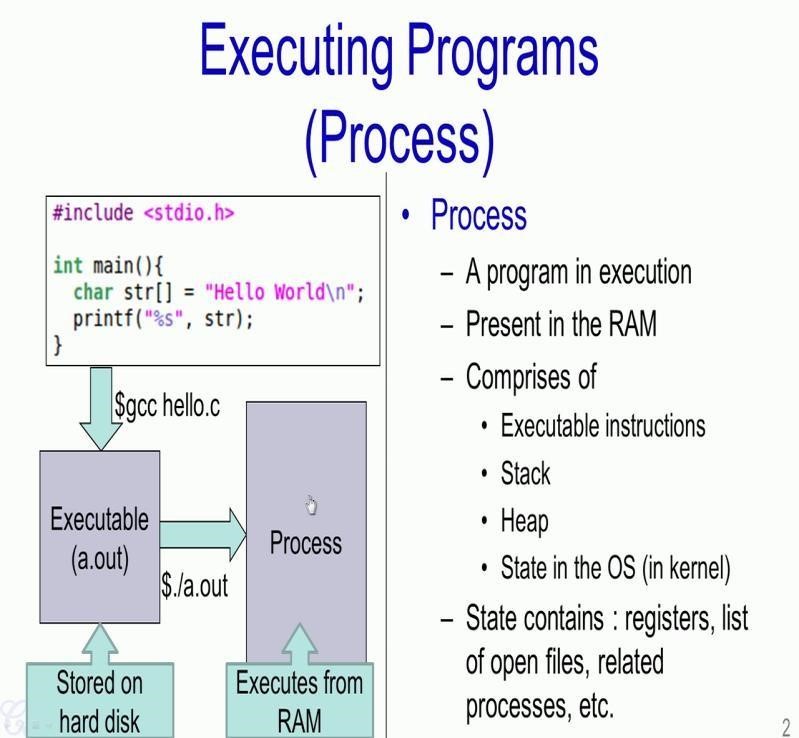
The final technique we can target is the Circular wait. So one way to prevent this is that process holding a resource cannot hold a resource and request for another one at the same time. Second way to prevent the circular wait is by ordering requests in a particular order - either sequential or hierarchical order.

**Unit 4 (Memory Management)**

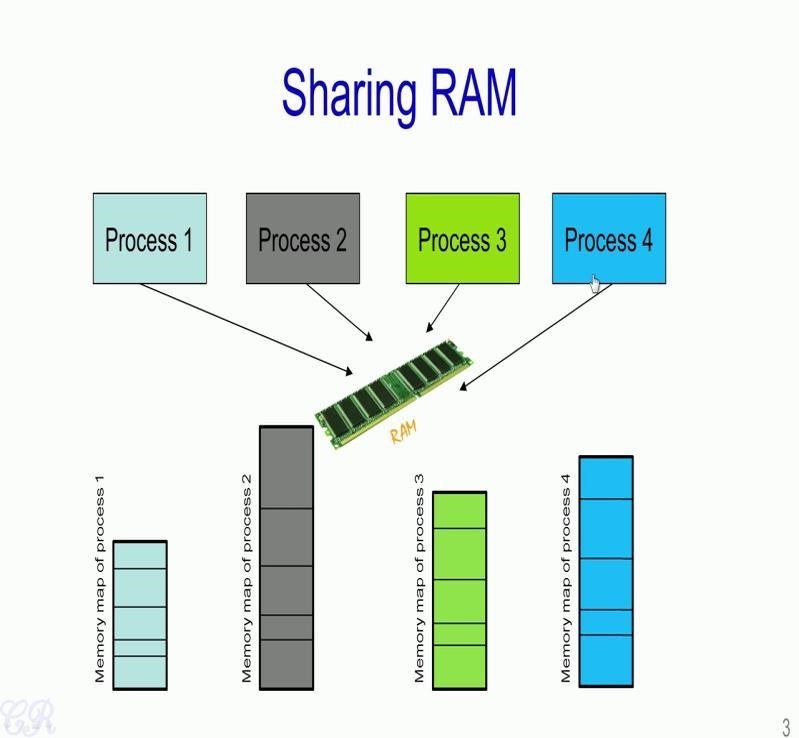
**4.1. Introduction:** The second most important hardware resource in the system is the Memory.

when we take up a particular program and this could be any program written in any language and when we compile it, we will get what is known as an Executable.

Now, the executable will be stored on the hard disk and whenever a user runs this program or executes this program, it creates what is known as a Process. So, this process is created by the operating system and it executes in RAM. So, essentially what the operating system would do is that the executable stored in the hard disk, would be loaded into the RAM and then on execution is passed to this particular program and this program will then execute in the CPU.



So, as we have seen this particular process which is present in memory comprises of several segments, such as the text segment which contains the Executable instructions, the Stack, Heap and also as we have seen some hidden metadata in the operating system such as registers, list of open files and related processes. So, all these actually constitute the process. Now, what is important for us is this process and it is presents in the RAM.

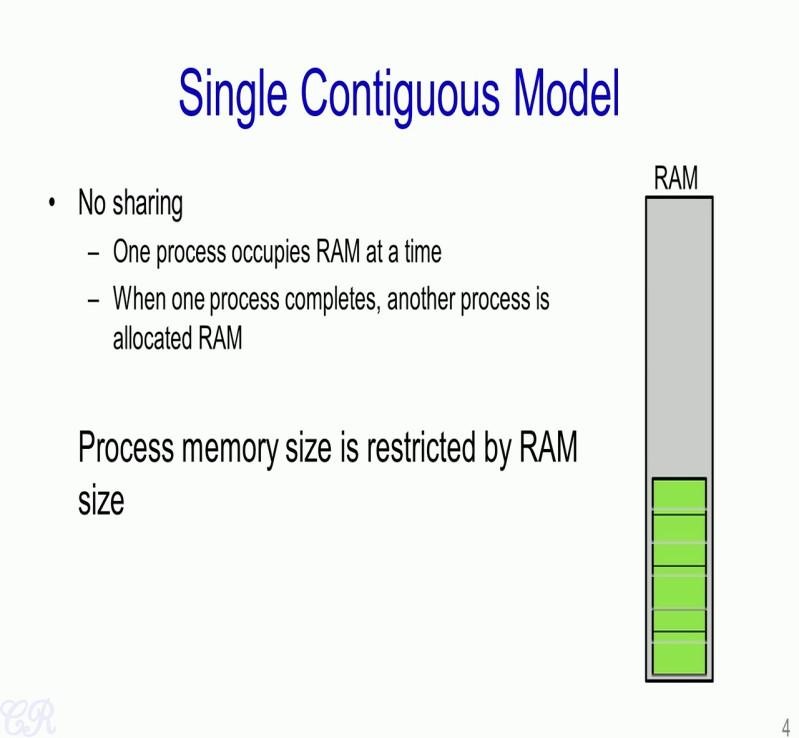


So as we know the RAM or Random Access Memory also called as the Main Memory is a limited resource in the system. So, each system would have something like 4, 8, 16 or 32 GB of RAM. At the same time we may have multiple processes executing almost simultaneously. So, like we have seen in the previous video (Lecture 4 video), these processes may execute in a time sliced manner and if there are multiple processor present in the system, then these processors could also execute in parallel.

However, in all these cases, these processes and their corresponding memory map should be present in the RAM. Essentially the memory map corresponding to process 1, the memory map corresponding to process 2, memory map of process 3 and the memory map of process 4 should be present in the RAM, in order that these 4 processes execute in parallel or in a multi tasked environment.

One of the most primitive ways of managing memory especially done for the older on

a operating systems is what is known as the Single Contiguous Model. So, essentially in this we have a RAM over here which is the RAM of the system and what is ensure or what is done by the operating system is that this RAM is occupied by one process at a time. Essentially at any particular instant there is only one process and it is memory map that is present in the RAM. So, after this process completes executing will the next process will be loaded into the RAM.



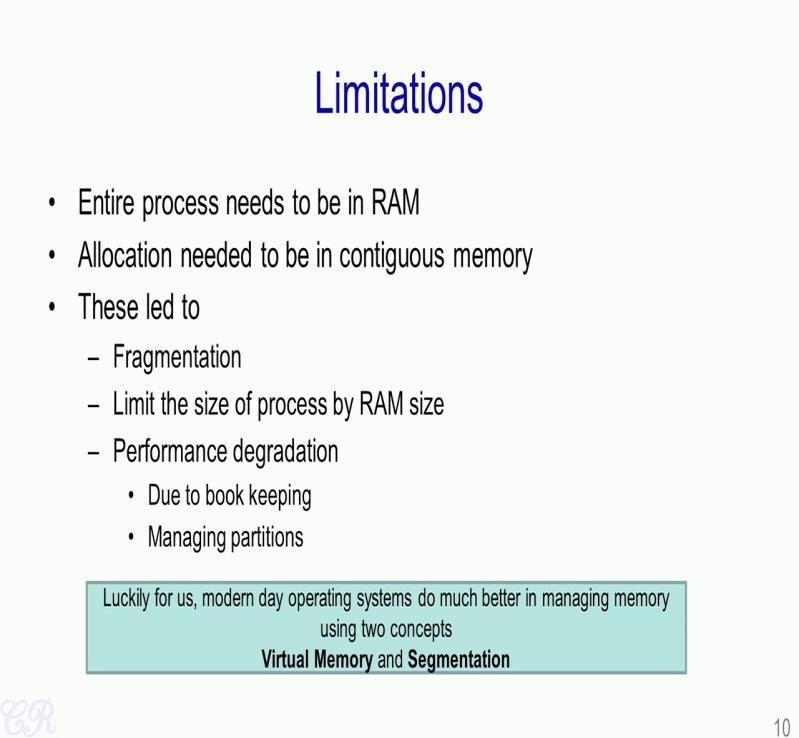
So, the drawbacks are quiet obvious.

The first is that we are forced to have a very sequential execution. When one process completes, only then the second process could occupy the RAM and so on. Another limitation of this particular model that is a single contiguous model is that the size of the process is limited by the RAM size. For instance, let us say we have a RAM which is of say 12 KB while this is seems to be a very small amount of RAM, this size of RAM is quiet common in embedded system. So, given this RAM of say 12 KB and let us say our process size is of 100 KB then it is quite obvious that this process cannot execute using this RAM. Essentially the RAM is not sufficient to hold the entire process.

The **contiguous model** is what is known as a Partition Model. Essentially in this model at any instant of time, we could have multiple processes that occupy the RAM simultaneously. For instance, in this particular case we have two processes. This blue process and the green process that occupy the RAM and therefore, the processor could then execute this process as well as this process either in parallel or in a time sliced manner.

Now, in order to manage such a partitions, the operating system would require something known as a Partition Table. So, typically this partition table would also be present in the RAM. It will be present in an area which is not shown over here. So, the partition table would have the base address of a process, the size of the process and a process identifier. For instance, the blue process has a memory or a base address of 0x0 indicating that is starting from 0th location in the RAM and this process has a size of 120 KB. So, 120 KB means up to this point.

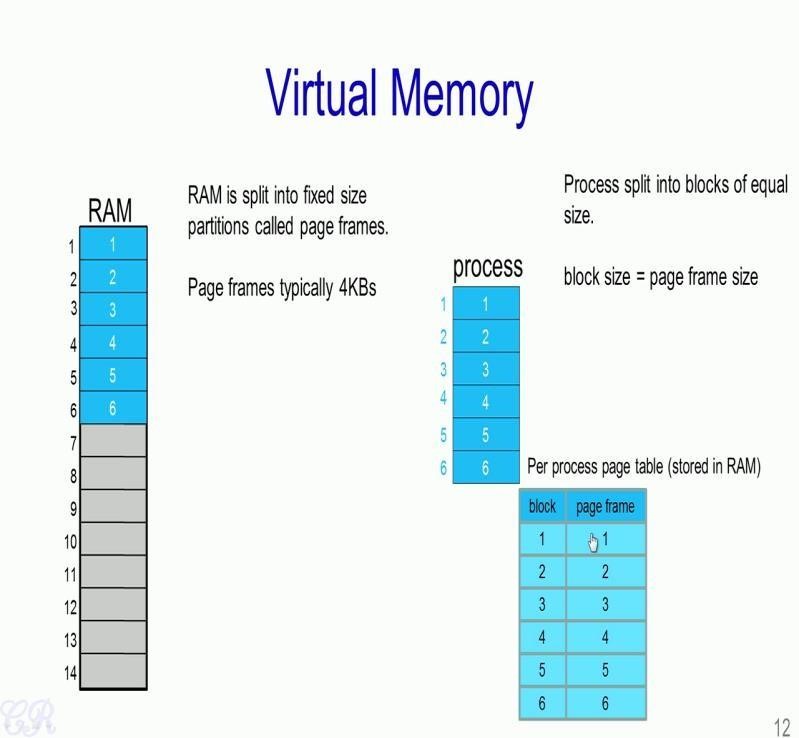
There is also a flag known as the usage flag which mentions whether this particular area in RAM is in use or it is free. For instance, let us take process 1 that is this one, the green one over here shown over here . So, this process 1 starts at the memory location 120K that is this point and has a size of 60K. So, it extends up to 180K and this area is also in use. Now, there is another entry in the partition table which is specified as free. So, this starts at 180K and extends for a size of 30K. So, this corresponds to this white area over here . So, the operating system could possibly use this free memory to run perhaps the third process and therefore, would be able to have three processes present in the RAM at the same time.



So, thus we have seen that the major limitation of the single contiguous model as well as the partition model is that the entire memory map of the process needed to be present in RAM during its entire execution. So, all allocation for the entire process needed to be in contiguous memory and because of these issues, it had led to fragmentation, limit on the size of the process due to base on depending on the RAM size and also performance degradation due to book keeping and also the management of partitions. So, luckily for us modern day operating systems do much better in managing memory. So, most modern day operating systems use two concepts that is virtual memory and segmentation.

**4.2. Virtual Memory**

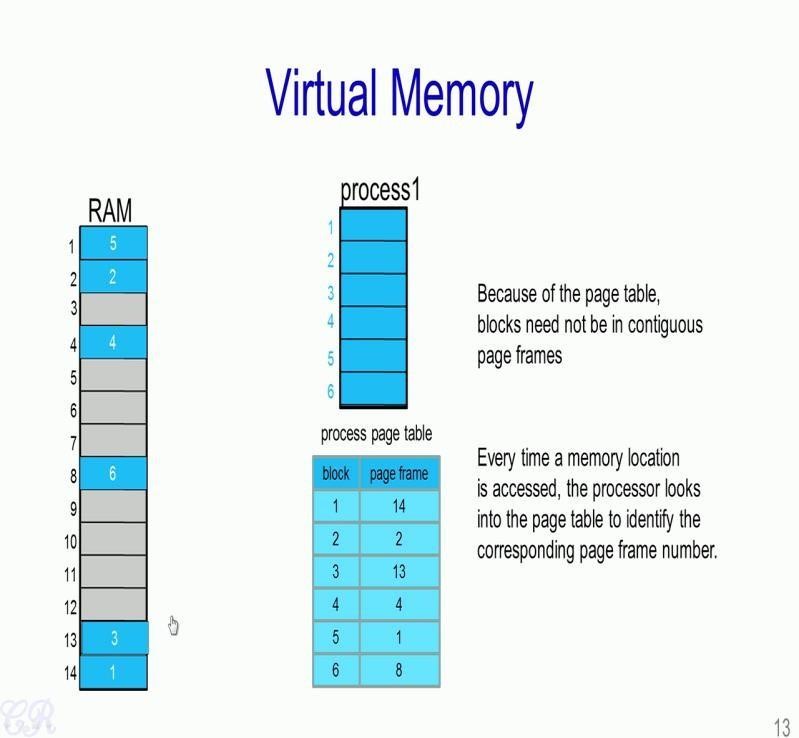
The most commonly used memory management technique in systems these days.



So in a virtual memory system, the entire RAM which is present in the system is split into equal sized partitions called Page frames. So typically, page frames would be of 4 kilobytes each. So in this RAM for instance, we have 14 page frames; which are numbered 1 to 14. And, each of these page frames have the same size.

In the previous Intel processors, all pages were fixed at 4KB. But, in more recent processor. we could have pages or page frames, which are of a larger size. In a similar way, the process which executes in the CPU or the process map corresponding to the process is also split into equal sized blocks. Now, the split of a process is in such a way that the block size of a process that is this size is equal to the page frame size; that is the size of each block in this process is equal to the page frame size.

Now because we have split the RAM like this as well as the processor’s memory in a similar way, what we can then do is allocate blocks in a process to page frames in the RAM. Additionally, what the operating system maintains is a table. This table is also present in the RAM and not shown over here. But, what it contains is the mapping from the process blocks to the page frames. For instance over here, the mapping is very simple; block 1 of the process gets mapped into page frame number 1, that is, block 1 of the process gets mapped into page frame number 1, block 2 gets mapped to page frame 2, block 3 gets mapped to page frame 3, and so on.



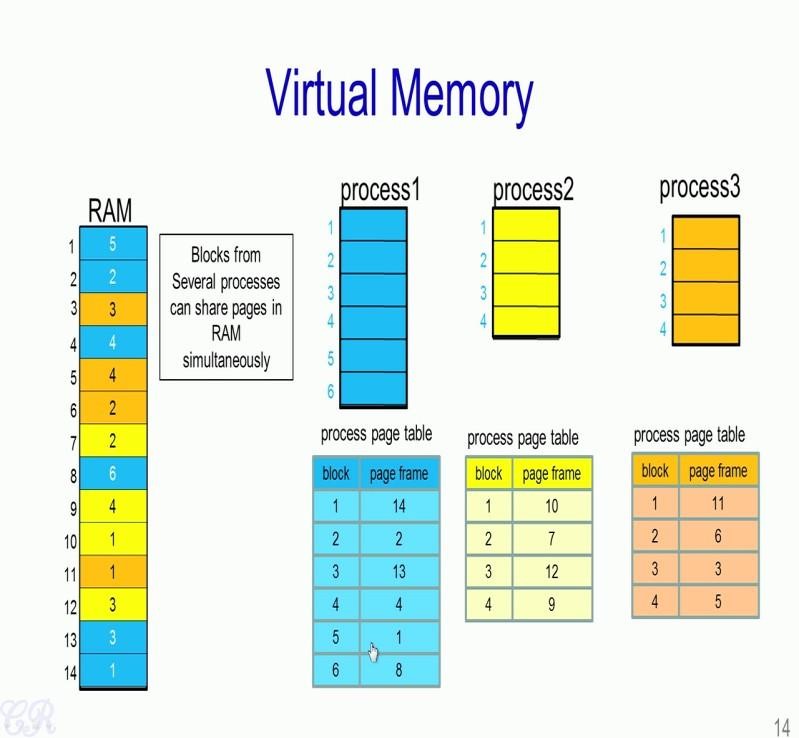
Now because we have such a table which provides the mapping between blocks of a

process to the corresponding page frame, what we can then do is we could have any kind of mapping that we choose. For instance, now we have block 1 of the process present in block 14 that is over here block 2 of the process present in page frame 2, block 3 of the process in page frame 13 and so on.

Essentially, what we are able to achieve with this particular process page table is that the blocks of the process need not be in contiguous page frames in the RAM. However, the overhead that we incur is that every time a particular memory location needs to be accessed in the processors memory map. The CPU would need to look up the processors page table, obtain the corresponding page frame number for that particular address and only then can the RAM be addressed or accessed. For instance, let us say that the program executing in the CPU accesses a memory location inside a block 3 of the process. So, this could be an instruction or a load or store to some data.

Now, when such an operation is executed a memory management unit present in the processor would intercept this access. And then, it is going to look up in to the page table and find that the corresponding page frame, which towards block number 3 is 13. Then, it is going to generate something known as a physical address, which will then look into the 13th page frame in RAM. As a result, every memory access has the additional overhead of looking into the page table, before the access into RAM can be made possible.

So this look up into the page table is the extra overhead. And, typically this overhead is partially mitigated by using something known as a TLB cache or a Translational lookaside Buffer cache. So, we will not go into details about the TLB cache. But, for our understanding with respect to the course we are studying, we need to remember that every memory access or every load or store of instructions fetched by the process during its execution would need to look up a particular page table and only then can the RAM be accessed.



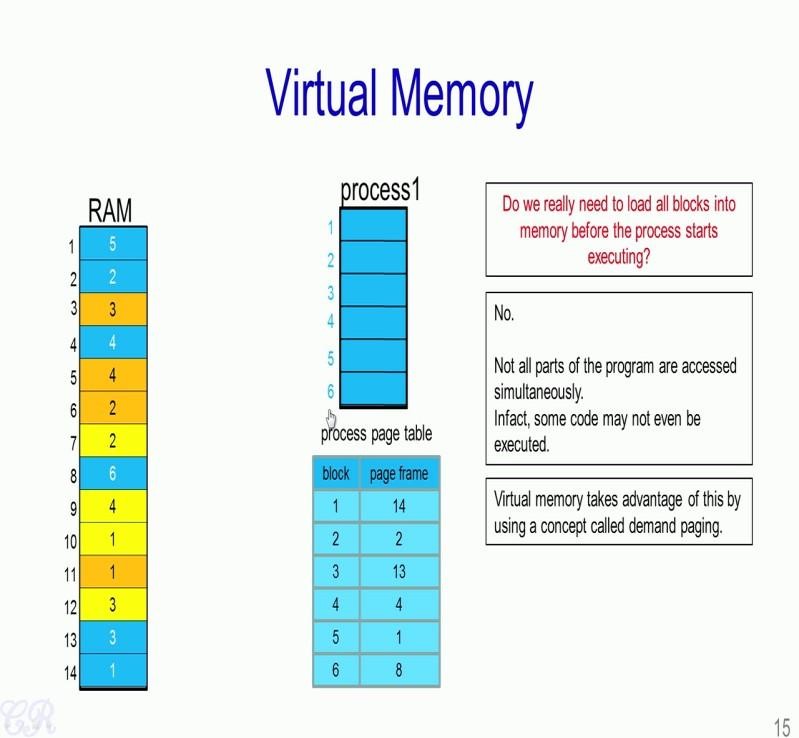
Now, what we mentioned was that every process executing in the system would have its own process page table Thus, if you have a second process that is process 2, a process 2 will also have associated page table; process 3 will also have its associated page table. Now, these processes or these processes as we have seen the memory associated with these processes are present in the user space region of the memory. However, the process page tables are present in the Kernels space. And therefore, any program you write in the user space, but not be able to determine what the process page table mappings are.

Now, since we have multiple processes executing in the system, each process could have various page frame mappings. For instance, process 1 has a mapping of all the blue page frames in this particular RAM; while process 2 has all the yellow page frames; when the third, process 3 has all the orange page frames.

Now, what we can see is that we are capable of sharing the RAM with multiple processes simultaneously. Now, assume that we have a system with a single CPU. This means that at a particular point in time, only a single process will execute. Now, this process will continue to execute until its time slice completes.

So, during the execution of the process, that is when for instance, process 1 is process’s executing in the CPU, this particular process page table will be the active page table. Therefore, any instruction or data which is loaded or stored will have, will look up this particular process page table to get the corresponding page frame in RAM. When there is a context switch from process 1 to say process 2, it would be this page table, which will become active.

So, any address that is accessed in process 2 will get the corresponding page frame number from this active page frame. Similarly, when process 3 gets executed in the CPU, this process age table will be the active page table. So, what we can see is that it is not possible for process 2 to access any of the page frames corresponding to process 1 or process 3.



The next thing we would look at is that do we really need to load all blocks into memory before the process starts to execute ? So, in the earlier video we had seen that when a process executes, the entire process was loaded into a RAM. So, now we ask ourself that is such a loading actually required? And, the answer is no. Essentially, what we notice about programs is that not all parts of the programs are access simultaneously.

In other words, there is a locality aspect in the way a particular program executes. What this means is that if some instructions are executed in let us say this particular block it is highly likely that the next set of instructions or the next few instructions that follow this would also be present in this block. So, once there is a instruction to another block, say block 3, it is quite likely that because of the locality of the program, there would be quite a few instructions executed from this block and so on.

In fact, you may have, and it is quite often the case that there may be several parts of the process memory, which are not even accessed at all. That is, they are neither executed, loaded or stored from memory. So, what the virtual memory concept does is that it takes advantage of this locality aspect during the execution by using a concept known as Demand Paging.

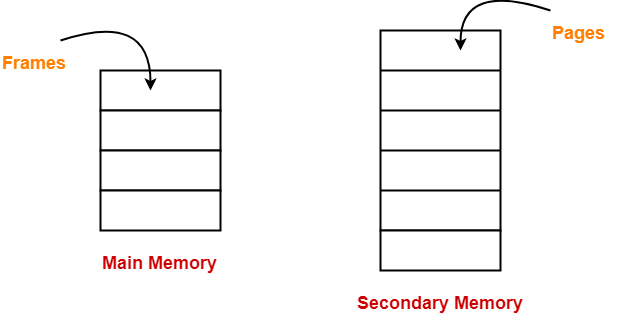
There are two popular techniques used for non-contiguous memory allocation-

1. Paging
2. Segmentation

## ****4.3. Paging****

 Paging is a fixed size partitioning scheme.

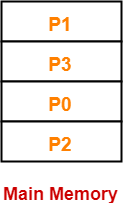
* In paging, secondary memory and main memory are divided into equal fixed size partitions.
* The partitions of secondary memory are called as **pages**.
* The partitions of main memory are called as **frames**.



* Each process is divided into parts where size of each part is same as page size.
* The size of the last part may be less than the page size.
* The pages of process are stored in the frames of main memory depending upon their availability.

## ****Example-****

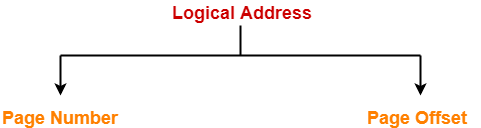
* Consider a process is divided into 4 pages P0, P1, P2 and P3.
* Depending upon the availability, these pages may be stored in the main memory frames in a non-contiguous fashion as shown-



## ****Step-01:****

CPU generates a logical address consisting of two parts-

1. Page Number
2. Page Offset



* Page Number specifies the specific page of the process from which CPU wants to read the data.
* Page Offset specifies the specific word on the page that CPU wants to read.

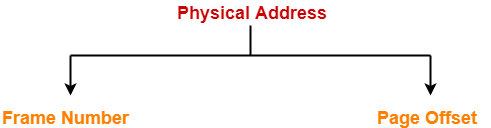
## ****Step-02:****

For the page number generated by the CPU,

* [**Page Table**](https://www.gatevidyalay.com/page-table-paging-in-operating-system/) provides the corresponding frame number (base address of the frame) where that page is stored in the main memory.

## ****Step-03:****

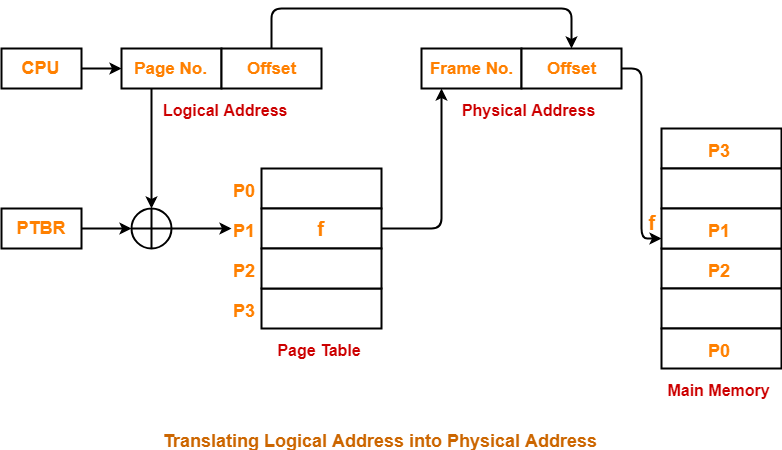
* The frame number combined with the page offset forms the required physical address.



* Frame number specifies the specific frame where the required page is stored.
* Page Offset specifies the specific word that has to be read from that page.

## ****Diagram-****

The following diagram illustrates the above steps of translating logical address into physical address-



## ****Advantages-****

The advantages of paging are-

* It allows to store parts of a single process in a non-contiguous fashion.
* It solves the problem of external fragmentation.

## ****Disadvantages-****

* It suffers from internal fragmentation.
* There is an overhead of maintaining a page table for each process.
* The time taken to fetch the instruction increases since now two memory accesses are required.

# 4.4. Segmentation

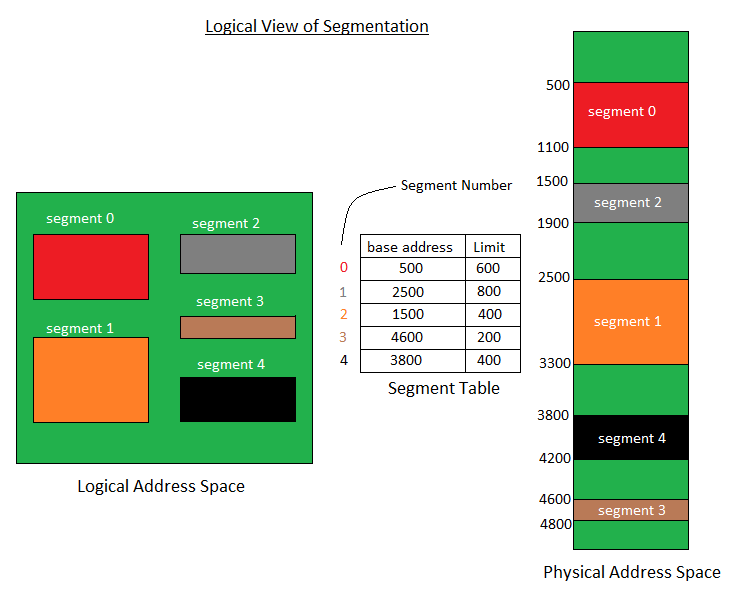
A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the same sizes are called segments. Segmentation gives user’s view of the process which paging does not give. Here the user’s view is mapped to physical memory.  
There are types of segmentation:

1. **Virtual memory segmentation –**  
   Each process is divided into a number of segments, not all of which are resident at any one point in time.
2. **Simple segmentation –**  
   Each process is divided into a number of segments, all of which are loaded into memory at run time, though not necessarily contiguously.

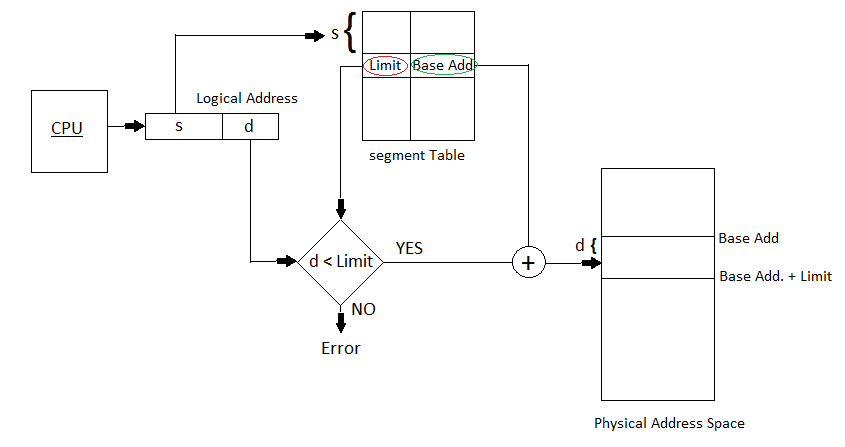
There is no simple relationship between logical addresses and physical addresses in segmentation. A table stores the information about all such segments and is called Segment Table.

**Segment Table –** It maps two-dimensional Logical address into one-dimensional Physical address. It’s each table entry has:

* **Base Address:**Itcontains the starting physical address where the segments reside in memory.
* **Limit:** It specifies the length of the segment.



Translation of Two dimensional Logical Address to one dimensional Physical Address.



Address generated by the CPU is divided into:

* **Segment number (s):** Number of bits required to represent the segment.
* **Segment offset (d):** Number of bits required to represent the size of the segment.

**Advantages of Segmentation –**

* No Internal fragmentation.
* Segment Table consumes less space in comparison to Page table in paging.

**Disadvantage of Segmentation –**

* As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing External fragmentation.

## Differences Between Paging and Segmentation

1. The basic difference between paging and segmentation is that a page is always of **fixed block size** whereas, a segment is of **variable size**.
2. Paging may lead to **internal fragmentation** as the page is of fixed block size, but it may happen that the process does not acquire the entire block size which will generate the internal fragment in memory. The segmentation may lead to **external fragmentation** as the memory is filled with the variable sized blocks.
3. In paging the user only provides a**single integer** as the address which is divided by the hardware into a **page number and Offset**. On the other hands, in segmentation the user specifies the address in two quantities i.e. **segment number and offset**.
4. The size of the page is decided or specified by the **hardware**. On the other hands, the size of the segment is specified by the **user**.
5. In paging, the **page table** maps the **logical address to the physical address**, and it contains base address of each page stored in the frames of physical memory space. However, in segmentation, the**segment table** maps the **logical address to the physical address**, and it contains segment number and offset (segment limit).

**Unit5. (I/O Management)**

**5.1. Introduction:** Input/Output devices are the devices that are responsible for the input/output operations in a computer system.

Basically there are following two types of input/output devices:

* Block devices
* Character devices

## Block Devices

A block device stores information in block with fixed-size and own-address.

It is possible to read/write each and every block independently in case of block device.

In case of disk, it is always possible to seek another cylinder and then wait for required block to rotate under head without mattering where the arm currently is. Therefore, disk is a block addressable device.

## Character Devices

A character device accepts/delivers a stream of characters without regarding to any block structure.

Character device isn't addressable.

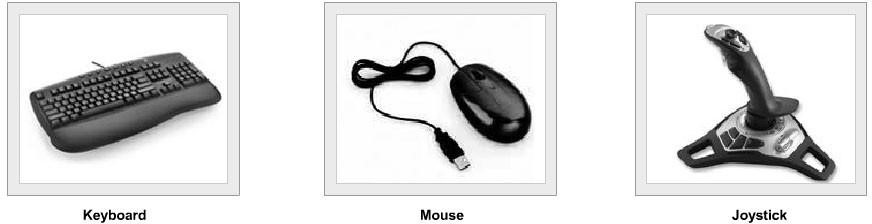
Character device doesn't have any seek operation.

There are too many character devices present in a computer system such as printer, mice, rats, network interfaces etc. These four are the common character devices.

## 5.2. Input/Output Devices Examples

Here are the list of some most popular and common input/output devices:

* Keyboard
* Mouse
* Monitor
* Modem
* Scanner
* Laser Printer



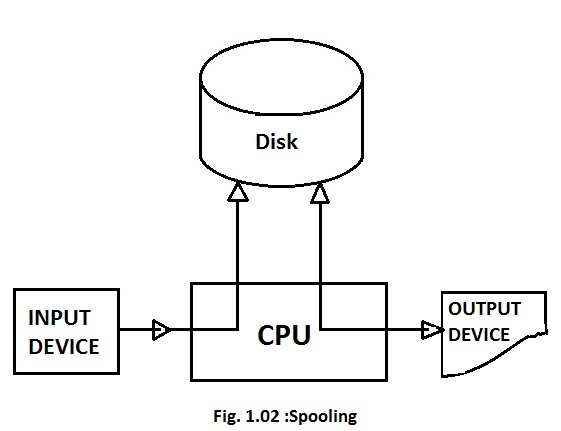
## 5.3. Buffering

A buffer in a computer environment means that a set amount of data is going to be stored in order to preload the required data right before it gets used by the CPU. Computers have many different devices that all operate at varying speeds, and a buffer is needed to act as a sort of temporary placeholder for everything that is interacting. This is done to keep everything running efficiently and without issues between all the devices, programs, and processes running at that time.

buffer is a region of memory used to temporarily hold data while it is being moved from one place to another.  
That would be the most simple yet sensible definition for a buffer irrespective of where it may appear. Computers often have different devices in it that work at different speeds. For example the RAM is much faster when compared to the Hard Disk.  Further the CPU of a computer is only capable of handling a specific amount of data in a given time.  
These and many other reasons make it  a need for operating systems to have Buffers or Temporary memory locations it can use. For example imagine that there are two different processes.  It can be tricky to transfer data between these processes as the processes may be at two different states at a given time.  
Let us say process A : Is sending a bitmap to the printer driver so that it can send it to the printer.  Unfortunately the driver is busy printing another page at that time. So until the driver is ready the OS stores the data in a buffer.  
The same concept is applied for other things like copying files to a USB drive, playing a  video, taking input from a IO device etc.

**5.4. Spooling**

Spooling is a process in which data is temporarily held to be used and executed by a device, program or the system. Data is sent to and stored in memory or other volatile storage until the program or computer requests it for execution.



SPOOL stands for simultaneous peripheral operation on-line. Actually what happens here is that, there is buffer for storing the data, usually the disk. The i/o devices can't match with the speed of a cpu. Hence, the output from the cpu will be stored in this spool(buffer) and the i/o devices can take the output from this buffer as and when required according to their speed. The cpu is hence not bound to this i/o device and can perform other operations. So, spooling keeps both the cpu and the i/o devices working at high rates without any waiting.

"Spool" is technically an acronym for simultaneous peripheral operations online.

Spooling works like a typical request queue or spool where data, instructions and processes from multiple sources are accumulated for execution later on. Generally, the spool is maintained on the computer’s physical memory, buffers or the I/O device-specific interrupts. The spool is processed in ascending order, working on the basis of a FIFO (first in, first out) algorithm.

The most common implementation of spooling can be found in typical input/output devices such as the keyboard, mouse and printer. For example, in printer spooling, the documents/files that are sent to the printer are first stored in the memory or printer spooler. Once the printer is ready, it fetches the data from that spool and prints it.

# 5.5. Difference between Spooling and Buffering

There are two ways by which Input/output subsystems can improve the performance and efficiency of the computer by using a memory space in main memory or on the disk and these two are spooling and buffering.

[Spooling](https://www.geeksforgeeks.org/what-exactly-spooling-is-all-about/)**–**  
Spooling stands for Simultaneous peripheral operation online. A spool is a similar to buffer as it holds the jobs for a device till the device is ready to accept the job. It considers disk as a huge buffer which can store as many jobs for the device till the output devices are ready to accept them.

Buffering –  
he main memory has an area called buffer that is used to store or hold the data temporarily that is being transmitted either between two devices or between a device or an application. Buffering is an act of storing data temporarily in the buffer. It helps in matching the speed of the data stream between the sender and receiver. If speed of the sender’s transmission is slower than receiver, then a buffer is created in main memory of the receiver, and it accumulates the bytes received from the sender and vice versa.

The basic difference between Spooling and Buffering is that Spooling overlaps the input/output of one job with the execution of another job while the buffering overlaps input/output of one job with the execution of the same job.

Differences between Spooling and Buffering –

* The key difference between spooling and buffering is that Spooling can handle the input/output of one job along with the computation of an another job at the same time while buffering handles input/output of one job along with its computation.
* Spooling is an stands for Simultaneous Peripheral Operation online. Whereas buffering is not an acronym.
* Spooling is more efficient than buffering, as spooling can overlap processing two jobs at a time.
* Buffering use limited area in main memory while Spooling uses the disk as a huge buffer.

**Comparison chart –**

|  | SPOOLING | BUFFERING |
| --- | --- | --- |
| Basic Difference | It overlap the input/output of one job with the execution of another job. | It overlaps the input/output of one job with the execution of the same job. |
| Full form (stands for) | Simultaneous peripheral operation online | No full form |
| Efficiency | Spooling is more efficient than buffering. | Buffering is less efficient than spooling. |
| Consider Size | It considers disk as a huge spool or buffer. | Buffer is a limited area in main memory. |

# Unit 6. (File Management)

**6.1. Introduction:** A file is a collection of related information that is recorded on secondary storage. Or file is a collection of logically related entities. From user’s perspective a file is the smallest allotment of logical secondary storage.

6.2. Components of a file:

A file's attributes vary from one operating system to another but typically consist of these:

* Name: Name is the symbolic file name and is the only information kept in human readable form.
* Identifier: This unique tag is a number that identifies the file within the file system; it is in non-human-readable form of the file.
* Type: This information is needed for systems which support different types of files or its format.
* Location: This information is a pointer to a device which points to the location of the file on the device where it is stored.
* Size: The current size of the file (which is in bytes, words, etc.) which possibly the maximum allowed size gets included in this attribute.
* Protection: Access-control information establishes who can do the reading, writing, executing, etc.
* Date, Time & user identification: This information might be kept for the creation of the file, its last modification and last used. These data might be useful for in the field of protection, security, and monitoring its usage.

6.3. Operation performed on the file

* A file is an abstract data type. To define a file properly, we need to consider the operation performed on the file.
* Operating System can provide system calls to create, write, read, reposition, delete and truncate files.

## Creating a File

* Creation of a file is the first operation. For creating a file two steps are necessary.

1. *Required space must be allocated.*
2. *An entry for new file must be made in the directory.*

* The directory entry records the name of the file and the location in the file system.

## Writing a File

* For file writing operation, we make a system call specifying both the name of a file and the information to be written into the file.
* System searches the entire directory structure to find the location of the specified file.
* System keeps a write pointer that keeps track of writing at the location in the file.
* The system updates write pointer each time whenever a file write operation occurs.

## Reading a File

* To perform file read operation, we use a system call that specifies name of the file and the block of the file to read from it.
* Again the directory is searched for the specified file.
* System keeps a read pointer that keeps track of reading location in the file.
* The system updates read pointer each time whenever a file read operation occurs

## Repositioning Within a File

* Repositioning within a file operation does not involve any actual input output.
* The directory is searched for the appropriate entry and the current file position is set to a given value.
* It is also known as files seek operation.

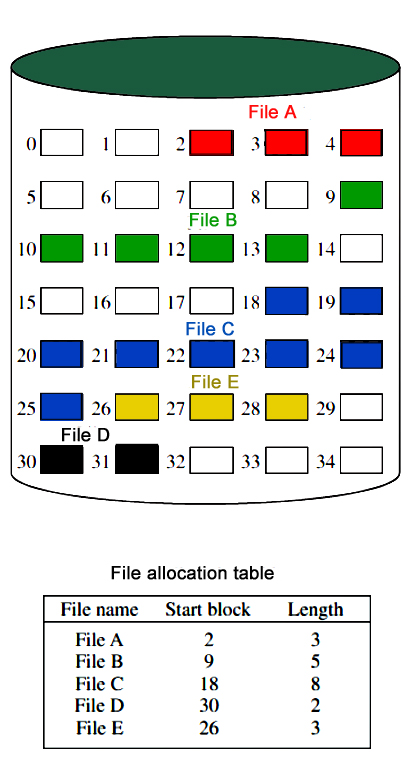
## Deleting a File

* File deletion operation also requires searching of a specified file entry within the directory structure.
* As soon as the file is deleted, space allocated to that file becomes available for further use.

## Truncating a File

* In some cases the user may want to erase the file contents but keep its attributes as it is. This operation is called *truncating a file*.
* Instead of delete a file and recreating it with same attributes, this function allows all attributes to remain unchanged except the file content.
* File length attribute is reset to a length zero and its file space is released.

**6.4. FILE ALLOCATION METHODS**

**6.4.1. Continuous Allocation:** A single continuous set of blocks is allocated to a file at the time of file creation. Thus, this is a pre-allocation strategy, using variable size portions. The file allocation table needs just a single entry for each file, showing the starting block and the length of the file. This method is best from the point of view of the individual sequential file. Multiple blocks can be read in at a time to improve I/O performance for sequential processing. It is also easy to retrieve a single block. For example, if a file starts at block b, and the ith block of the file is wanted, its location on secondary storage is simply b+i-1.  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/directory-file-allocation.jpg)

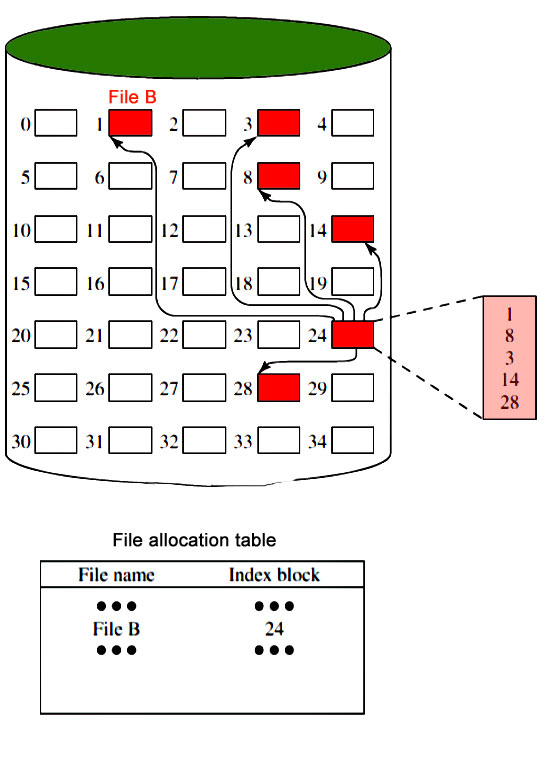
**Disadvantage**

* External fragmentation will occur, making it difficult to find contiguous blocks of space of sufficient length. Compaction algorithm will be necessary to free up additional space on disk.
* Also, with pre-allocation, it is necessary to declare the size of the file at the time of creation.

**6.4.2. Linked Allocation(Non-contiguous allocation) :** Allocation is on an individual block basis. Each block contains a pointer to the next block in the chain. Again the file table needs just a single entry for each file, showing the starting block and the length of the file. Although pre-allocation is possible, it is more common simply to allocate blocks as needed. Any free block can be added to the chain. The blocks need not be continuous. Increase in file size is always possible if free disk block is available. There is no external fragmentation because only one block at a time is needed but there can be internal fragmentation but it exists only in the last disk block of file.

**Disadvantage:**

* Internal fragmentation exists in last disk block of file.
* There is an overhead of maintaining the pointer in every disk block.
* If the pointer of any disk block is lost, the file will be truncated.
* It supports only the sequencial access of files.

**6.4.3. Indexed Allocation:**  
It addresses many of the problems of contiguous and chained allocation. In this case, the file allocation table contains a separate one-level index for each file: The index has one entry for each block allocated to the file. Allocation may be on the basis of fixed-size blocks or variable-sized blocks. Allocation by blocks eliminates external fragmentation, whereas allocation by variable-size blocks improves locality. This allocation technique supports both sequential and direct access to the file and thus is the most popular form of file allocation.  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/directory-indexing.jpg)

**Unit 7. (Linux Operating System)**

**7.1. Introduction:** An operating system is an interface between the user of a computer and the computer hardware. It is a collection of software that manages computer hardware resources and offers common services for programs of the computer. The short term of the operating system is OS. And, it is, an essential component of the system software in a computer system. The main purpose of an OS is to afford an environment in which a user can execute a program in an efficient or convenient manner.

Linux operating system is one of the popular versions of the UNIX operating system, which is designed to offer a free or low cost operating system for personal computer users. It gained the reputation as a fast performing and very efficient system.This is a remarkably complete operating system, including a GUI (graphical user interface), TCP/IP, the Emacs editor,can X Window System, etc.

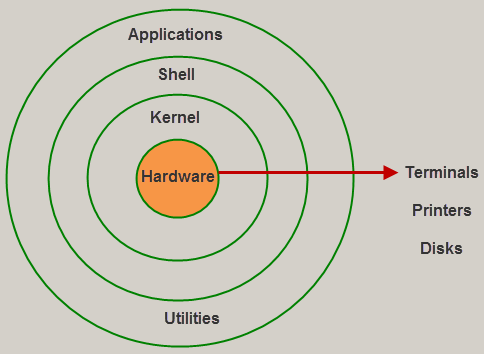
## 7.2. The History of LINUX Operating System

The History of Linux began in the 1991 with the beginning of a personal project by a Finland  student Linus Torvalds to create a new free operating system kernel. Since then, the resulting Linux Kernel has been marked by constant growth throughout the history.

* In the year 1991, Linux was introduced by a Finland student Linus Torvalds.
* Hewlett Packard UniX(HP-UX) 8.0 was released.
* In the year 1992, Hewlett Packard 9.0 was released.
* In the year 1993, NetBSD 0.8 and FreeBSD 1.0 released.
* In the year 1994, Red Hat Linux was introduced, Caldera was founded by Bryan Sparks and Ransom Love and NetBSD1.0 Released.
* In the year 1995, FreeBSD 2.0 and HP UX 10.0 was released.
* In the year 1996, K Desktop Environment was developed by Matthias Ettrich.
* In the year 1997, HP-UX 11.0 was released.
* In the year 1998, the fifth generation of SGI Unix i.e IRIX 6.5 , Sun Solaris 7 operating system and Free BSD 3.0 was released.
* In the year 2000, the agreement of Caldera Systems with SCO server software division and the professional services division was announced.
* In the year 2001, Linus Torvalds released the Linux 2.4 version source code.
* In the year 2001, Microsoft filed a trademark suit against Lindows.com
* In the year 2004, Lindows name was changed to Linspire.
* In the year 2004, the first release of Ubuntu was released.
* In the year 2005, The project, openSUSE began a free distribution from Novell’s community.
* In the year 2006, Oracle released its own distribution of Red Hat.
* In the year 2007, Dell started distributing laptops with Ubuntu pre installed in it.
* In the year 2011, Linux kernel 3.0 version was released.
* In the year 2013,  Googles Linux based Android claimed 75% of the smartphone market share, in terms of the number of phones shipped.
* In the year 2014, Ubuntu claimed 22,000,000 users.

### 7.3. Linux System Architecture

The Linux Operating System’s architecture primarily has these components: the Kernel, Hardware layer, System library, Shell and System utility.



1. The kernel is the core part of the operating system, which is  responsible for all the major activities of the LINUX operating system. This operating system consists of [different modules](https://www.elprocus.com/different-types-of-memory-modules-used-embedded-system/) and interacts directly with the underlying hardware. The kernel offers the required abstraction to hide  application programs or low-level hardware details to the system.

2. System libraries are special functions, that are used to implement the functionality of the operating system and do not require code access rights of kernel modules.

3. System Utility programs are liable to do individual, and specialized-level tasks.

4. Hardware layer of the LINUX operating system consists of peripheral devices such as RAM, HDD, CPU.

5. The shell is an interface between the user and the kernel, and it affords services of the kernel. It takes commands from the user and executes kernel’s functions. The Shell is present in different types of operating systems, which are classified into two types:command line shells and graphical shells.

The command line shells provide a command line interface, while the graphical line shells provide a graphical user interface. Though both shells perform operations, but the graphical user interface shells perform slower than the command line interface shells. Types of shells are classified into four:

* Korn shell
* Bourne shell
* C shell
* POSIX shell

**7.4. Commands of Linux**

## 1. ls

The ls command - the list command - functions in the [Linux terminal](http://www.informit.com/store/linux-kernel-development-9780672329463) to show all of the major directories filed under a given file system. For example, the command:

ls /applications

...will show the user all of the folders stored in the overall applications folder.

The ls command is used for viewing files, folders and directories.

## 2. cd

The cd command - change directory - will allow the user to change between file directories. As the name command name suggest, you would use the cd command to circulate between two different directories. For example, if you wanted to change from the home directory to the A directory, you would input the following command:

cd/A/applications

As you might have noted, the path name listed lists in reverse order. Logically cd/A/applications reads change to the A directory which is stored in the applications directory. All Linux commands follow a logical path.

## 3. mv

The mv command - move - allows a user to move a file to another folder or directory. Just like dragging a file located on a PC desktop to a folder stored within the "Documents" folder, the mv command functions in the same manner. An example of the mv command is:

mv/a/applications/majorapps /a/applications/minorapps

The first part of the command mv/arora/applications/majorapps lists the application to be moved. In this case, a. The second part of the command /a/applications/minorapps lists where a will be moved to - from majorapps to minorapps.

## 4. man

The man command - the manual command - is used to show the manual of the inputted command. Just like a film on the nature of film, the man command is the meta command of the [Linux CLI](http://www.informit.com/articles/article.aspx?p=1339466). Inputting the man command will show you all information about the command you are using. An example:

man cd

The inputting command will show the manual or all relevant information for the change directory command.

## 5. mkdir

The mkdir - make directory - command allows the user to make a new directory. Just like making a new directory within a PC or Mac desktop environment, the mkdir command makes new directories in a Linux environment. An example of the mkdir command

mkdir testdirectory

The example command made the directory "testdirectory".

## 6. rmdir

The rmdir - remove directory - command allows the user to remove an existing command using the Linux CLI. An example of the rmdir command:

rmdir testdirectory

The example command removed the directory "testdirectory".

It should be noted: both the mkdir and rmdir commands make and remove directories. They do not make files and they will also not remove a directory which has files in it. The mkdir will make an empty directory and the rmdir command will remove an empty directory.

7. touch

The touch command - a.k.a. the make file command - allows users to make files using the Linux CLI. Just as the mkdir command makes directories, the touch command makes files. Just as you would make a .doc or a .txt using a PC desktop, the touch command makes empty files. An example of the touch command:

touch testfile.txt

The example touch command effectively created the file testfile.txt. As noted by the extension, the file created is a .txt or text file. To equate, a .txt file in Linux is akin to a .txt notebook file within a Windows or Mac OS.

8. rm

The rm command - remove - like the rmdir command is meant to remove files from your Linux OS. Whereas the rmdir command will remove directories and files held within, the rm command will delete created files. An example of the rm command:

rm testfile.txt

The aforementioned command removed testfile.txt. Interestingly, whereas the rmdir command will only delete an empty directory, the rm command will remove both files and directories with files in it. This said, the rm command carries more weight than the rmdir command and should be used with more specificity.

9. locate

The locate - a.k.a. find - command is meant to find a file within the Linux OS. If you don't know the name of a certain file or you aren't sure where the file is saved and stored, the locate command comes in handy. A locate command example:

locate -i \*red\*house\*\*city\*

The stated command will locate an file with the a file name containing "Red", "House" and "City". A note on the input: the use of "-i" tells the system to search for a file unspecific of capitalization (Linux functions in lower case). The use of the asterik "\*" signifies searching for a wildcard. A wildcard tells the system to pull any and all files containing the search criteria.

By specifying -i with wildcards, the locate CLI command will pull back all files containing your search criteria effectivley casting the widest search net the system will allow.

10. clear

The clear command does exactly what it says. When your Linux CLI gets all mucked up with various readouts and information, the clear command clears the screen and wipes the board clean. Using the clear command will take the user back to the start prompt of whatever directory you are currently operating in. To use the clear command simply type clear.

QUESTIONS

1. Define operating system.
2. Write any two features of LinuX OS.
3. What is the purpose of pwd command.
4. Why should one use windows operating system?
5. Define home directory.
6. Define absolute path.
7. What are wild cards.
8. Which command is used to change file permission?
9. Why the '–r' option of 'rm' is dangerous?
10. Define open source.
11. What is a loop?
12. Define kernel.
13. What is 'Vi' editor?
14. What is 'kill' command?
15. What is input redirection?
16. Define tail command.
17. What is the purpose of 'who' command?
18. How will you start & quit Vi editor?
19. Describe various Linux file types.
20. What is shell script? Why we need it?
21. Explain various file permissions in Linux.
22. How do we count the number of words, characterS & lines in a specified file?
23. Write short note on encryption?
24. What is significance of 'chmod' command?
25. Define the responsibilities of system administrator.
26. Compare Linux and Unix.
27. Define the following commands. a) Mail b) News
28. Differentiate between write and wall command.
29. How files are created renamed and deleted in windows?
30. Differentiate between while and until loop?
31. Write a shell script to find factorial of a number.
32. Explain about following five commands. i) mv ii) mkdir iii) merge iv) grep v) whoami
33. How does system administrator add new groups, delete groups and modify groups.
34. Explain the file access permissions available in Linux to secure files & folders.
35. Explain about pros and cons of Linux operating system in detail.
36. Explain the main purpose of an operating system?
37. What are real-time systems?
38. What is a virtual memory?
39. Describe the objective of multiprogramming.
40. What is time- sharing system?
41. Briefly explain FCFS.
42. What is RR scheduling algorithm?
43. What are necessary conditions which can lead to a deadlock situation in a system?
44. What is the basic function of paging?
45. What is fragmentation?
46. Give an example of a Process State.
47. When designing the file structure for an operating system, what attributes are considered?
48. What is multitasking?
49. What is spooling?
50. What are interrupts?
51. What is GUI?
52. What is preemptive multitasking?